Logical Methods in Computer Science
Vol. 3 (4:11) 2007, pp. 1–36
www.lmcs-online.org

Submitted Jul. 21, 2007
Published Nov. 19, 2007

GENERIC TRACE SEMANTICS VIA COINDUCTION *

ICHIRO HASUO [a], BART JACOBS [b], AND ANA SOKOLOVA [c]

[a] Institute for Computing and Information Sciences, Radboud University Nijmegen, the Netherlands and Research Institute for Mathematical Sciences, Kyoto University, Japan
URL: http://www.cs.ru.nl/~ichiro

[b] Institute for Computing and Information Sciences, Radboud University Nijmegen, the Netherlands
URL: http://www.cs.ru.nl/~bart

[c] Department of Computer Sciences, University of Salzburg, Austria
e-mail address: anas@cs.uni-salzburg.at

Abstract. Trace semantics has been defined for various kinds of state-based systems, notably with different forms of branching such as non-determinism vs. probability. In this paper we claim to identify one underlying mathematical structure behind these "trace semantics," namely coinduction in a Kleisli category. This claim is based on our technical result that, under a suitably order-enriched setting, a final coalgebra in a Kleisli category is given by an initial algebra in the category **Sets**. Formerly the theory of coalgebras has been employed mostly in **Sets** where coinduction yields a finer process semantics of bisimilarity. Therefore this paper extends the application field of coalgebras, providing a new instance of the principle "process semantics via coinduction."

1. Introduction

Trace semantics is a commonly used semantic relation for reasoning about state-based systems. Trace semantics for labeled transition systems is found on the coarsest edge of the linear time-branching time spectrum [57]. Moreover, trace semantics is defined for a variety of systems, among which are probabilistic systems [49].

In this paper we claim that these various forms of "trace semantics" are instances of a general construction, namely coinduction in a Kleisli category. Our point of view here is categorical, coalgebraic in particular. Hence this paper demonstrates the abstraction power

1998 ACM Subject Classification: F.3.1, F.3.2, G.3.

Key words and phrases: coalgebra, category theory, trace semantics, monad, Kleisli category, process semantics, non-determinism, probability.

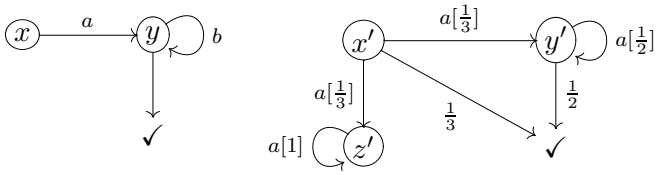
* Earlier versions [16,17] of this paper have been presented at the 1st International Conference on Algebra and Coalgebra in Computer Science (CALCO 2005), Swansea, UK, September 2005, and at the 8th International Workshop on Coalgebraic Methods in Computer Science (CMCS 2006), Vienna, Austria, March 2006.

[a] Supported by PRESTO research promotion program, Japan Science and Technology Agency.
[b] Also part-time at Technical University Eindhoven, the Netherlands.
[c] Supported by the Austrian Science Fund (FWF) project P18913-N15.

DOI:10.2168/LMCS-3 (4:11) 2007

of categorical/coalgebraic methods in computer science, uncovering basic mathematical structures underlying various concrete examples.

1.1. **"Trace semantics" in various contexts.** First we motivate our contribution through examples of various forms of "trace semantics." Think of the following three state-based, branching systems.



A context-free grammar (for Peano Arithmetic)

Terminal symbols:	$0, \mathsf{s}$
Non-terminal symbol:	$\mathbf{T}$
Generation rules:	
$\mathbf{T} \to 0$	
$\mathbf{T} \to \mathsf{s}\mathbf{T}$	

(1.1)

- The first one is a non-deterministic system with a special state ✓ denoting successful termination. To its state x we can assign its *trace set*:
$$\mathsf{tr}(x) = \{a, ab, abb, \dots\} = ab^* \ , \tag{1.2}$$
that is, the *set* of the possible linear-time behavior (namely words) that can arise through an execution of the system.[1] In this case the trace set $\mathsf{tr}(x)$ is also called the *accepted language*; formally it is defined (co)recursively by the following equations. For an arbitrary state x,
$$\begin{array}{rcl} \langle\rangle \in \mathsf{tr}(x) & \iff & x \to \checkmark \\ a \cdot \sigma \in \mathsf{tr}(x) & \iff & \exists y.\, (\, x \xrightarrow{a} y \ \wedge \ \sigma \in \mathsf{tr}(y)\,) \end{array} \tag{1.3}$$
Here $\langle\rangle$ denotes the empty word; $\sigma = a_1 a_2 \dots a_n$ is a word.
- The second system has a different type of branching, namely probabilistic branching. Here $x' \xrightarrow{a[1/3]} y'$ denotes: at the state x', a transition to y' outputting a occurs with probability $1/3$. Now, to the state x', we can assign its *trace distribution*:
$$\mathsf{tr}(x) = \left[\begin{array}{llll} \langle\rangle \mapsto \frac{1}{3}, & a \mapsto \frac{1}{3}\cdot\frac{1}{2}, & a^2 \mapsto \frac{1}{3}\cdot\frac{1}{2}\cdot\frac{1}{2}, & \cdots \\ a^n \mapsto \frac{1}{3}\cdot\left(\frac{1}{2}\right)^n, & \cdots & & \end{array} \right] \ , \tag{1.4}$$
that is, the probability distribution over the set of linear-time behavior.[2] Its formal (corecursive) definition is as follows.
$$\begin{array}{rcl} \mathsf{tr}(x)(\langle\rangle) & = & \mathsf{Pr}(x \to \checkmark) \ , \\ \mathsf{tr}(x)(a \cdot \sigma) & = & \sum_{y \in X} \mathsf{Pr}(x \xrightarrow{a} y) \cdot \mathsf{tr}(y)(\sigma) \ , \end{array} \tag{1.5}$$
where $\mathsf{Pr}(\dots)$ denotes the probability of a transition.
- The third example can be thought of as a state-based system, with non-terminal symbols as states. It is non-deterministic because a state $\mathbf{T}$ has two possible transitions. It is natural to call the following set of parse-trees its "trace semantics."
$$\mathsf{tr}(\mathbf{T}) = \left\{ \begin{array}{cccc} \mathsf{0} & \mathsf{s} & \mathsf{s} & \\ & \mathsf{0} & \mathsf{s} & \cdots \\ & & \mathsf{0} & \end{array} \right\}$$

[1] The infinite trace ab^ω is out of our scope here: we will elaborate this point later in Section 4.2.

[2] Here again, we do not consider the infinite trace $a^\omega \mapsto 1/3$.

It is again a set of "linear-time behavior" as in the first example, although the notion of linear-time behavior is different here. Linear-time behavior—that is, what we observe after we have resolved all the non-deterministic branchings in the system—is now a parse-tree instead of a word.

1.2. **Coalgebras and coinduction.** In recent years the theory of *coalgebras* has emerged as the "mathematics of state-based systems" [25,26,47]. In the categorical theory of coalgebras, an important definition/reasoning principle is *coinduction*: a system (identified with a coalgebra $c : X \to FX$) is assigned a unique morphism beh_c into the final coalgebra.

$$\begin{array}{ccc} FX & \overset{F(\mathsf{beh}_c)}{- - - - - - - \to} & FZ \\ {\scriptstyle c}\uparrow & & \cong\uparrow{\scriptstyle \text{final}} \\ X & \underset{\mathsf{beh}_c}{- - - - - - - - \to} & Z \end{array}$$

The success of coalgebras is largely due to the fact that, when **Sets** is taken as the base category, the final coalgebra semantics is fully abstract with respect to the conventional notion of *bisimilarity*: for states x and y of coalgebras $X \xrightarrow{c} FX$ and $Y \xrightarrow{d} FY$,

$$\mathsf{beh}_c(x) = \mathsf{beh}_d(y) \qquad \Longleftrightarrow \qquad x \text{ and } y \text{ are bisimilar.}$$

This is the case for a wide variety of systems (i.e. for a variety of functors F), hence *coinduction in* **Sets** *captures bisimilarity*.

However, there is not so much work so far that captures other behavioral equivalences (coarser than bisimilarity) by the categorical principle of coinduction. The current work—capturing trace semantics by coinduction in a Kleisli category—therefore extends the application field of the theory of coalgebras.

1.3. **Our contributions.** Our technical contributions are summarized as follows. Assume that T is a monad on **Sets** which has a suitable order structure; we shall denote its Kleisli category by $\mathcal{K}\ell(T)$.

- *Trace semantics via coinduction in a Kleisli category.* Commutativity of the coinduction diagram

$$\begin{array}{ccc} \overline{F}X & \overset{\overline{F}(\mathsf{tr}_c)}{- - - - - - - \to} & \overline{F}Z \\ {\scriptstyle c}\uparrow & & \cong\uparrow{\scriptstyle \text{final}} \\ X & \underset{\mathsf{tr}_c}{- - - - - - - \to} & Z \end{array} \qquad \text{in} \quad \begin{array}{l} \mathcal{K}\ell(T), \\ \text{the Kleisli category for } T \end{array} \tag{1.6}$$

  is shown to be equivalent to the conventional recursive definition of trace semantics such as (1.3) and (1.5). This is true for both trace set semantics (for non-deterministic systems) and trace distribution semantics (for probabilistic systems). The induced arrow tr_c thus gives (conventional) trace semantics for a system c.
- *Identification of the final coalgebra in a Kleisli category.* We show that

an initial algebra in **Sets**
coincides with
a final coalgebra in $\mathcal{K}\ell(T)$.

In particular, the final coalgebra in **Rel** is the initial algebra in **Sets**, because the category **Rel** of sets and relations is a Kleisli category for a suitable monad. This coincidence happens in the following two steps:
- the initial algebra in **Sets** lifts to a Kleisli category, due to a suitable adjunction-lifting result;
- in a Kleisli category we have *initial algebra-final coalgebra coincidence*. Here we use the classical result by Smyth and Plotkin [51], namely *limit-colimit coincidence* which is applicable in a suitably order-enriched category.

Note the presence of two parameters in (1.6): a monad T and an endofunctor F, both on **Sets**. The monad T specifies the *branching type* of systems. We have three leading examples:[3]

- the powerset monad $\mathcal{P}$ modeling *non-deterministic* or *possibilistic* branching;
- the subdistribution monad $\mathcal{D}$
$$\mathcal{D}X = \{d : X \to [0,1] \mid \sum_{x \in X} d(x) \leq 1\}$$
modeling *probabilistic* branching; and
- the lift monad $\mathcal{L} = 1 + (\_)$ modeling system with *exception* (or *deadlock*, *non-termination*).

The functor F specifies the *transition type* of systems: our understanding of "transition type" shall be clarified by the following examples.

- In labeled transition systems (LTSs) with explicit termination—no matter if they are non-deterministic or even probabilistic—a state either
  - terminates ($x \to \checkmark$), or
  - outputs one symbol and moves to another state ($x \xrightarrow{a} x'$),

  in one transition. This "transition type" is expressed by the functor $FX = 1 + \Sigma \times X$, where Σ is the output alphabet and $1 = \{\checkmark\}$.
- In context-free grammars (CFGs) as state-based systems, a state evolves into a sequence of terminal and non-terminal symbols in a transition. The functor
$$FX = (\Sigma + X)^*$$
with Σ being the set of terminal symbols, expresses this transition type.

Clear separation of branching and transition types is important in our generic treatment of trace semantics. The transition type F determines the set of linear-time behavior (which is in fact given by the initial F-algebra in **Sets**). We model a system by a coalgebra $X \xrightarrow{c} \overline{F}X$ in the Kleisli category $\mathcal{K}\ell(T)$—see (1.6)—where $\overline{F}$ is a suitable lifting of F in $\mathcal{K}\ell(T)$. By the definition of a Kleisli category we will easily see the following bijective correspondence.

$$\frac{X \xrightarrow{c} \overline{F}X \ \text{ in } \mathcal{K}\ell(T)}{X \xrightarrow{c} TFX \ \text{ in } \mathbf{Sets}}$$

Hence our system—a *function* of the type $X \to TFX$—first resolves a branching of type T and then makes a transition of type F. Many branching systems allow such representation so that our generic coalgebraic trace semantics applies to them.

[3]Other examples include the monad $X \mapsto (\mathbb{N} \cup \{\infty\})^X$ for multisets, the monad $X \mapsto [0,\infty]^X$ for real valuations, and the monad $X \mapsto \mathcal{P}(M \times \_)$ with a monoid M for timed systems (cf. [29]). These monads can be treated in a similar way as our leading examples. We leave out the details.

1.4. **Generic theory of traces and simulations.** In the study of coalgebras as 'categorical presentation of state-based systems', there are three ingredients playing crucial roles: *coalgebras* as systems; *coinduction* yielding process semantics; and *morphisms of coalgebras* as behavior-preserving maps. In this paper we study the first two in a Kleisli category. What about morphisms of coalgebras?

In [14] this question is answered by identifying *lax/oplax morphisms of coalgebras* in a Kleisli category as *forward/backward simulations*. Use of traces and simulations is a common technique in formal verification of systems (see e.g. [41]): a desirable property is expressed in terms of traces; and then a system is shown to satisfy the property by finding a suitable simulation. Therefore this paper, together with [14], forms an essential part of developing a "generic theory of traces and simulations" using coalgebras in a Kleisli category. The categorical genericity—especially the fact that we can treat non-deterministic and probabilistic branching in a uniform manner—is exploited in [19] to obtain a simulation-based proof method for a probabilistic notion of anonymity for network protocols. Currently we are investigating how much more applicational impact can be brought about by our generic theory of traces and simulations.

1.5. **Testing and trace semantics.** Since the emergence of the theory of coalgebras, the significance of *modal logics* as specification languages has been noticed by many authors. This is exemplified by the slogan in [36]: 'modal logic is to coalgebras what equational logic is to algebras'. Inspired by coalgebras on Stone spaces and the corresponding modal logic, recent developments [5, 6, 31, 32, 34, 37, 45] have identified the following situation as the essential mathematical structure underlying modal logics for coalgebras.

$$F^{\mathrm{op}} \circlearrowleft \mathbb{C}^{\mathrm{op}} \underset{S^{\mathrm{op}}}{\overset{P}{\underset{\longleftarrow}{\overset{\longrightarrow}{\top}}}} \mathbb{A} \circlearrowright M \qquad \text{together with} \quad MP \overset{\delta}{\Longrightarrow} PF^{\mathrm{op}}$$

In fact, it is noticed in [45] that such a situation not only hosts a modal logic but also a more general notion of *testing* (in the sense of [53, 57], also called *testing scenarios*). Therefore we shall call the above situation a *testing situation*.

In the last technical section of the paper we investigate coalgebraic trace semantics for the special case $T = \mathcal{P}$ (modeling non-determinism) from this testing point of view. First, we present some basic facts on testing situations, especially on the relationship between the induced *testing equivalence* and the final coalgebra semantics. These two process equivalences are categorically presented as kernel pairs, which enables a fairly simple presentation of the theory of coalgebraic testing. In addition, we observe that the coinduction scheme in the Kleisli category $\mathcal{K}\ell(\mathcal{P})$ gives rise to a canonical testing situation, in which the set of tests is given by an initial F-algebra.

The material on testing in the last section has not been presented in the earlier versions [16, 17] of this paper.

1.6. **Organization of the paper.** In Section 2 we observe that a coalgebra in a Kleisli category is an appropriate "denotation" of a branching system, when we focus on trace semantics. In Section 3 we present our main technical result that an initial algebra in **Sets** yields a final coalgebra in $\mathcal{K}\ell(T)$. The relationship to axiomatic domain theory—which employs similar mathematical arguments—is also discussed here. Section 4 presents some examples of the use of coinduction in $\mathcal{K}\ell(T)$ and argues that the coinduction principle is

a general form of trace semantics. In Section 5 we review the preceding material from the testing point of view.

2. Coalgebras in a Kleisli category

In the study of coalgebras as "categorical presentations of state-based systems," the category **Sets** of sets and functions has been traditionally taken as a base category (see e.g. [25, 47]). An important fact in such a setting is that bisimilarity is often captured by coinduction.[4]

However, bisimilarity is not the only process equivalence. In some applications one would like coarser equivalences, for example in order to abstract away internal branching structures. One of such coarser semantics, which has been extensively studied, is *trace equivalence*. For example, the process algebra CSP [21] has trace semantics as its operational model. Trace equivalence is coarser than bisimilarity, as the following classic example of "trace-equivalent but not bisimilar" systems illustrates.

It is first noticed in [46] that the Kleisli category for the powerset monad is an appropriate base category for trace semantics for non-deterministic systems. This observation is pursued further in [16, 17, 24]. In [15] it is recognized that the same is true for the subdistribution monad for probabilistic systems. The current paper provides a unified framework which yields those preceding results, in terms of **Cppo**-enrichment of a Kleisli category; see Section 2.3. In this section we first aim to justify the use of coalgebras in a Kleisli category.

2.1. Monads and Kleisli categories.

Here we recall the relevant facts about monads and Kleisli categories. For simplicity we exclusively consider monads on **Sets**.

A *monad* on **Sets** is a categorical construct. It consists of

- an endofunctor T on **Sets**;
- a *unit* natural transformation $\eta : \mathrm{id} \Rightarrow T$, that is, a function $X \stackrel{\eta_X}{\to} TX$ for each set X satisfying a suitable naturality condition; and
- a *multiplication* natural transformation $\mu : T^2 \Rightarrow T$, consisting of functions $T^2X \stackrel{\mu_X}{\to} TX$ with X ranging over sets.

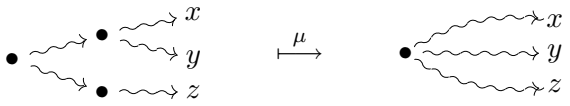
The unit and multiplication are required to satisfy the following compatibility conditions.

$$\begin{array}{ccccc} TX & \xrightarrow{\eta_{TX}} & T^2X & \xleftarrow{T\eta_X} & TX \\ & \searrow^{\mathrm{id}} & \downarrow{\scriptstyle \mu_X} & \swarrow_{\mathrm{id}} & \\ & & TX & & \end{array} \qquad \begin{array}{ccc} T^3X & \xrightarrow{T\mu_X} & T^2X \\ {\scriptstyle \mu_{TX}}\downarrow & & \downarrow{\scriptstyle \mu_X} \\ T^2X & \xrightarrow[\mu_X]{} & TX \end{array}$$

See [3, 42] for the details.

[4] Non-examples include LTSs with unbounded branching degree. They are modeled as coalgebras for $FX = \mathcal{P}(\Sigma \times X)$. Lambek's Lemma readily shows that this choice of F does not have a final coalgebra in **Sets**, because it would imply an isomorphism $Z \stackrel{\cong}{\to} \mathcal{P}(\Sigma \times Z)$ which is impossible for cardinality reasons.

The monad structures play a crucial role in modeling "branching." Intuitively, the unit η embeds a non-branching behavior as a trivial branching (with only one possibility to choose). The multiplication μ "flattens" two successive branchings into one branching, abstracting away internal branchings:



(2.1)

The following examples will illustrate how this flattening phenomenon is a crucial feature of trace semantics.

In this paper we concentrate on the three monads mentioned in the introduction: $\mathcal{L}$, $\mathcal{P}$ and $\mathcal{D}$.

- The lift monad $\mathcal{L} = 1 + (\_)$—where we denote $1 = \{\bot\}$ with $\bot$ meaning *deadlock*—has a standard monad structure induced by a coproduct. For example, the multiplication $\mu^{\mathcal{L}}_X : 1 + 1 + X \to 1 + X$ carries $x \in X$ to itself and both $\bot$'s to $\bot$.
- The powerset monad $\mathcal{P}$ has a unit given by singletons and a multiplication given by unions. The monad $\mathcal{P}$ models non-deterministic branching: the "flattening" in (2.1) corresponds to the following application of the multiplication of $\mathcal{P}$.

$$\begin{array}{ccc} \mathcal{P}\mathcal{P}X & \xrightarrow{\quad\mu^{\mathcal{P}}_X\quad} & \mathcal{P}X \\ \{\,\{x,y\},\{z\}\,\} & \longmapsto & \{x,y,z\} \end{array}$$

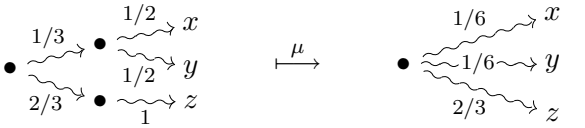
  The monad $\mathcal{P}$'s action on arrows (as a functor) is given by direct images: for $f : X \to Y$, the function $\mathcal{P}f : \mathcal{P}X \to \mathcal{P}Y$ carries a subset $u \subseteq X$ to the subset $\{f(x) \mid x \in u\} \subseteq Y$.
- The subdistribution monad $\mathcal{D}$ has a unit given by the *Dirac distributions*.

$$\begin{array}{ccc} X & \xrightarrow{\quad\eta^{\mathcal{D}}_X\quad} & \mathcal{D}X \\ x & \longmapsto & \left[\begin{array}{ll} x \mapsto 1 & \\ x' \mapsto 0 & (\text{for } x' \neq x) \end{array}\right] \end{array}$$

  Its multiplication is given by multiplying the probabilities along the way. That is,

$$\mu^{\mathcal{D}}_X(\xi) = \lambda x.\ \sum_{d \in \mathcal{D}X} \xi(d) \cdot d(x)\ ,$$

  which models "flattening" of the following kind.



  that is,

$$\left[\begin{array}{cc} \left[\begin{array}{l} x \mapsto 1/2 \\ y \mapsto 1/2 \end{array}\right] & \mapsto\ 1/3 \\ [z \mapsto 1] & \mapsto\ 2/3 \end{array}\right] \quad \overset{\mu}{\longmapsto} \quad \left[\begin{array}{l} x \mapsto 1/6 \\ y \mapsto 1/6 \\ z \mapsto 2/3 \end{array}\right] .$$

  The monad $\mathcal{D}$'s action on arrows (as a functor) is given as a suitable adaptation of "direct images." Namely, for $f : X \to Y$, the function $\mathcal{D}f : \mathcal{D}X \to \mathcal{D}Y$ carries $d \in \mathcal{D}X$ to $[y \mapsto \sum_{x \in f^{-1}(y)} d(x)] \in \mathcal{D}Y$.

Given any monad T, its *Kleisli category* $\mathcal{K}\ell(T)$ is defined as follows. Its objects are the objects of the base category, hence sets in our consideration. An arrow $X \to Y$ in $\mathcal{K}\ell(T)$ is the same thing as an arrow $X \to TY$ in the base category, here **Sets**.

$$\frac{X \longrightarrow Y \ \text{ in } \mathcal{K}\ell(T)}{X \longrightarrow TY \ \text{ in } \mathbf{Sets}}$$

Identities and composition of arrows are defined using the unit and the multiplication of T. Moreover, there is a canonical adjunction

$$\mathbf{Sets} \underset{K}{\overset{J}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathcal{K}\ell(T) \tag{2.2}$$

such that J carries $X \xrightarrow{f} Y$ in **Sets** to $X \xrightarrow{\eta_Y \circ f} Y$ in $\mathcal{K}\ell(T)$. See [3, 42] for details.

The relevance in this paper is that a Kleisli category can be thought of as a category where the branching is implicit. For example, an arrow $X \to Y$ in the Kleisli category $\mathcal{K}\ell(\mathcal{P})$ is a function $X \to \mathcal{P}Y$ hence a "non-deterministic function." When $T = \mathcal{D}$, then by writing $X \to Y$ in the Kleisli category we mean a function with probabilistic branching. Moreover, composition of arrows in $\mathcal{K}\ell(T)$ is given by

$$X \xrightarrow{f} Y \xrightarrow{g} Z \quad \text{in } \mathcal{K}\ell(T) \quad = \quad X \xrightarrow{f} TY \xrightarrow{Tg} T^2Z \xrightarrow{\mu_Z} TZ \quad \text{in } \mathbf{Sets};$$

that is, making one transition (by g) after another (by f), and then flattening (by μ_Z). For example, this general definition instantiates as follows when $T = \mathcal{D}$. For $X \xrightarrow{f} Y \xrightarrow{g} Z$,

$$(g \circ f)(x)(z) = \textstyle\sum_{y \in Y} f(x)(y) \cdot g(y)(z) \ .$$

Remark 2.1. Our use of the *sub*-distribution monad instead of the distribution monad

$$\mathcal{D}_{=1}(X) = \{d : X \to [0,1] \mid \sum_{x \in X} d(x) = 1\}$$

needs some justification. Looking at the trace distribution (1.4), one sees that the probabilities add up only to $2/3$ and not to 1; this is because the infinite trace (namely $a^\omega \mapsto 1/3$) are not present. Therefore in this example, although the state-based system can be modeled as a coalgebra in the category $\mathcal{K}\ell(\mathcal{D}_{=1})$, its trace semantics can only be expressed as an arrow in $\mathcal{K}\ell(\mathcal{D})$.

When a system is modeled as a coalgebra in $\mathcal{K}\ell(\mathcal{D})$, a state may have a (sub)distribution over possible transitions which adds up to less than 1. In that case the missing probability can be understood as the probability for *deadlock*.

Technically, we use the monad $\mathcal{D}$ instead of $\mathcal{D}_{=1}$ because we need the minimum element (a *bottom*) so that the Kleisli category becomes **Cppo**-enriched (Theorem 3.3). A bottom is available for $\mathcal{D}$ as the zero distribution $[x \mapsto 0]$, but not for $\mathcal{D}_{=1}$.

2.2. **Lifting functors by distributive laws.** In this paper a state-based system is presented as a coalgebra $X \to \overline{F}X$ in $\mathcal{K}\ell(T)$, where $\overline{F} : \mathcal{K}\ell(T) \to \mathcal{K}\ell(T)$ is a lifting of $F : \mathbf{Sets} \to \mathbf{Sets}$. This lifting $F \mapsto \overline{F}$ is equivalent to a *distributive law* $FT \Rightarrow TF$. The rest of this section elaborates on this point.

Various kinds of state-based, branching systems are expressed as a function of the form $X \xrightarrow{c} TFX$ with T a monad (for branching type) and F a functor (for transition type). The following examples are already hinted at in the introduction.

- For $T = \mathcal{P}$ and $F = 1 + \Sigma \times \_$, a function $X \xrightarrow{c} TFX$ is an LTS with explicit termination. For example, consider the following system

$$\begin{aligned} X &\longrightarrow \mathcal{P}(1 + \Sigma \times X) \\ x &\longmapsto \{\checkmark, (a_1, x_1), (a_2, x_2)\} \end{aligned}$$

  where $\checkmark$ is the element of 1.[5] Then the state x can make three possible transitions, namely: $x \to \checkmark$ (successful termination), $x \xrightarrow{a_1} x_1$, and $x \xrightarrow{a_2} x_2$, when written in a conventional way.
- By replacing $T = \mathcal{P}$ by $\mathcal{D}$, but keeping F the same, we obtain a probabilistic system such as the one in the middle of (1.1). For example,

$$\begin{aligned} X &\longrightarrow \mathcal{D}(1 + \Sigma \times X) \\ x' &\longmapsto \begin{bmatrix} (a, y') \mapsto 1/3 \\ (a, z') \mapsto 1/3 \\ \checkmark \mapsto 1/3 \end{bmatrix} . \end{aligned}$$

- For $T = \mathcal{P}$ and $F = (\Sigma + \_)^*$, a function $X \xrightarrow{c} TFX$ is a CFG with Σ the terminal alphabet (but without finiteness conditions e.g. on the state space). See [16] for more details.

All these systems are modeled by a function $X \xrightarrow{c} TFX$, hence an arrow $X \xrightarrow{c} FX$ in $\mathcal{K}\ell(T)$. Our question here is: is c a coalgebra in $\mathcal{K}\ell(T)$? In other words: is the functor F on **Sets** also a functor on $\mathcal{K}\ell(T)$?

Hence, to develop a generic theory of traces in $\mathcal{K}\ell(T)$, we need to lift F to a functor $\overline{F}$ on $\mathcal{K}\ell(T)$. A functor $\overline{F}$ is said to be a *lifting* of F if the following diagram commutes. Here J is the left adjoint in (2.2).

$$\begin{array}{ccc} \mathcal{K}\ell(T) & \xrightarrow{\overline{F}} & \mathcal{K}\ell(T) \\ {\scriptstyle J}\uparrow & & \uparrow{\scriptstyle J} \\ \mathbf{Sets} & \xrightarrow[F]{} & \mathbf{Sets} \end{array} \tag{2.3}$$

The following fact is presented in [43]; see also [39, 40]. Its proof is straightforward.

Lemma 2.2. *A lifting $\overline{F}$ of F is in bijective correspondence with a **distributive law** $\lambda : FT \Rightarrow TF$. A distributive law λ is a natural transformation which is compatible with T's monad structure, in the following way.*

$$\begin{array}{ccc} FX & \xrightarrow{F\eta_X} & FTX \\ & \searrow^{\eta_{FX}} & \downarrow{\scriptstyle \lambda_X} \\ & & TFX \end{array} \qquad\qquad \begin{array}{ccccc} FT^2X & \xrightarrow{\lambda_{TX}} & TFTX & \xrightarrow{T\lambda_X} & T^2FX \\ {\scriptstyle F\mu_X}\downarrow & & & & \downarrow{\scriptstyle \mu_{FX}} \\ FTX & & \xrightarrow[\lambda_X]{} & & TFX \end{array}$$

□

[5] Note that the singleton $1 = \{\checkmark\}$ here in $F = 1 + \Sigma \times \_$ has a different interpretation from $1 = \{\bot\}$ in $T = \mathcal{L} = 1 + \_$. The intuition is as follows. On the one hand, when an execution hits successful termination $\checkmark$, it yields its history of observations as its trace. On the other hand, when an execution hits deadlock $\bot$ then it yields no trace no matter what is the history before hitting $\bot$. This distinction will be made formal in Example 4.3.

A distributive law λ induces a lifting $\overline{F}$ as follows. On objects: $\overline{F}X = FX$. Given $f : X \to Y$ in $\mathcal{K}\ell(T)$, we need an arrow $\overline{F}f : FX \to FY$ in $\mathcal{K}\ell(T)$. Recall that f is a function $X \to TY$ in **Sets**; one takes $\overline{F}f$ to be the arrow which corresponds to the function

$$FX \xrightarrow{Ff} FTY \xrightarrow{\lambda_Y} TFY \quad \text{in } \mathbf{Sets} \ .$$

A distributive law specifies how a transition (of type F) "distributes" over a branching (of type T). Let us look at an example. For $T = \mathcal{P}$ and $F = 1 + \Sigma \times \_$ (the combination for LTSs with explicit termination), we have the following distributive law.

$$\begin{array}{ccc} 1 + \Sigma \times (\mathcal{P}X) & \xrightarrow{\quad\lambda_X\quad} & \mathcal{P}(1 + \Sigma \times X) \\ \checkmark & \longmapsto & \{\checkmark\} \\ (a, S) & \longmapsto & \big\{(a, x) \mid x \in S\big\} \end{array}$$

For example,

$$\bullet \xrightarrow{a} \begin{array}{l} x \\ y \\ z \end{array} \quad \overset{\lambda}{\longmapsto} \quad \bullet \begin{array}{l} \xrightarrow{a} x \\ \xrightarrow{a} y \\ \xrightarrow{a} z \end{array} \qquad \text{that is} \qquad \big(a, \{x, y, z\}\big) \overset{\lambda}{\longmapsto} \big\{(a, x), (a, y), (a, z)\big\} \ ,$$

where waving arrows $\rightsquigarrow$ denote branchings.

Throughout the paper we need the global assumption that a functor F has a lifting $\overline{F}$ on $\mathcal{K}\ell(T)$, or equivalently, that there is a distributive law $\lambda : FT \Rightarrow TF$. Now we present some sufficient conditions for existence of λ. In most examples one of these conditions holds.

First, take $T = \mathcal{P}$, in which case we have $\mathcal{K}\ell(\mathcal{P}) \cong \mathbf{Rel}$, the category of sets and binary relations. We can provide the following condition that uses relation liftings, whose definition is found [24].

Lemma 2.3 (From [24])**.** *Let $F\colon \mathbf{Sets} \to \mathbf{Sets}$ be a functor that preserves weak pullbacks. Then there exists a distributive law $\lambda\colon F\mathcal{P} \Rightarrow \mathcal{P}F$ given by*

$$\lambda_X(u) = \big\{\, v \in FX \mid (v, u) \in \mathrm{Rel}_F(\in_X) \,\big\} \ ,$$

where $u \in F\mathcal{P}X$ and $\mathrm{Rel}_F(\in_X) \subseteq FX \times F\mathcal{P}X$ is the F-relation lifting of the membership relation $\in_X$. □

In fact, the functor $\overline{F} : \mathbf{Rel} \to \mathbf{Rel}$ induced by this distributive law carries an arrow $R : X \to Y$ in $\mathcal{K}\ell(\mathcal{P})$—which is a binary relation between X and Y—to its F-relation lifting $\mathrm{Rel}_F(R)$. That is,

$$\overline{F}R = \mathrm{Rel}_F(R) \quad : \ FX \longrightarrow FY \tag{2.4}$$

in $\mathcal{K}\ell(\mathcal{P}) \cong \mathbf{Rel}$.

Now let us consider a monad T which is not $\mathcal{P}$. When a monad T is *commutative* and a functor F is *shapely*, we can provide a canonical distributive law. The class of such monads and functors is wide and all the examples in this paper are contained.

- A *commutative* monad [33] is intuitively a monad whose corresponding algebraic theory has only commutative operators. We exploit the fact that a commutative monad is equipped with an arrow called *double strength*

  $$\mathsf{dst}_{X,Y} : TX \times TY \longrightarrow T(X \times Y)$$

  for any sets X and Y; the double strength must be compatible with the monad structure of T in an obvious way.

Our three examples of monads are all commutative, with the following double strengths.

$$\begin{array}{lcl} \mathsf{dst}^{\mathcal{L}}_{X,Y}(u,v) & = & \begin{cases} (u,v) & \text{if } u \in X \text{ and } v \in Y, \\ \bot & \text{if } u = \bot \text{ or } v = \bot, \end{cases} \\ \mathsf{dst}^{\mathcal{P}}_{X,Y}(u,v) & = & u \times v \ , \\ \mathsf{dst}^{\mathcal{D}}_{X,Y}(u,v) & = & \lambda(x,y).\ u(x) \cdot v(y) \ . \end{array} \tag{2.5}$$

- The family of *shapely functors* [27][6] on **Sets** is defined inductively by the following BNF notation:
$$F ::= \mathrm{id} \mid \Sigma \mid F_1 \times F_2 \mid \coprod_{i \in I} F_i \ ,$$
where Σ denotes the constant functor into an arbitrary set Σ. Notice that taking infinite product is not allowed, nor exponentiation to the power of an infinite set. This is in order to ensure that we find an initial F-algebra as a suitable ω-colimit—see Proposition A.1.

Lemma 2.4. *Let $T\colon$ **Sets** $\to$ **Sets** be a commutative monad, and $F\colon$ **Sets** $\to$ **Sets** a shapely functor. Then there is a distributive law $\lambda\colon FT \Rightarrow TF$.*

Proof. The construction of a distributive law is done inductively on the construction of shapely F.

- If F is the identity functor, then the λ is the identity natural transformation $T \Rightarrow T$.
- If F is a constant functor, say $X \mapsto \Sigma$, then λ is the unit $\eta_\Sigma\colon \Sigma \to T\Sigma$ at $\Sigma \in$ **Sets**.
- If $F = F_1 \times F_2$ we use induction in the form of distributive laws $\lambda^{F_i}\colon F_iT \Rightarrow TF_i$ for $i \in \{1,2\}$ to form the composite:
$$F_1TX \times F_2TX \xrightarrow{\lambda^{F_1} \times \lambda^{F_2}} TF_1X \times TF_2X \xrightarrow{\mathsf{dst}} T(F_1X \times F_2X) \ .$$
- If F is a coproduct $\coprod_{i \in I} F_i$ then we use laws $\lambda^{F_i}\colon F_iT \Rightarrow TF_i$ for $i \in I$ in:
$$\coprod_{i \in I} F_i(TX) \xrightarrow{[T(\kappa_i) \circ \lambda^{F_i}]_{i \in I}} T(\coprod_{i \in I} F_iX) \ .$$

It is straightforward to check that such λ is natural and compatible with the monad structure. □

We have provided some *sufficient* conditions for a distributive law to exist, that is, for a functor F to be lifted to $\mathcal{K}\ell(T)$. This does not mean the results in the sequel hold exclusively for commutative monads and shapely functors.

2.3. **Order-enriched structures of Kleisli categories.** The notion of branching naturally involves a partial order: one branching is bigger than another if the former offers "more possibilities" than the latter. Formally, this order appears as the **Cppo**-*enriched structure* of a Kleisli category. It plays an important role in the initial algebra-final coalgebra coincidence in Section 3.1.

A **Cppo**-*enriched category* $\mathbb{C}$ is a category where:

- Each homset $\mathbb{C}(X,Y)$ carries a partial order $\sqsubseteq$ as in
$$X \underset{f}{\overset{g}{\rightrightarrows}} Y \quad (\sqsubseteq)$$

[6]Shapely functors here are called *polynomial* functors by some authors, although other authors allow infinite powers or the powerset construction.

which makes $\mathbb{C}(X,Y)$ an ω-cpo with a bottom. This means:
- for an increasing ω-chain of arrows from X to Y,
$$f_0 \sqsubseteq f_1 \sqsubseteq \dots \quad : \quad X \longrightarrow Y \ ,$$
there exists its join $\bigsqcup_{n<\omega} f_n : X \to Y$;
- for any X and Y there exists a bottom arrow $\bot_{X,Y} : X \to Y$ which is the minimum in $\mathbb{C}(X,Y)$.

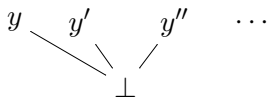
• Moreover, composition of arrows is continuous as a function $\mathbb{C}(X,Y)\times\mathbb{C}(Y,Z) \to \mathbb{C}(X,Z)$. This means that the following joins are preserved:[7]
$$g \circ \big(\bigsqcup_{n<\omega} f_n\big) \ = \ \bigsqcup_{n<\omega}(g \circ f_n) \quad \text{and} \quad \big(\bigsqcup_{n<\omega} f_n\big) \circ h \ = \ \bigsqcup_{n<\omega}(f_n \circ h) \ .$$
Note that composition need not preserve bottoms (i.e. it is not necessarily *strict*).

This is in fact an instance of a more general notion of $\mathbb{V}$-*enriched categories* where $\mathbb{V}$ is the category **Cppo** of pointed (i.e. with $\bot$) cpo's and continuous (but not necessarily strict) functions. See [7, 28, 38] for more details on enriched category theory, and [1] on cpo's and domain theory.

Lemma 2.5. *For our three examples $\mathcal{L}$, $\mathcal{P}$ and $\mathcal{D}$ of a monad T, the Kleisli category $\mathcal{K}\ell(T)$ is* **Cppo***-enriched. Moreover, composition of arrows is left-strict: $\bot \circ f = \bot$.*

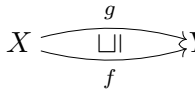
The left-strictness of composition will be necessary later.

Proof. Notice first that a set TY for $T \in \{\mathcal{L}, \mathcal{P}, \mathcal{D}\}$ carries a cpo structure with $\bot$. The set $\mathcal{L}Y = \{\bot\} + Y$ carries the flat order with a bottom:



embodying the idea that $\bot$ denotes non-termination or *deadlock*—in contrast to $\checkmark$ for successful termination. The set $\mathcal{P}Y$ carries an inclusion order; in $\mathcal{D}Y$ we define $d \sqsubseteq e$ if $d(y) \le e(y)$ for each $y \in Y$. The bottom element in $\mathcal{D}Y$ is the zero distribution $[y \mapsto 0]$: this belongs to the set $\mathcal{D}Y$ because $\mathcal{D}$ is the *sub*-distribution monad.

The cpo structure of a homset $\mathcal{K}\ell(T)(X,Y)$ comes from that of TY in a pointwise manner:



$$\text{if and only if} \qquad \forall x \in X.\ f(x) \sqsubseteq_{TY} g(x) \ .$$

It is laborious but straightforward to show that composition in $\mathcal{K}\ell(T)$ is continuous and left-strict. □

[7] This component-wise preservation of joins is equivalent to the continuity of the composition function. See [1, Lemma 3.2.6].

We are concerned with coalgebras $X \to \overline{F}X$ in the category $\mathcal{K}\ell(T)$, which we assume is **Cppo**-enriched. Hence it comes natural to require that functor $\overline{F}$ is somehow compatible with the **Cppo**-enriched structure of $\mathcal{K}\ell(T)$. The obvious choice is to require that $\overline{F}$ is a **Cppo**-*enriched functor* (see e.g. [7]), i.e. $\overline{F}$ is *locally continuous*. It means that for an increasing ω-chain $f_n : X \to Y$, we have

$$\overline{F}(\bigsqcup_{n<\omega} f_n) = \bigsqcup_{n<\omega} (\overline{F} f_n) \ .$$

This is indeed the assumption chosen in axiomatic domain theory. We will come back to this point later in Section 3.3. However, for our later purpose, we only need the weaker condition of *local monotonicity*: $f \sqsubseteq g$ implies $\overline{F}f \sqsubseteq \overline{F}g$.

For a monad $T = \{\mathcal{L}, \mathcal{P}, \mathcal{D}\}$ and a shapely functor F (recall Lemma 2.4), the lifted $\overline{F}$ is indeed locally continuous. We emphasize again that this does not mean our results in Section 3 hold exclusively for shapely functors.

Lemma 2.6. *Let F be a shapely functor and $T \in \{\mathcal{L}, \mathcal{P}, \mathcal{D}\}$. The lifting $\overline{F} : \mathcal{K}\ell(T) \to \mathcal{K}\ell(T)$ induced by Lemma 2.4 is locally continuous.*

Proof. By induction on the construction of shapely functors.

- $F = \mathrm{id}$, the identity functor. Then $\overline{F} = \mathrm{id}$ which satisfies the condition.
- $F = \Sigma$, a constant functor. Then $\overline{F}$ maps every arrow to the identity map on Σ in $\mathcal{K}\ell(T)$. This is obviously locally continuous.
- $F = F_1 \times F_2$. First notice that, for $f : X \to Y$ in $\mathcal{K}\ell(T)$, we obtain $\overline{F}f$ as the following composite in **Sets**.

$$\begin{array}{ccc} F_1X \times F_2X & \xrightarrow{\overline{F_1}f \times \overline{F_2}f} & TF_1Y \times TF_2Y \\ & \searrow^{\overline{F}f} & \downarrow{\scriptstyle \mathsf{dst}_{F_1Y, F_2Y}} \\ & & T(F_1Y \times F_2Y) \end{array}$$

  Because the order in $\mathcal{K}\ell(T)(FX, FY)$ is pointwise, it suffices to show the following: $\mathsf{dst} : TX \times TY \to T(X \times Y)$ is a continuous map between cpo's. It is easy to check that this is indeed the case. See (2.5).
- $F = \coprod_{j \in J} F_j$. For $f : X \to Y$ in $\mathcal{K}\ell(T)$, we obtain the map $\overline{F}f$ as the composite $[T\kappa_j]_{j \in J} \circ \coprod_{j \in J} \mathcal{K}\ell(F_j)(f)$ in **Sets**. Since the order on the homset is pointwise, it suffices to show that each $T\kappa_j : TF_jY \to T(\coprod_{j \in J} F_jY)$ is continuous. This is easy. □

3. Final coalgebra in a Kleisli category

In this section we shall prove our main technical result: the initial F-algebra in **Sets** yields the final $\overline{F}$-coalgebra in $\mathcal{K}\ell(T)$. It happens in the following two steps: first, the initial algebra in **Sets** is lifted to the initial algebra in $\mathcal{K}\ell(T)$; second we have the initial algebra-final coalgebra coincidence in $\mathcal{K}\ell(T)$. For the latter we use the classical result [51] of limit-colimit coincidence. This is where the **Cppo**-enriched structure of $\mathcal{K}\ell(T)$ plays a role.

In the proof we use two standard constructions: initial/final sequences [2] and limit-colimit coincidence [51]. The reader who is not familiar with these constructions is invited to look at Appendices A.1 and A.2 where we briefly recall them.

Remark 3.1. The proof of our main theorem (Theorem 3.3) can be simplified if we suitably strengthen the assumptions. First, if we assume local *continuity* of the lifted functor $\overline{F}$ (instead of local *monotonicity* that is assumed in our main theorem), then the initial algebra-final coalgebra coincidence follows from a standard result in axiomatic domain theory; see Section 3.3. Furthermore, for the special case $T = \mathcal{P}$ in which case $\mathcal{K}\ell(\mathcal{P}) \cong \mathbf{Rel}$, the initial algebra-final coalgebra coincidence is almost obvious due to the duality $\mathbf{Rel} \cong \mathbf{Rel}^{\mathrm{op}}$; see Section 3.2.

3.1. **The initial algebra in Sets is the final coalgebra in $\mathcal{K}\ell(T)$.** First, it is standard that an initial algebra in **Sets** is lifted to an initial algebra in $\mathcal{K}\ell(T)$. Such a phenomenon is studied for instance in [11,44] in the context of combining datatypes (modeled by an initial algebra) and effectful computations (modeled by a Kleisli category). For this result we do not need an order structure.

Proposition 3.2. *Let T be a monad and F be a endofunctor, both on a category $\mathbb{C}$. Assume that we have a distributive law $FT \Rightarrow TF$—or equivalently, we have a lifting $\overline{F}$ on $\mathcal{K}\ell(T)$. If F has an initial algebra $\alpha : FA \xrightarrow{\cong} A$ in $\mathbb{C}$, then*

$$J\alpha = \eta_A \circ \alpha \ : \quad \overline{F}A \longrightarrow A \qquad \text{in } \mathcal{K}\ell(T)$$

is an initial $\overline{F}$-algebra. Here J is the canonical Kleisli left adjoint as in (2.2).

We will use an instance of this result for $\mathbb{C} = \mathbf{Sets}$.

Proof. It follows from [20, Theorem 2.14] that a distributive law lifts the canonical Kleisli adjunction to an adjunction between the categories $\mathbf{Alg}(F)$ and $\mathbf{Alg}(\overline{F})$ of algebras.

$$\begin{array}{ccc} \mathbf{Alg}(F) & \overset{J'}{\underset{}{\rightleftarrows}} & \mathbf{Alg}(\overline{F}) \\ \downarrow & & \downarrow \\ \mathbb{C} & \overset{J}{\underset{K}{\rightleftarrows}} & \mathcal{K}\ell(T) \end{array}$$

The left adjoint J' preserves the initial object (see e.g. [42]). □

Second, we use the initial algebra-final coalgebra coincidence in $\mathcal{K}\ell(T)$—which holds in a suitable order-enriched setting—to identify the final coalgebra in $\mathcal{K}\ell(T)$. This is our main theorem.

Theorem 3.3 (Main theorem)**.** *Assume the following:*

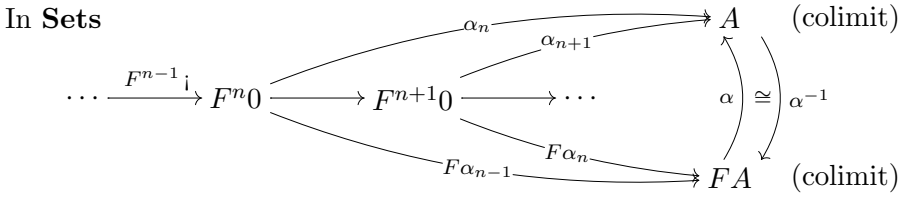
1. *A monad T on **Sets** is such that its Kleisli category $\mathcal{K}\ell(T)$ is **Cppo**-enriched and composition in $\mathcal{K}\ell(T)$ is left-strict.*
2. *For an endofunctor F on **Sets**, we have a distributive law $\lambda : FT \Rightarrow TF$. Equivalently, F has a lifting $\overline{F}$ on $\mathcal{K}\ell(T)$. Moreover, the lifting $\overline{F}$ is locally monotone.*
3. *The functor F preserves ω-colimits in **Sets**, hence has an initial algebra via the initial sequence (see Proposition A.1).*

Then the initial F-algebra $\alpha : FA \xrightarrow{\cong} A$ yields a final $\overline{F}$-coalgebra in $\mathcal{K}\ell(T)$ by

$$(J\alpha)^{-1} = J(\alpha^{-1}) = \eta_{FA} \circ \alpha^{-1} \ : \quad A \longrightarrow \overline{F}A \qquad \text{in } \mathcal{K}\ell(T) \ .$$

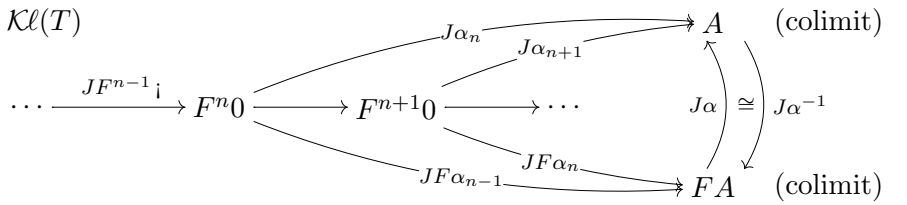
We first present the main line of the proof. Some details are provided in the form of subsequent lemmas. Note that the assumptions are satisfied by $T \in \{\mathcal{L}, \mathcal{P}, \mathcal{D}\}$ and shapely F; see Lemmas 2.5 and 2.4.

Proof. By the assumption (3) we obtain the initial algebra via the initial sequence in **Sets**.



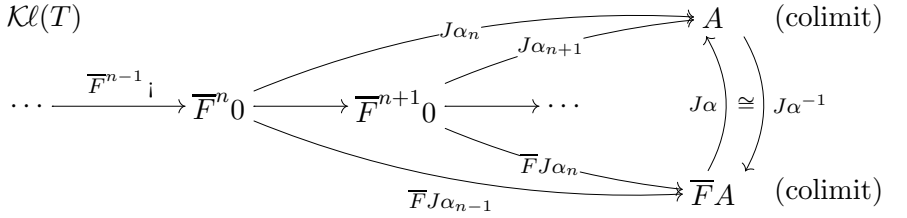
(3.1)

Here $0 = \emptyset \in \mathbf{Sets}$ is initial and $¡ : 0 \to X$ is the unique arrow from 0 to an arbitrary X. We apply the functor $J : \mathbf{Sets} \to \mathcal{K}\ell(T)$ to the whole diagram. Since J is a left adjoint it preserves colimits: hence the two cocones in the following diagram are both colimits again.



(3.2)

The ω-chain in this diagram is in fact the initial sequence for the functor $\overline{F}$ (Lemma 3.4) because, for example, a left adjoint J preserves initial objects. Moreover the lower cone is the image of the upper cone under $\overline{F}$; see the diagram (2.3). Hence the diagram (3.2) is equal to the following one. Recall that $\overline{F}X = FX$ on objects.



(3.3)

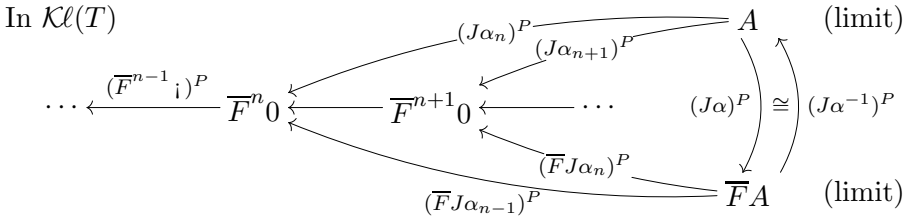
Thus Proposition A.1 yields that $J\alpha : \overline{F}A \xrightarrow{\cong} A$ is an initial $\overline{F}$-algebra. This can be seen as a more concrete proof of Proposition 3.2.

Now we show the initial algebra-final coalgebra coincidence in $\mathcal{K}\ell(T)$. This is done by reversing all the arrows in (3.3) and transforming the diagram into the one of the final sequence and its limits.

We notice (Lemma 3.6) that each arrow $\overline{F}^n ¡$ in the initial sequence is an embedding (Definition A.4). Hence the limit-colimit coincidence Theorem A.8 says that every arrow in the diagram is an embedding. Note that $J\alpha$ and $J\alpha^{-1}$, inverse to each other, form an embedding-projection pair.

By taking the corresponding projections—they are uniquely determined (Lemma A.5) and are denoted by $(\_)^P$—we obtain the next diagram. The limit-colimit coincidence Theorem A.8 says that the two resulting cones are both limits. It is also obvious that the

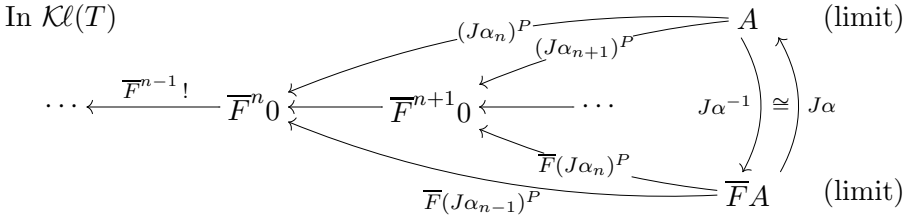
whole diagram commutes.



(3.4)

The ω^{op}-chain here is indeed a final sequence: Lemma 3.5 shows—using the assumption (1) on left-strictness—that 0 is also final in $\mathcal{K}\ell(T)$, and according to Lemma 3.6 we have $(\overline{F}^n\,¡\,)^P = \overline{F}^n\,!$ where $! : X \to 0$ is the unique arrow to the final object 0 in $\mathcal{K}\ell(T)$. As to the lower cone we have $\big(\overline{F}J\alpha_n\big)^P = \overline{F}\big((J\alpha_n)^P\big)$ by Lemma 3.7.

Hence the diagram (3.4) is equal to the following one, showing the final sequence for $\overline{F}$, its limit (the upper one) and that limit mapped by $\overline{F}$ (the lower one) which is again a limit.



(3.5)

By Proposition A.2 we conclude that $J\alpha^{-1}$ is a final $\overline{F}$-coalgebra. □

In the remainder of this section the lemmas used in the above proof are presented. We rely on the same assumptions as in Theorem 3.3.

Lemma 3.4. *The ω-chain in the diagram (3.2) is indeed the initial sequence for $\overline{F}$. That is, we have for each $n < \omega$,*

$$JF^n\big(\,¡^{\mathbf{Sets}}\big) = \overline{F}^n\big(\,¡^{\mathcal{K}\ell(T)}\big) \;:\; JF^n 0 \longrightarrow JF^{n+1}0 \quad \textit{in } \mathcal{K}\ell(T),$$

where $¡^{\mathbf{Sets}} : 0 \to F0$ in **Sets** *and $¡^{\mathcal{K}\ell(T)} : 0 \to F0$ in $\mathcal{K}\ell(T)$ denote the unique maps.*

Proof. By induction on n. For $n = 0$ the two maps are equal due to the initiality of $J0 = 0$ in $\mathcal{K}\ell(T)$. For the step case we use the commutativity $JF = \overline{F}J$ of (2.3). □

Lemma 3.5. *The empty set 0 is both an initial and a final object in $\mathcal{K}\ell(T)$.*

In particular, this implies that the object $T0$ is final in **Sets**.

Proof. The functor $J :$ **Sets** $\to \mathcal{K}\ell(T)$ preserves initial objects since it is a left adjoint. Therefore $0 = J0$ is initial in $\mathcal{K}\ell(T)$. Finality follows essentially from the left-strictness assumption: for each set X there exists at least one arrow $X \to 0$ in $\mathcal{K}\ell(T)$, for example $\bot_{X,0}$. To show the uniqueness of such an arrow, take an arbitrary arrow $f : X \to 0$ in $\mathcal{K}\ell(T)$. Recalling that the bottom map $\bot_{0,0} : 0 \to 0$ is also the identity arrow in $\mathcal{K}\ell(T)$ because of initiality, we obtain

$$f \;=\; \mathrm{id} \circ f \;=\; \bot_{0,0} \circ f \;\overset{(*)}{=}\; \bot_{X,0} \;,$$

where the compositions are taken in $\mathcal{K}\ell(T)$ and the equality marked by $(*)$ holds by left-strictness of composition. □

Lemma 3.6. *Each arrow $\overline{F}^n\, ¡$ in the initial sequence for $\overline{F}$, as in the diagram (3.3), is an embedding. Its corresponding projection is given by*

$$\big(\overline{F}^n\, ¡\big)^P = \overline{F}^n\, ! \qquad \text{in} \quad F^n 0 \underset{\overline{F}^n\, ¡}{\overset{\overline{F}^n\, !}{\leftleftarrows}} F^{n+1}0 \quad .$$

Proof. We show that $(\overline{F}^n\, ¡, \overline{F}^n\, !)$ is an embedding-projection pair for all $n < \omega$. We have $\overline{F}^n\, ! \circ \overline{F}^n\, ¡ = \mathrm{id}$ because $! \circ ¡ = \mathrm{id}$. For the other half we have

$$\begin{aligned} \overline{F}^n\, ¡ \circ \overline{F}^n\, ! &= \overline{F}^n(¡ \circ !) \\ &= \overline{F}^n(\bot_{0,F0} \circ !) && \text{initiality of 0 in } \mathcal{K}\ell(T) \\ &= \overline{F}^n(\bot_{F0,F0}) && \text{composition is left-strict} \\ &\sqsubseteq \overline{F}^n(\mathrm{id}) = \mathrm{id} && \overline{F} \text{ is locally monotone.} \end{aligned}$$

□

Lemma 3.7. *We have $\big(\overline{F}J\alpha_n\big)^P = \overline{F}\big((J\alpha_n)^P\big)$. Hence the lower cone in the diagram (3.4) is the image of the upper cone under $\overline{F}$.*

Proof. It is easy to check that $\big(\overline{F}J\alpha_n,\, \overline{F}\big((J\alpha_n)^P\big)\big)$ indeed form an embedding-projection pair. Therein we use the monotonicity of $\overline{F}$'s action on arrows. □

3.2. **Simpler proof in $\mathcal{K}\ell(\mathcal{P}) \cong$ Rel.** When $T = \mathcal{P}$ we have the self-duality

$$Op \quad : \quad \mathcal{K}\ell(\mathcal{P})^{\mathrm{op}} \xrightarrow{\cong} \mathcal{K}\ell(\mathcal{P}) \quad .$$

This is because of the following bijective correspondence between functions

$$\frac{X \xrightarrow{\ f\ } \mathcal{P}Y \ \text{ in } \mathbf{Sets}}{Y \xrightarrow{\ f^\vee\ } \mathcal{P}X \ \text{ in } \mathbf{Sets}}$$

given by $f^\vee(y) = \{x \in X \mid y \in f(x)\}$. Recalling $\mathcal{K}\ell(\mathcal{P}) \cong \mathbf{Rel}$, this mapping $f \mapsto f^\vee$ corresponds to taking the opposite relation.

Due to this "global" duality $\mathcal{K}\ell(\mathcal{P}) \cong \mathcal{K}\ell(\mathcal{P})^{\mathrm{op}}$, the proof of Theorem 3.3 is drastically simplified for $T = \mathcal{P}$. It essentially relies on the lifted self duality $\mathbf{Alg}(\overline{F}) \cong \mathbf{Alg}(\overline{F}^{\mathrm{op}})$, where the latter is isomorphic to $(\mathbf{Coalg}(\overline{F}))^{\mathrm{op}}$. We do not need here an order structure of $\mathcal{K}\ell(\mathcal{P})$ nor local monotonicity of $\overline{F}$.

Theorem 3.8. *Let $F : \mathbf{Sets} \to \mathbf{Sets}$ be a functor which preserves weak pullbacks, and $\overline{F} : \mathcal{K}\ell(\mathcal{P}) \to \mathcal{K}\ell(\mathcal{P})$ be its lifting induced by relation lifting (Lemma 2.3). Then the initial F-algebra in* **Sets** *yields the final $\overline{F}$-coalgebra in $\mathcal{K}\ell(\mathcal{P})$.*

Proof. We have the following situation because of the self-duality of $\mathcal{K}\ell(\mathcal{P})$.

$$\mathbf{Sets} \underset{K}{\overset{J}{\underset{\longleftarrow}{\overset{\longrightarrow}{\bot}}}} \mathcal{K}\ell(\mathcal{P}) \xrightarrow[\cong]{Op^{\mathrm{op}}} \mathcal{K}\ell(\mathcal{P})^{\mathrm{op}}$$

with F acting on $\mathbf{Sets}$, $\overline{F}$ on $\mathcal{K}\ell(\mathcal{P})$, and $\overline{F}^{\mathrm{op}}$ on $\mathcal{K}\ell(\mathcal{P})^{\mathrm{op}}$.

The adjunction $J \dashv K$ and the isomorphism $Op : \mathcal{K}\ell(\mathcal{P})^{\mathrm{op}} \xrightarrow{\cong} \mathcal{K}\ell(\mathcal{P})$ lift to those between the categories of algebras.

$$\begin{array}{ccccccc}
\mathbf{Alg}(F) & \underset{\bot}{\overset{J'}{\rightleftarrows}} & \mathbf{Alg}(\overline{F}) & \xrightarrow[\cong]{(Op')^{\mathrm{op}}} & \mathbf{Alg}(\overline{F}^{\mathrm{op}}) & \xrightarrow{\cong} & (\mathbf{Coalg}(\overline{F}))^{\mathrm{op}} \\
\downarrow & & \downarrow & & \downarrow & & \\
\mathbf{Sets} & \underset{K}{\overset{J}{\rightleftarrows}} & \mathcal{K}\ell(\mathcal{P}) & \xrightarrow[\cong]{Op^{\mathrm{op}}} & \mathcal{K}\ell(\mathcal{P})^{\mathrm{op}} & &
\end{array}$$

Indeed, $J \dashv K$ lifts due to Proposition 3.2; the lifted isomorphism $Op' : \mathbf{Alg}(\overline{F}) \xrightarrow{\cong} \mathbf{Alg}(\overline{F}^{\mathrm{op}})$ is because of the following commutativity:

$$\begin{array}{ccc}
\mathcal{K}\ell(\mathcal{P})^{\mathrm{op}} & \xrightarrow{Op} & \mathcal{K}\ell(\mathcal{P}) \\
{\scriptstyle \overline{F}^{\mathrm{op}}}\downarrow & & \downarrow{\scriptstyle \overline{F}} \\
\mathcal{K}\ell(\mathcal{P})^{\mathrm{op}} & \xrightarrow[Op]{} & \mathcal{K}\ell(\mathcal{P})
\end{array} \tag{3.6}$$

which is because: $\overline{F}R = \mathrm{Rel}_F(R)$ (see (2.4)); and taking relation liftings is compatible with opposite relations (i.e. $\mathrm{Rel}_F(R^{\mathrm{op}}) = (\mathrm{Rel}_F R)^{\mathrm{op}}$, see [22]). Moreover the category $\mathbf{Alg}(\overline{F}^{\mathrm{op}})$ is obviously isomorphic to $(\mathbf{Coalg}(\overline{F}))^{\mathrm{op}}$.

Therefore the initial object in $\mathbf{Alg}(F)$ is carried to that in $(\mathbf{Coalg}(\overline{F}))^{\mathrm{op}}$, hence the final object in $\mathbf{Coalg}(\overline{F})$. □

For monads such as $T = \mathcal{D}$ a "global" self-duality $\mathcal{K}\ell(T) \cong \mathcal{K}\ell(T)^{\mathrm{op}}$ is not available. Instead, in the proof of Theorem 3.3, we exploit the "partial" duality which holds between the colimit/limit of the initial/final sequence.

3.3. **Related work: axiomatic domain theory.** The initial algebra-final coalgebra coincidence is heavily exploited in the field of *axiomatic domain theory*, e.g. in [9, 12, 13, 50]. There, categories which have coinciding initial algebra and final coalgebra for each endofunctor are called *algebraically compact categories*. They draw special attention as suitable "categories of domains" for denotational semantics of datatype construction. The relevance comes as follows.

Let $\mathbb{C}$ be a "category of domains." We think of an object of the category $\mathbb{C}$ as a type. A "recursive" datatype constructor—a prototypical example is $(X, Y) \mapsto Y^X$—is presented as a bifunctor $G : \mathbb{C}^{\mathrm{op}} \times \mathbb{C} \to \mathbb{C}$. Note the presence of both covariance and contravariance. We expect that such a category $\mathbb{C}$ has a canonical fixed point $\mathrm{Fix}\, G$ such that

$$G(\mathrm{Fix}\, G, \mathrm{Fix}\, G) \xrightarrow{\cong} \mathrm{Fix}\, G \ ,$$

which models the recursive type determined by the datatype constructor G. Freyd [12] showed that if $\mathbb{C}$ is algebraically compact, then we can construct such a fixed point as a suitable initial algebra; moreover this fixed point is shown by Fiore [9] to be a canonical one in a suitable sense. The rough idea here is that the covariant part of G is taken care of by an initial algebra; the contravariant part is by a final coalgebra; the initial algebra-final coalgebra coincidence yields a fixed point of overall G.

Typical examples of algebraically compact categories are enriched over **Cppo** or one of its variants. This conforms the traditional use of the word "domain" for certain cpo's (e.g. in [1]).

Although we utilize the initial algebra-final coalgebra coincidence result in $\mathcal{K}\ell(T)$, we are not so much interested in algebraic compactness of $\mathcal{K}\ell(T)$. This is because our motivation is different from that of axiomatic domain theory. In studying trace semantics for coalgebras, we need not deal with *every* endofunctor on $\mathcal{K}\ell(T)$, but only such an endofunctor $\overline{F}$ which is a lifting of $F : \mathbf{Sets} \to \mathbf{Sets}$.

In a different context of functional programming, the work [44] also studies initial algebras and final coalgebras in a Kleisli category. The motivation there is to combine *data types* and *effects*. More specifically, an initial algebra and a final coalgebra support the *fold* and the *unfold* operators, respectively, used in recursive programs over datatypes. A computational effect is presented as a monad, and its Kleisli category is the category of effectful computations.

The difference between [44] and the current work is as follows. In [44], the original category of pure functions is already algebraically compact; the paper studies the conditions for the algebraic compactness to be carried over to Kleisli categories. In contrast, in the current work, it is a monad—with a suitable order structure, embodying the essence of "branching"—which yields the initial algebra-final coalgebra coincidence on a Kleisli category; the coincidence is not present in the original category **Sets**.

3.3.1. *Local continuity vs. local monotonicity.* In axiomatic domain theory, **Cppo**-enriched categories are said to be algebraically compact because, "in a 2-category setting" [13], every endofunctor has an initial algebra and a final coalgebra. Concretely this means: "every locally continuous functor."

In this spirit, we could have made a stronger assumption of $\overline{F}$'s local continuity in Theorem 3.3 instead of local monotonicity. If we do so, in fact, the proof of Theorem 3.3 becomes much simpler: the following proposition (Lemma in [13, p.98]) immediately yields the initial algebra-final coalgebra coincidence for a locally continuous $\overline{F}$.

Proposition 3.9 ([13])**.** *Let $\mathbb{D}$ be a* **Cppo***-enriched category whose composition is left-strict, and $G : \mathbb{D} \to \mathbb{D}$ be a locally continuous endofunctor. An initial algebra $\beta : GB \xrightarrow{\cong} B$, if it exists, yields a final coalgebra $\beta^{-1} : B \xrightarrow{\cong} GB$.*

Proof. Given a coalgebra $d : Y \to GY$, the function

$$\Phi \ : \ \mathbb{D}(Y, B) \longrightarrow \mathbb{D}(Y, B) \ , \qquad f \longmapsto \beta \circ Gf \circ d$$

is continuous due to the local continuity of G. Hence it has the least fixed point $\bigsqcup_{n<\omega} \Phi^n(\bot)$; this proves existence of a morphism from d to β^{-1}.

$$\begin{array}{ccc} GY & \longrightarrow & GB \\ {\scriptstyle d}\uparrow & & \cong\uparrow{\scriptstyle \beta^{-1}} \\ Y & \longrightarrow & B \end{array}$$

Now we shall show its uniqueness. Assume that $g : Y \to B$ is a morphism of coalgebras as above, that is, $\Phi(g) = g$. Similarly to Φ, we define a function $\Psi : \mathbb{D}(B, B) \to \mathbb{D}(B, B)$ as

the one which carries $h : B \to B$ to $\beta \circ Gh \circ \beta^{-1}$. We have

$$\begin{aligned}
\bigsqcup_n \Phi^n(\bot) &= \bigsqcup_n \Phi^n(B \xrightarrow{g} B \xrightarrow{\bot} Y) && \text{composition is left-strict, so } \bot \circ g = \bot \\
&= \bigsqcup_n \big(\Psi^n(\bot) \circ \Phi^n(g)\big) && \Phi^n(\bot \circ g) = \Psi^n(\bot) \circ \Phi^n(g), \text{ by induction} \\
&= \big(\bigsqcup_n \Psi^n(\bot)\big) \circ \big(\bigsqcup_n \Phi^n(g)\big) && \text{composition is continuous} \\
&= \bigsqcup_n \Phi^n(g) && \bigsqcup_n \Psi^n(\bot) = \mathrm{id},\ (*) \\
&= g && \Phi(g) = g \text{ by assumption.}
\end{aligned}$$

Here $(*)$ holds because $\bigsqcup_n \Psi^n(\bot)$, being a fixed point for Ψ, is the unique morphism of algebras from β to β. This shows that the morphism g must be the least fixed point of Φ. □

For our main Theorem 3.3 we can do with only local monotonicity of the lifted functor $\overline{F}$, by taking a closer look at the initial/final sequences. However at this stage it is not clear how much we gain from this generality: up to now we have not found an example where the functor $\overline{F}$ is only locally monotone (and not locally continuous).

4. Finite trace semantics via coinduction

In this section we shall further illustrate the observation that the principle of coinduction, when employed in $\mathcal{K}\ell(T)$, captures trace semantics of state-based systems. As we have shown in the previous section, an initial algebra in **Sets** constitutes the semantic domain, i.e. is a final coalgebra in $\mathcal{K}\ell(T)$. Viewing an initial algebra as the set of well-founded terms (such as finite words or finite-depth parse trees), this fact means that the "trace semantics" induced by coinduction is inevitably *finite*, in the sense that it captures only finite behavior. Here we will elaborate on this finiteness issue as well.

4.1. **Trace semantics by coinduction.** As we have seen in Section 2.2 various types of state-based systems allow their presentation as coalgebras $X \to \overline{F}X$ in a Kleisli category $\mathcal{K}\ell(T)$. For example,

- LTSs with explicit termination, with $T = \mathcal{P}$ and $F = 1 + \Sigma \times \_$;
- probabilistic LTSs (also called *generative probabilistic transition systems* in [52, 58]) with explicit termination, with $T = \mathcal{D}$ and $F = 1 + \Sigma \times \_$;
- context-free grammars with $T = \mathcal{P}$ and $F = (\Sigma + \_)^*$.

The main observation underlying this work is the following. If we instantiate the parameters

$$T \text{ for branching type} \qquad \text{and} \qquad F \text{ for transition type}$$

in the coinduction diagram

$$\begin{array}{ccc}
\overline{F}X & \xrightarrow{\overline{F}(\mathsf{tr}_c)} & \overline{F}A \\
{\scriptstyle c}\uparrow & & \cong\uparrow{\scriptstyle J\alpha^{-1}} \\
X & \xrightarrow[\mathsf{tr}_c]{} & A
\end{array} \qquad \text{in } \mathcal{K}\ell(T) \tag{4.1}$$

with one of the above choices, then the commutativity of the diagram is equivalent to the corresponding (conventional) definition of trace semantics in Section 1.1. Therefore we claim that the diagram (4.1) is the mathematical principle underlying various "trace semantics," no matter if it is "trace set" (non-deterministic) or "trace distribution" (probabilistic).

Corollary 4.1 (Trace semantics for coalgebras)**.** *Assume that T and F are such as in Theorem 3.3, and $\alpha : FA \xrightarrow{\cong} A$ is an initial F-algebra in* **Sets***. Given a coalgebra $c : X \to TFX$ in* **Sets***, we can assign a function*

$$\mathsf{tr}_c \;:\; X \longrightarrow TA \qquad \text{in } \mathbf{Sets}$$

*which is, as an arrow $X \to A$ in $\mathcal{K}\ell(T)$, the unique one making the diagram (4.1) commute. We shall call this function tr_c the **(finite) trace semantics** for the coalgebra c.* □

Example 4.2. As further illustration we give details for the choice of parameters $T = \mathcal{P}$ and $F = 1 + \Sigma \times \_$. This is the suitable choice to deal with the first system in (1.1).

Now the coinduction diagram looks as follows. Recall that an initial F-algebra is carried by the set Σ^* of finite words.

$$\begin{array}{ccc} 1 + \Sigma \times X & \xrightarrow{1+\Sigma\times \mathsf{tr}_c} & 1 + \Sigma \times \Sigma^* \\ {\scriptstyle c}\uparrow & & \cong\uparrow{\scriptstyle J([\mathsf{nil},\mathsf{cons}])^{-1}} \\ X & \xrightarrow[\mathsf{tr}_c]{} & \Sigma^* \end{array} \qquad \text{in } \mathcal{K}\ell(\mathcal{P}) \tag{4.2}$$

It assigns, to a system c, a function $\mathsf{tr}_c : X \to \mathcal{P}(\Sigma^*)$ which carries a state $x \in X$ to the set of finite words on Σ which can possibly arise as an execution "trace" of c starting from x. The commutativity states equality of two arrows $X \rightrightarrows 1 + \Sigma \times \Sigma^*$ in $\mathcal{K}\ell(\mathcal{P})$, that is, functions $X \rightrightarrows \mathcal{P}(1 + \Sigma \times \Sigma^*)$. Let us denote these functions by

$$u = (1 + \Sigma \times \mathsf{tr}_c) \circ c \quad \text{(up, then right)}, \qquad v = J([\mathsf{nil}, \mathsf{cons}])^{-1} \circ \mathsf{tr}_c \quad \text{(right, then up)}.$$

For each $x \in X$, the following conditions—derived straightforwardly by definition of composition of $\mathcal{K}\ell(\mathcal{P})$, lifting of the functor $1 + \Sigma \times \_$, etc.—specify u and v's value at x, as a subset of $1 + \Sigma \times \Sigma^*$.

$$\begin{array}{rcl} \checkmark \in u(x) & \iff & \checkmark \in c(x) \\ (a,\sigma) \in u(x) & \iff & \exists x' \in X.\, \big((a,x') \in c(x) \;\wedge\; \sigma \in \mathsf{tr}_c(x') \big) \\ \checkmark \in v(x) & \iff & \langle\rangle \in \mathsf{tr}_c(x) \\ (a,\sigma) \in v(x) & \iff & a \cdot \sigma \in \mathsf{tr}_c(x) \end{array}$$

Commutativity of (4.2) amounts to $u = v$; this gives the condition (1.3).

From a different point of view we can also express that as follows: finality of the coalgebra $\Sigma^* \xrightarrow{\cong} 1 + \Sigma \times \Sigma^*$ in (4.2) ensures that the conventional recursive definition (1.3) uniquely determines a function $\mathsf{tr}_c : X \to \mathcal{P}(\Sigma^*)$. Hence tr_c is *well-defined*.

An easy consequence of the recursive definition (1.3) is

$$a_1 \ldots a_n \in \mathsf{tr}_c(x) \quad \iff \quad \exists x_1, \ldots, x_n \in X. \quad x \xrightarrow{a_1} \cdots \xrightarrow{a_n} x_n \to \checkmark \;\; .$$

Therefore every trace $a_1 \ldots a_n \in \mathsf{tr}_c(x)$ has termination $\checkmark$ implicit at its tail. In particular, the set $\mathsf{tr}_c(x)$ is not necessarily prefix-closed: $a_1 \ldots a_n a_{n+1} \ldots a_{n+m} \in \mathsf{tr}_c(x)$ does not imply $a_1 \ldots a_n \in \mathsf{tr}_c(x)$.

Example 4.3. Let us take $T = \mathcal{L}$ (the lift monad) and $F = 1 + \Sigma \times \_$. In this case a coalgebra $X \xrightarrow{c} \mathcal{L}(1 + \Sigma \times X)$ in **Sets** is a system which can

- get into a deadlock ($c(x) = \bot$ where $\mathcal{L} = \{\bot\} + \_$),
- successfully terminate ($c(x) = \checkmark$ where $F = \{\checkmark\} + \Sigma \times \_$), or
- output a letter from Σ and move to the next state ($c(x) = (a, x')$).

By examining trace semantics for such systems, we shall formally put the difference between the computational meanings of the two elements, $\bot$ and $\checkmark$.

The coinduction diagram (4.1) instantiates to the same diagram as (4.2), but now in the category $\mathcal{K}\ell(\mathcal{L})$. Easy calculation shows that its commutativity amounts to the following condition. The function

$$X \xrightarrow{\mathsf{tr}_c} \mathcal{L}(\Sigma^*) = \{\bot\} + \Sigma^* \qquad \text{in } \mathbf{Sets}$$

satisfies, for each $x \in X$,

$$\begin{array}{lll} \mathsf{tr}_c(x) = \langle\rangle & \iff & c(x) = \checkmark \ , \\ \mathsf{tr}_c(x) = a \cdot \sigma & \iff & \exists x' \in X.\ \big(c(x) = (a, x') \wedge \mathsf{tr}_c(x') = \sigma\big) \ , \\ \mathsf{tr}_c(x) = \bot & \iff & c(x) = \bot \quad \text{or} \quad \exists x' \in X.\ \big(c(x) = (a, x') \wedge \mathsf{tr}_c(x') = \bot\big) \ . \end{array} \tag{4.3}$$

Here $\sigma \in \Sigma^*$ is a word in Σ.

For the systems under consideration, we can think of three different kinds of possible executions.

- An execution eventually hitting $\checkmark$, that is, $x \xrightarrow{a_1} \cdots \xrightarrow{a_n} x_n \to \checkmark$. By the condition (4.3) it yields a word $\mathsf{tr}_c(x) = a_1 \dots a_n$ as its trace.
- An execution eventually hitting $\bot$, that is, $x \xrightarrow{a_1} \cdots \xrightarrow{a_n} x_n \to \bot$. By the third line of (4.3) we see that $\mathsf{tr}_c(x_n) = \bot$; moreover $\mathsf{tr}_c(x_{n-1}) = \cdots = \mathsf{tr}_c(x) = \bot$. It properly reflects our intuition that a state x that eventually goes into deadlock does not yield a finite (or terminating) trace.
- An execution not hitting $\checkmark$ nor $\bot$, that is, $x \xrightarrow{a_1} x_1 \xrightarrow{a_2} \cdots$. In this case, the only possible solution of the "recursive equation" (4.3) is $\mathsf{tr}_c(x) = \mathsf{tr}_c(x_1) = \cdots = \bot$. The intuition here is: a state leading to *livelock* does not yield a finite trace.

4.2. **Infinite traces.** The trace semantics obtained via coinduction (Corollary 4.1) assigns, to each state $x \in X$, "a set of" (if $T = \mathcal{P}$) or "a distribution over" (if $T = \mathcal{D}$) elements of the initial algebra A. Elements of A are thought of as possible linear behavior of the system determined by the transition type (i.e. the functor F).

Now the intuition is that an initial F-algebra A consists of the well-founded (or finite-depth) terms and a final F-coalgebra Z consists of the possibly non-well-founded (or infinite-depth) terms. For example,

- for $F = 1 + \Sigma \times \_$, $A = \Sigma^*$ consists of all the finite words, and $Z = \Sigma^\infty = \Sigma^* + \Sigma^\omega$ is augmented with *streams*, i.e. infinite words;
- for $F = (\Sigma + \_)^*$, A is the set of finite-depth *skeletal parse trees* (see [16]), and Z additionally contains infinite-depth ones;
- for $F = \Sigma \times \_$ which models LTSs *without* explicit termination, $A = 0$ and $Z = \Sigma^\omega$.

Therefore our trace semantics $X \to TA$ only takes account of finite, well-founded linear-time behavior but not infinite ones. This is why the trace set (1.2) does not contain ab^ω; and also why we have been talking about LTSs *with* explicit termination—otherwise the finite trace semantics is always empty.

Designing a coalgebraic framework to capture possibly infinite trace semantics is the main aim of [24]. The work is done exclusively in a non-deterministic setting and the main result reads as follows.

Theorem 4.4 (Possibly infinite trace semantics for coalgebras, [24])**.** *Let F be a shapely functor on* **Sets***, and $\zeta : Z \xrightarrow{\cong} FZ$ be a final coalgebra in* **Sets***. The coalgebra*

$$J\zeta \ : \ Z \longrightarrow \overline{F}Z \qquad \text{in } \mathcal{K}\ell(\mathcal{P})$$

is weakly final: that is, given a coalgebra $c : X \to \overline{F}X$, there is a morphism from c to $J\zeta$ but the morphism is not necessarily unique.

$$\begin{array}{ccc} \overline{F}X & \overset{\overline{F}(\mathsf{tr}_c^\infty)}{\rightsquigarrow} & \overline{F}Z \\ {\scriptstyle c}\uparrow & & \cong\uparrow{\scriptstyle J\zeta} \\ X & \underset{\mathsf{tr}_c^\infty}{\rightsquigarrow} & Z \end{array} \qquad \text{in } \mathcal{K}\ell(\mathcal{P}) \tag{4.4}$$

Still there is a canonical choice tr_c^∞ among such morphisms, namely the one which is maximal with respect to the inclusion order. We shall call the function $\mathsf{tr}_c^\infty : X \to \mathcal{P}Z$ the ***possibly-infinite trace semantics*** *for c.* □

Note here that, when we take $F = 1 + \Sigma \times \_$ and $T = \mathcal{P}$ (the choice for LTSs with termination), commutativity of (4.4) boils down to exactly the same conditions as (1.3):

$$\langle\rangle \in \mathsf{tr}_c^\infty(x) \iff x \to \checkmark, \qquad a \cdot \sigma \in \mathsf{tr}_c^\infty(x) \iff \exists y.\ (\, x \xrightarrow{a} y \ \wedge \ \sigma \in \mathsf{tr}_c^\infty(y)\,). \tag{4.5}$$

Weak finality of $\Sigma^\infty \xrightarrow{\cong} 1 + \Sigma \times \Sigma^\infty$ (corresponding to $Z \xrightarrow{\cong} \overline{F}Z$ in (4.4)) means the following. The recursive definition (4.5)—although it looks valid at the first sight—does *not* uniquely determine the infinite trace map $\mathsf{tr}_c^\infty : X \to \mathcal{P}(\Sigma^\infty)$. Instead, the map tr_c^∞ is the maximal one among those which satisfy (4.5).

As an example take the first system in (1.1). We expect its possibly-infinite trace map $X \to \mathcal{P}(\Sigma^\infty)$ to be such that $x \mapsto ab^* + ab^\omega$ and $y \mapsto b^* + b^\omega$. Indeed this satisfies (4.5) and is moreover the maximal. However, the function $x \mapsto ab^*$ and $y \mapsto b^*$—this is actually the finite trace $X \to \mathcal{P}(\Sigma^*)$ embedded along $\Sigma^* \hookrightarrow \Sigma^\infty$—also satisfies (4.5). In fact, [16, Section 5] shows a general fact that such an embedding of the finite trace map is the minimal one among those morphisms which make the diagram (4.4) commute.

The coalgebraic characterization (Theorem 4.4) of possibly-infinite trace semantics is not yet fully developed. In particular the current proof of Theorem 4.4 (in [24]) is fairly concrete and a categorical principle behind it is less clear than the one behind finite traces. Consequently the result's applicability is limited: we do not know whether the result holds in a probabilistic setting; or whether it holds for any weak-pullback-preserving functor F.

5. Trace semantics as testing equivalence

In this section we will observe that, in a non-deterministic setting, the coalgebraic finite trace semantics (i.e. coinduction in $\mathcal{K}\ell(\mathcal{P})$) gives rise to a canonical *testing situation* in which a test is an element of the initial F-algebra A in **Sets**. Here F specifies the transition type, just as before. The notion of testing situations (Definition 5.1) and its variants have attracted many authors' attention in the context of coalgebraic modal logic; our aim here is to demonstrate genericity and pervasiveness of the notion of testing situations by presenting an example which is not much like modal logic (that is, propositional logic plus modality).

In Section 5.1 we introduce the notion of testing situations and investigate some of their general properties. Our main concern there is the comparison between two process equivalences, namely *testing equivalence* and *equivalence modulo final coalgebra semantics*.

We present the equivalences categorically as suitable kernel pairs; this makes the arguments simple and clean. In Section 5.2 we present the canonical testing situation for trace semantics. Moreover we show that it is *expressive*: the testing captures final coalgebra semantics, which is now trace semantics.

5.1. **Testing situations.** Recent studies [5, 6, 32, 35, 37, 45] on coalgebra and modal logic have identified (variants of) the following categorical situation as the essential underlying structure. Following [45], we prefer using a more general term "testing": it subsumes "modal logic" in the following sense. We learn properties of a system through pass or failure of *tests*; modal logic constitutes a special case where tests are modal formulas.

Definition 5.1. A *testing situation* is the following situation of a contravariant adjunction $S^{\mathrm{op}} \dashv P$ and two endofunctors F, M

$$F^{\mathrm{op}} \circlearrowleft \mathbb{C}^{\mathrm{op}} \underset{S^{\mathrm{op}}}{\overset{P}{\rightleftarrows}} \mathbb{A} \circlearrowright M \qquad (5.1)$$

plus a "denotation" natural transformation $\delta : MP \Rightarrow PF^{\mathrm{op}} : \mathbb{C}^{\mathrm{op}} \to \mathbb{A}$, which consists of arrows $MPX \xrightarrow{\delta_X} PFX$ in $\mathbb{A}$.

Note that the denotation δ is a parameter: the same "syntax for tests" $M : \mathbb{A} \to \mathbb{A}$ can have different interpretations with different δ.

The requirements in Definition 5.1 are the same as in [32, 45]. They are what we need to compare two process semantics, namely *testing equivalence*—which arises naturally from the concept of testing—and final coalgebra semantics.[8] We shall explain each ingredient's role, using the well-established terminology of modal logic.

- The endofunctor $F : \mathbb{C} \to \mathbb{C}$ makes $\mathbf{Coalg}(F)$ the category of "systems," or "Kripke models" in modal logic.
- The category $\mathbb{A}$—typical examples being **Bool** of Boolean algebras or **Heyt** of Heyting algebras—is that of "propositional logic." The functor M specifies "modality": modal operators and axioms. Then $\mathbf{Alg}(M)$ is the category of "modal algebras"; the initial M-algebra $ML \xrightarrow{\cong} L$ is a "modal logic" consisting of modal formulas, modulo logical equivalence.
- The denotation δ specifies how the modality M is interpreted via transitions of type F. This allows to give "Kripke semantics" for the modal logic: given a coalgebra (or a "Kripke model") $c : X \to FX$, interpretation $[\![\_]\!]_c$ of modal formulas therein is given by the following induction.

$$\begin{array}{ccc} ML & \dashrightarrow & MPX \\ {\scriptstyle \text{initial}}\downarrow{\scriptstyle \cong} & & \downarrow{\scriptstyle \delta_X} \\ L & & PFX \\ & & \downarrow{\scriptstyle Pc} \\ L & \underset{[\![\_]\!]_c}{\dashrightarrow} & PX \end{array} \qquad (5.2)$$

[8] In fact we can be even more liberal: existence of a denotation δ can be replaced by existence of a lifting $\hat{P} : \mathbf{Coalg}(F)^{\mathrm{op}} \to \mathbf{Alg}(M)$ of P. The results in this section nevertheless hold in that case. The latter condition (there is a lifting $\hat{P}$) is strictly weaker than the former (there is a natural transformation δ): obviously δ induces $\hat{P}$ but not the other way round. Let $\mathbb{C} = \omega^{\mathrm{op}}, \mathbb{A} = \omega, P = \mathrm{id}, F = (1 + \_)^{\mathrm{op}}$ and $M = 2 + \_$. Then both $\mathbf{Coalg}(F)$ and $\mathbf{Alg}(M)$ are the empty category hence P has the trivial lifting. However there is no natural transformation $MPX \to PF^{\mathrm{op}}X$.

- Why a right adjoint S of P^{op}? It allows us, via transposition, to assign a modal "theory" to each state of a Kripke model.

$$\frac{L \xrightarrow{[\![\_]\!]_c} PX \quad \text{in } \mathbb{A}}{X \xrightarrow{\mathsf{th}_c} SL \quad \text{in } \mathbb{C}} \; (S^{\mathrm{op}} \dashv P) \tag{5.3}$$

  The theory $\mathsf{th}_c(x)$ associated with a state x contains precisely the modal formulas that hold at x.

Following the above intuition, we define the categorical notion of testing equivalence—two states are testing-equivalent if they have the same modal theory.

Definition 5.2. Assume that we have a testing situation (5.1), and that $\mathbb{C}$ has finite limits. On a coalgebra $X \xrightarrow{c} FX$, the *testing equivalence* TestEq_c is the kernel pair of the theory map th_c defined by (5.2) and (5.3). Equivalently,

$$\mathsf{TestEq}_c \overset{\langle p_1, p_2 \rangle}{\rightarrowtail} X \times X \underset{\mathsf{th}_c \circ \pi_2}{\overset{\mathsf{th}_c \circ \pi_1}{\rightrightarrows}} SL \tag{5.4}$$

is an equalizer.

Similarly, we introduce the categorical notion of "equivalence modulo final coalgebra semantics"; we shall call it *FCS-equivalence* for short.

Definition 5.3. Assume that there is a final F-coalgebra $\zeta : Z \xrightarrow{\cong} FZ$, and that $\mathbb{C}$ has finite limits. On a coalgebra $X \xrightarrow{c} FX$, the *FCS-equivalence* FCSEq_c is the kernel pair of the unique map $\mathsf{beh}_c : X \to Z$ induced by finality. Equivalently,

$$\mathsf{FCSEq}_c \overset{\langle q_1, q_2 \rangle}{\rightarrowtail} X \times X \underset{\mathsf{beh}_c \circ \pi_2}{\overset{\mathsf{beh}_c \circ \pi_1}{\rightrightarrows}} Z \tag{5.5}$$

is an equalizer.

It is easily seen that the two "relations" TestEq_c and FCSEq_c on X are *equivalence relations* in the sense of [23, Section 1.3]. That is, they satisfy the reflexivity, symmetry, and transitivity conditions when the conditions are suitably formulated in categorical terms.

Now our concern is the comparison between two process semantics TestEq_c and FCSEq_c, as subobjects of $X \times X$. The following lemma is crucial for our investigation; in fact it is important for coalgebraic modal logic in general and appears e.g. as [32, Theorem 3.3].

Lemma 5.4. *A morphism of F-coalgebras preserves theory maps. That is,*

$$\begin{array}{ccc} FX & \xrightarrow{Ff} & FY \\ {\scriptstyle c}\uparrow & & \uparrow{\scriptstyle d} \\ X & \xrightarrow[f]{} & Y \end{array} \quad \textit{implies} \quad \begin{array}{ccc} X & & \\ {\scriptstyle f}\downarrow & \searrow^{\mathsf{th}_c} & \\ Y & \xrightarrow[\mathsf{th}_d]{} & SL \end{array} \quad .$$

Proof. The following induction diagram proves $Pf \circ [\![\_]\!]_d = [\![\_]\!]_c$. Naturality of δ plays an important role there.

$$\begin{array}{ccccc} ML & \overset{Ff}{\dashrightarrow} & MPY & \xrightarrow{MPf} & MPX \\ {\scriptstyle \text{initial}}\downarrow{\scriptstyle \cong} & & \downarrow{\scriptstyle \delta_Y} & & \downarrow{\scriptstyle \delta_X} \\ & & PFY & \xrightarrow[PFf]{} & PFX \\ & & \downarrow{\scriptstyle Pd} & & \downarrow{\scriptstyle Pc} \\ L & \underset{[\![\_]\!]_d}{\dashrightarrow} & PY & \xrightarrow[Pf]{} & PX \end{array} .$$

Then the claim follows from naturality of the transposition (5.3). □

We show that in a testing situation like (5.1), tests respect final coalgebra semantics. That is, testing does not distinguish two FCS-equivalent states.

Proposition 5.5. *Consider such a testing situation and equivalence relations as in Definitions 5.2 and 5.3. For any coalgebra $X \xrightarrow{c} FX$ we have an inclusion*

$$\mathsf{FCSEq}_c \leq \mathsf{TestEq}_c$$

of subobjects of $X \times X$.

Proof. It suffices to show that the arrow $\langle q_1, q_2 \rangle$ in (5.5) equates the parallel arrows in (5.4); then the claim follows from universality of an equalizer.

$$\begin{aligned} \mathsf{th}_c \circ \pi_1 \circ \langle q_1, q_2 \rangle &= \mathsf{th}_c \circ q_1 \\ &= \mathsf{th}_\zeta \circ \mathsf{beh}_c \circ q_1 && (*) \\ &= \mathsf{th}_\zeta \circ \mathsf{beh}_c \circ q_2 && \text{due to (5.5)} \\ &= \mathsf{th}_c \circ q_2 && (*) \\ &= \mathsf{th}_c \circ \pi_2 \circ \langle q_1, q_2 \rangle \ . \end{aligned}$$

Here $(*)$ is an instance of Lemma 5.4: beh_c is a morphism of coalgebras from c to the final ζ. □

The converse $\mathsf{TestEq}_c \leq \mathsf{FCSEq}_c$ does not hold in general. For a fixed type of systems (i.e. for fixed $F : \mathbb{C} \to \mathbb{C}$), we can think of logics with varying degree of expressive power; this results in process equivalences with varying granularity. This view is systematically presented by van Glabbeek in [57] as the *linear time-branching time spectrum*—a categorical version of which we consider as an important direction of future work.

It is when we have $\mathsf{FCSEq}_c \overset{\cong}{\Rightarrow} \mathsf{TestEq}_c$ that a modal logic (considered as a testing situation) is said to be *expressive*. Recall that FCSEq_c usually coincides with *bisimilarity* if $\mathbb{C}$ is **Sets**: in this case an expressive logic *captures* bisimilarity.

The following proposition states a (rather trivial) equivalent condition for a testing situation to be expressive. For more ingenious sufficient conditions—which essentially rely on the transpose of δ being monic—see e.g. [32].

Proposition 5.6. *Consider a testing situation as in Definitions 5.2 and 5.3. The testing is expressive, that is, for any coalgebra $X \xrightarrow{c} FX$ we have*

$$\mathsf{TestEq}_c \overset{\cong}{\Rightarrow} \mathsf{FCSEq}_c$$

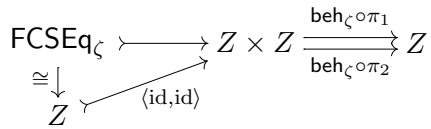
as subobjects of $X \times X$, if and only if the theory map $\mathsf{th}_\zeta : Z \to SL$ for the final coalgebra is a mono.

Proof. We first prove the "if" direction. In view of Proposition 5.5, it suffices to show that $\langle p_1, p_2\rangle$ in (5.4) equalizes $\mathsf{beh}_c \circ \pi_1$ and $\mathsf{beh}_c \circ \pi_2$ (which proves $\mathsf{TestEq}_c \leq \mathsf{FCSEq}_c$).

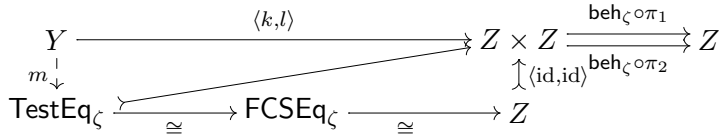
$$\begin{aligned} \mathsf{th}_\zeta \circ \mathsf{beh}_c \circ p_1 &= \mathsf{th}_c \circ p_1 && \text{by Lemma 5.4} \\ &= \mathsf{th}_c \circ p_2 && \text{due to (5.4)} \\ &= \mathsf{th}_\zeta \circ \mathsf{beh}_c \circ p_2 && \text{by Lemma 5.4} \end{aligned}$$

We have $\mathsf{beh}_c \circ p_1 = \mathsf{beh}_c \circ p_2$ since th_ζ is a mono.

To prove the "only if" direction, first we observe that the FCS-equivalence on the final coalgebra $\zeta : Z \xrightarrow{\cong} FZ$ is the diagonal relation: that is,



This is because $\mathsf{beh}_\zeta = \mathrm{id} : Z \to Z$. Now assume that $\mathsf{th}_\zeta \circ k = \mathsf{th}_\zeta \circ l$ for $k, l : Y \rightrightarrows Z$. Universality of an equalizer TestEq_ζ induces a mediating arrow m in the following diagram.



The whole diagram commutes since $\mathsf{TestEq}_\zeta \cong \mathsf{FCSEq}_\zeta$ (by assumption) and $\mathsf{FCSEq}_\zeta \cong Z$ (by the above observation), both as subobjects of $Z \times Z$. This proves $k = l$. □

Remark 5.7. The literature [5, 6] considers more restricted settings than the testing situations in Definition 5.1. There an adjunction $S^{\mathrm{op}} \dashv P$ is replaced by a dual equivalence of categories, and a denotation δ is required to be a natural isomorphism. These additional restrictions allow one to say more about the situations: logics are always expressive; the main concern of [6] is how to present an abstract modal logic $M : \mathbb{A} \to \mathbb{A}$ by concrete syntax. However, for our purpose in Section 5.2 the greater generality of our notion of testing situations is needed.

5.2. **Canonical testing for trace semantics in $\mathcal{K}\ell(\mathcal{P})$.** In this section we shall present a canonical testing situation for coalgebras in $\mathcal{K}\ell(\mathcal{P})$. We shall also show that the testing is "expressive," in the sense that the testing captures final coalgebra semantics. The intuition is as follows.

Trace semantics for non-deterministic systems assigns to each system c its "(finite) trace set" map $\mathsf{tr}_c : X \to \mathcal{P}A$, where A carries an initial algebra in **Sets**. This suggests a natural testing framework where: an element t of A is a test; a state $x \in X$ of a system passes a test t if and only if the trace set of x includes t (i.e. $x \models t \iff t \in \mathsf{tr}_c(x)$). An important point here is that A, carrying an initial algebra in **Sets**, usually gives a *well-founded syntax* for tests.[9]

We focus on a non-deterministic setting (i.e. $T = \mathcal{P}$) in this section and leave a probabilistic one as future work. Although the above intuition is true in probabilistic settings

[9]Recall the construction of an initial algebra in **Sets** via the initial sequence (Proposition A.1). The set A is the colimit (*union* in **Sets**) of the initial sequence $0 \to F0 \to F^20 \to \cdots$. Each F^n0 can be thought of as the set of terms with depth $\leq n$.

as well—where the 2-valued (pass/failure) observation scheme is replaced by the refined $[0,1]$-valued one—we do not know yet how to extend the current material to probabilistic settings. The difficulty is that the category $\mathcal{K}\ell(\mathcal{D})$ is not self-dual, as opposed to $\mathcal{K}\ell(\mathcal{P})$; see (5.6) below.

The canonical testing situation which captures finite trace semantics is the following one.

$$\mathcal{K}\ell(\mathcal{P})^{\mathrm{op}} \underset{Op^{\mathrm{op}}}{\overset{Op}{\underset{\cong}{\rightleftarrows}}} \mathcal{K}\ell(\mathcal{P}) \underset{J}{\overset{K}{\underset{\top}{\rightleftarrows}}} \mathbf{Sets} \qquad \overline{F}^{\mathrm{op}} \circlearrowleft \qquad F \circlearrowright \tag{5.6}$$

Here $J \dashv K$ is the canonical Kleisli adjunction. Recall the self duality $Op : \mathcal{K}\ell(\mathcal{P})^{\mathrm{op}} \xRightarrow{\cong} \mathcal{K}\ell(\mathcal{P})$ from Section 3.2. The denotation is given by (the components of) the distributive law $\lambda : F\mathcal{P} \Rightarrow \mathcal{P}F$. The following lemma establishes naturality of the denotation.

Lemma 5.8. *Let $F : \mathbf{Sets} \to \mathbf{Sets}$ be a functor which preserves weak pullbacks, and $\overline{F}$ be its lifting induced by the relation lifting (Lemma 2.3). Then the components $F\mathcal{P}X \overset{\lambda_X}{\to} \mathcal{P}FX$ of the corresponding distributive law λ also form a natural transformation*

$$F \circ K \circ Op \Longrightarrow K \circ Op \circ \overline{F}^{\mathrm{op}} \ : \ \mathcal{K}\ell(\mathcal{P})^{\mathrm{op}} \longrightarrow \mathbf{Sets} \ .$$

Proof. The desired natural transformation is obtained from another natural transformation

$$\lambda' \ : \ FK \Longrightarrow K\overline{F} \ : \ \mathcal{K}\ell(\mathcal{P}) \longrightarrow \mathbf{Sets}$$

which we describe in a moment, by post-composing the functor Op. That is, the desired one is the composite

$$FK\,Op \overset{\lambda' \circ Op}{\Longrightarrow} K\overline{F}\,Op \overset{(*)}{=} K\,Op\overline{F}^{\mathrm{op}} \ ,$$

or equivalently, in a 2-categorical presentation,

$$\begin{array}{ccccc} \mathcal{K}\ell(\mathcal{P})^{\mathrm{op}} & \xrightarrow{Op} & \mathcal{K}\ell(\mathcal{P}) & \xrightarrow{K} & \mathbf{Sets} \\ {\scriptstyle \overline{F}^{\mathrm{op}}}\downarrow & (*)/\!/ & {\scriptstyle \overline{F}}\downarrow & \not\Downarrow_{\lambda'} & \downarrow{\scriptstyle F} \\ \mathcal{K}\ell(\mathcal{P})^{\mathrm{op}} & \xrightarrow[Op]{} & \mathcal{K}\ell(\mathcal{P}) & \xrightarrow[K]{} & \mathbf{Sets} \end{array} \ .$$

Here the equality $(*)$ is the one in (3.6).

Now we describe the natural transformation λ'. Its components are given by those of λ; naturality of λ' is an easy consequence of λ's being a distributive law. Indeed, given an arrow $f : X \to Y$ in $\mathcal{K}\ell(\mathcal{P})$, the following shows that the naturality square commutes.

$$\begin{aligned} K\overline{F}f \circ \lambda_X &= \mu^{\mathcal{P}}_{FY} \circ \mathcal{P}\overline{F}f \circ \lambda_X && \text{definition of } K \\ &= \mu^{\mathcal{P}}_{FY} \circ \mathcal{P}\lambda_Y \circ \mathcal{P}Ff \circ \lambda_X && \text{definition of } \overline{F} \\ &= \mu^{\mathcal{P}}_{FY} \circ \mathcal{P}\lambda_Y \circ \lambda_{\mathcal{P}Y} \circ F\mathcal{P}f && \text{naturality of } \lambda \\ &= \lambda_Y \circ F\mu^{\mathcal{P}}_Y \circ F\mathcal{P}f && \lambda \text{ is compatible with the multiplication } \mu^{\mathcal{P}} \text{ of } \mathcal{P} \\ &= \lambda_Y \circ FKf && \text{definition of } K \end{aligned}$$

□

The previous lemma establishes that the situation (5.6) is indeed a testing situation as defined in Definition 5.1.

In the previous Section 5.1, the use of testing situations is demonstrated through comparing testing equivalence and final coalgebra semantics, both described as suitable kernel

pairs. Unfortunately this argument is not valid in the current situation (5.6), since the category $\mathcal{K}\ell(\mathcal{P})$ does not have kernel pairs.

Still, we shall claim that the situation (5.6) is "expressive," in the sense that final coalgebra semantics is captured by testing. This claim is supported by the following fact: in the current situation the two arrows tr_c and th_c simply coincide. Therefore their kernel relations—in any reasonable formalization—should coincide as well.

Proposition 5.9. *Let $X \xrightarrow{c} \overline{F}X$ be a coalgebra in $\mathcal{K}\ell(\mathcal{P})$. In the testing situation (5.6), the following arrows in $\mathcal{K}\ell(\mathcal{P})$ coincide.*

- $\mathsf{tr}_c : X \to A$, *giving the final coalgebra (trace) semantics for c.*
- $\mathsf{th}_c : X \to A$, *giving the testing semantics, i.e. the set of passed tests.*

Therefore the testing is "expressive": tests from an initial F-algebra captures trace semantics (which is via a final $\overline{F}$-coalgebra).

Here A is the carrier of an initial F-algebra, hence that of a final $\overline{F}$-coalgebra. Note that, in the general setting in Section 5.1, the codomains of tr_c and th_c need not coincide.

Proof. We shall show that the transpose

$$\mathsf{tr}_c^{\vee} \;:\; A \longrightarrow \mathcal{P}X \qquad \text{in } \mathbf{Sets}$$

of tr_c under the adjunction in (5.6) makes the diagram (5.2)—which defines $[\![\_]\!]_c$—commute. This proves $\mathsf{tr}_c^{\vee} = [\![\_]\!]_c$, hence $\mathsf{tr}_c = [\![\_]\!]_c{}^{\vee} = \mathsf{th}_c$.

First note that the transpose $\mathsf{tr}_c^{\vee} : A \to \mathcal{P}X$ is given by the arrow $Op(\mathsf{tr}_c) : A \to X$ in $\mathcal{K}\ell(\mathcal{P})$ thought of as an arrow in **Sets**. In the sequel we shall write $Op(\mathsf{tr}_c)$ for $\mathsf{tr}_c^{\vee}$.

Commutativity of the diagram (4.1)—defining tr_c—yields the following equality.

$$Op(\mathsf{tr}_c) \circ Op(J\alpha^{-1}) = Op(c) \circ Op(\overline{F}^{\mathrm{op}}\mathsf{tr}_c) \qquad \text{in } \mathcal{K}\ell(\mathcal{P}).$$

By the definition of composition in $\mathcal{K}\ell(\mathcal{P})$, it reads as follows in **Sets**.

$$\mu_X \circ \mathcal{P}(Op(\mathsf{tr}_c)) \circ Op(J\alpha^{-1}) = \mu_X \circ \mathcal{P}(Op(c)) \circ Op(\overline{F}^{\mathrm{op}}\mathsf{tr}_c) \tag{5.7}$$

We use this equality in showing that $Op(\mathsf{tr}_c)$ makes the diagram (5.2) commute.

$$\begin{aligned}
Op(\mathsf{tr}_c) \circ \alpha &= \mu_X \circ \eta_X \circ Op(\mathsf{tr}_c) \circ \alpha && \text{unit law}\\
&= \mu_X \circ \mathcal{P}(Op(\mathsf{tr}_c)) \circ \eta_A \circ \alpha && \text{naturality of } \eta\\
&= \mu_X \circ \mathcal{P}(Op(\mathsf{tr}_c)) \circ Op(J\alpha^{-1}) && Op(J\alpha^{-1}) = J\alpha = \eta_A \circ \alpha\\
&= \mu_X \circ \mathcal{P}(Op(c)) \circ Op(\overline{F}^{\mathrm{op}}\mathsf{tr}_c) && \text{by (5.7)}\\
&= \mu_X \circ \mathcal{P}(Op(c)) \circ \overline{F}Op(\mathsf{tr}_c) && Op\overline{F}^{\mathrm{op}} = \overline{F}Op, \text{ (3.6)}\\
&= \mu_X \circ \mathcal{P}(Op(c)) \circ \lambda_X \circ FOp(\mathsf{tr}_c) && \text{definition of } \overline{F}\\
&= KOp(c) \circ \lambda_X \circ FOp(\mathsf{tr}_c)\ . &&
\end{aligned}$$

Recall that M in (5.2) is now F; P in (5.2) is now KOp. This concludes the proof. □

The proposition establishes a connection between two semantics for $\overline{F}$-coalgebras in $\mathcal{K}\ell(\mathcal{P})$, namely: tr_c via a final $\overline{F}$-coalgebra, and th_c via an initial F-algebra. One may well say that it is a "degenerate" case because, as we have shown in Section 3, coinduction in $\mathcal{K}\ell(\mathcal{P})$ and induction in **Sets** are essentially the same principle. Our emphasis is more on the fact that the coincidence of induction and coinduction yields a rather uncommon example of testing situations. Testing situations are of interest in modal logic—where the underlying contravariant adjunction $S^{\mathrm{op}} \dashv P : \mathbb{A} \to \mathbb{C}^{\mathrm{op}}$ in (5.1) is often the Stone duality or one of its variants. Our example $\mathcal{K}\ell(\mathcal{P})^{\mathrm{op}} \leftrightarrows$ **Sets** here does not look like one of those familiar examples.

6. Conclusions and future work

We have developed a mathematical principle underlying "trace semantics" for various kinds of branching systems, namely coinduction in a Kleisli category. This general view is supported by a technical result that a final coalgebra in a Kleisli category is induced by an initial algebra in **Sets**.

The possible instantiations of our generic framework include non-deterministic systems and probabilistic systems, but do *not* yet include systems with both non-deterministic and probabilistic branching. The importance of having both of these branchings in system verification has been claimed by many authors e.g. [48, 60], with an intuition that probabilistic branching models the choices "made by the system, i.e. on *our* side," while (coarser) non-deterministic choices are "made by the (unknown) environment of the system, i.e. on the *adversary's* side." A typical example of such systems is given by *probabilistic automata* introduced by Segala [48].

In fact this combination of non-deterministic and probabilistic branching is a notoriously difficult one from a theoretical point of view [8, 54, 59]: many mathematical tools that are useful in a purely non-deterministic or probabilistic setting cease to work in the presence of both. For our framework of generic trace semantics, the problem is that we could not find a suitable monad T with an order structure.

We have used the order-enriched structure of a Kleisli category (expressing "more possibilities") to obtain the initial algebra-final coalgebra coincidence result. However, an order structure is not the only one that can yield such coincidence: other examples include metric, quasi-metric and quantale-enriched structures (in increasing generality). See e.g. [10, 56] for the potential use of such enriched structures in a coalgebraic setting. The relation of the current work to such structures is yet to be investigated.

In the discipline of process algebra, a system is represented by an *algebraic* term (such as $a.P \parallel a.Q$) and a structural operational semantics (SOS) rule determines its dynamics, that is, its *coalgebraic* structure. This is where "algebra meets coalgebra" and the interaction is studied e.g. in [4, 30, 55]. In our recent work [18] we claim the importance of the *microcosm principle* in this context and provide a "general compositionality theorem": under suitable assumptions, the final coalgebra semantics is compatible with the algebraic structure. The results of the current paper say that the final coalgebra semantics can be interpreted as finite trace semantics, hence the result in [18] also yields a general compositionality result for trace semantics.

In this paper we have included some material—on possibly-infinite traces and testing situations—which, unfortunately, we have worked out only in a non-deterministic setting. A fully general account on these topics is left as future work.

Finally, there are so many different process semantics for branching systems, between two edges of bisimilarity and trace equivalence in the linear time-branching time spectrum [57]. How to capture them in a coalgebraic setting is, we believe, an important and challenging question.

Acknowledgment

Thanks are due to Jiří Adámek, Chris Heunen, Stefan Milius, Tarmo Uustalu and the anonymous referees for helpful discussions and comments.

References

[1] S. Abramsky and A. Jung. Domain theory. In S. Abramsky, D.M. Gabbai, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 3, pages 1–168. Oxford Univ. Press, 1994.
[2] J. Adámek and V. Koubek. Least fixed point of a functor. *Journ. Comp. Syst. Sci*, 19(2):163–178, 1979.
[3] M. Barr and C. Wells. *Toposes, Triples and Theories*. Springer, Berlin, 1985. Available online.
[4] F. Bartels. *On generalised coinduction and probabilistic specification formats. Distributive laws in coalgebraic modelling*. PhD thesis, Free Univ. Amsterdam, 2004.
[5] M.M. Bonsangue and A. Kurz. Duality for logics of transition systems. In V. Sassone, editor, *FoSSaCS*, volume 3441 of *Lect. Notes Comp. Sci.*, pages 455–469. Springer, 2005.
[6] M.M. Bonsangue and A. Kurz. Presenting functors by operations and equations. In L. Aceto and A. Ingólfsdóttir, editors, *FoSSaCS*, volume 3921 of *Lect. Notes Comp. Sci.*, pages 172–186. Springer, 2006.
[7] F. Borceux. *Handbook of Categorical Algebra*, volume 50, 51 and 52 of *Encyclopedia of Mathematics*. Cambridge Univ. Press, 1994.
[8] L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. PhD thesis, Radboud Univ. Nijmegen, 2006.
[9] M.P. Fiore. *Axiomatic Domain Theory in Categories of Partial Maps*. Distinguished Dissertations in Computer Science. Cambridge Univ. Press, 1996.
[10] M.P. Fiore. A coinduction principle for recursive data types based on bisimulation. *Inf. & Comp.*, 127(2):186–198, 1996.
[11] M.M. Fokkinga. Monadic maps and folds for arbitrary datatypes. *Memoranda Informatica, University of Twente*, 94–28, 1994.
[12] P.J. Freyd. Algebraically complete categories. In A. Carboni, M.C. Pedicchio, and G. Rosolini, editors, *Como Conference on Category Theory*, number 1488 in Lect. Notes Math., pages 95–104. Springer, Berlin, 1991.
[13] P.J. Freyd. Remarks on algebraically compact categories. In M.P. Fourman, P.T. Johnstone, and A.M. Pitts, editors, *Applications of Categories in Computer Science*, number 177 in LMS, pages 95–106. Cambridge Univ. Press, 1992.
[14] I. Hasuo. Generic forward and backward simulations. In C. Baier and H. Hermanns, editors, *International Conference on Concurrency Theory (CONCUR 2006)*, volume 4137 of *Lect. Notes Comp. Sci.*, pages 406–420. Springer, Berlin, 2006.
[15] I. Hasuo and B. Jacobs. Coalgebraic trace semantics for probabilistic systems. In P. Mosses, J. Power, and M. Seisenberger, editors, *CALCO-jnr Workshop*, 2005.
[16] I. Hasuo and B. Jacobs. Context-free languages via coalgebraic trace semantics. In J.L. Fiadeiro, N. Harman, M. Roggenbach, and J.J.M.M. Rutten, editors, *International Conference on Algebra and Coalgebra in Computer Science (CALCO'05)*, volume 3629 of *Lect. Notes Comp. Sci.*, pages 213–231. Springer, Berlin, 2005.
[17] I. Hasuo, B. Jacobs, and A. Sokolova. Generic trace theory. In N. Ghani and A.J. Power, editors, *International Workshop on Coalgebraic Methods in Computer Science (CMCS 2006)*, volume 164 of *Elect. Notes in Theor. Comp. Sci.*, pages 47–65. Elsevier, Amsterdam, 2006.
[18] I. Hasuo, B. Jacobs, and A. Sokolova. The microcosm principle and concurrency in coalgebras, 2007. Preprint, available from `http://www.cs.ru.nl/~ichiro/papers`.

[19] I. Hasuo and Y. Kawabe. Probabilistic anonymity via coalgebraic simulations. In R. De Nicola, editor, *European Symposium on Programming (ESOP 2007)*, volume 4421 of *Lect. Notes Comp. Sci.*, pages 379–394. Springer, 2007.
[20] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Inf. & Comp.*, 145:107–152, 1998.
[21] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
[22] J. Hughes and B. Jacobs. Simulations in coalgebra. *Theor. Comp. Sci.*, 327(1-2):71–108, 2004.
[23] B. Jacobs. *Categorical Logic and Type Theory*. North Holland, Amsterdam, 1999.
[24] B. Jacobs. Trace semantics for coalgebras. In J. Adámek and S. Milius, editors, *Coalgebraic Methods in Computer Science*, volume 106 of *Elect. Notes in Theor. Comp. Sci.* Elsevier, Amsterdam, 2004.
[25] B. Jacobs and J.J.M.M. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, 62:222–259, 1997.
[26] B. Jacobs. Introduction to coalgebra. Towards mathematics of states and observations. Draft of a book, `www.cs.ru.nl/B.Jacobs/PAPERS/index.html`, 2005.
[27] C.B. Jay. A semantics for shape. *Science of Comput. Progr.*, 25:251–283, 1995.
[28] G.M. Kelly. *Basic Concepts of Enriched Category Theory*. Number 64 in LMS. Cambridge Univ. Press, 1982.
[29] M. Kick, A.J. Power, and A. Simpson. Coalgebraic semantics for timed processes. *Inf. & Comp.*, 204(4):588–609, 2006.
[30] B. Klin. From bialgebraic semantics to congruence formats. In *Workshop on Structural Operational Semantics (SOS 2004)*, volume 128 of *Elect. Notes in Theor. Comp. Sci.*, pages 3–37, 2005.
[31] B. Klin. Bialgebraic operational semantics and modal logic. In *Logic in Computer Science*, pages 336–345. IEEE Computer Society, 2007.
[32] B. Klin. Coalgebraic modal logic beyond **Sets**. In *MFPS XXIII*, volume 173, pages 177–201. Elsevier, Amsterdam, 2007.
[33] A. Kock. Monads on symmetric monoidal closed categories. *Arch. Math.*, XXI:1–10, 1970.
[34] C. Kupke, A. Kurz, and Y. Venema. Stone coalgebras. *Theor. Comp. Sci.*, 327(1-2):109–134, 2004.
[35] C. Kupke, A. Kurz, and D. Pattinson. Algebraic semantics for coalgebraic logics. *Elect. Notes in Theor. Comp. Sci.*, 106:219–241, 2004.
[36] A. Kurz. *Logics for Coalgebras and Applications to Computer Science*. PhD thesis, Universität München, April 2000.
[37] A. Kurz. Coalgebras and their logics. *SIGACT News*, 37(2):57–77, 2006.
[38] F.W. Lawvere. Metric spaces, generalized logic, and closed categories. *Seminario Matematico e Fisico. Rendiconti di Milano*, 43:135–166, 1973. Reprinted in *Theory and Applications of Categories*, 1:1–37, 2002.
[39] M. Lenisa, A.J. Power, and H. Watanabe. Distributivity for endofunctors, pointed and co-pointed endofunctors, monads and comonads. In H. Reichel, editor, *Coalgebraic Methods in Computer Science*, volume 33 of *Elect. Notes in Theor. Comp. Sci.* Elsevier, Amsterdam, 2000.
[40] M. Lenisa, J. Power, and H. Watanabe. Category theory for operational semantics. *Theor. Comp. Sci.*, 327(1–2):135–154, 2004.
[41] N. Lynch and F. Vaandrager. Forward and backward simulations. I. Untimed systems. *Inf. & Comp.*, 121(2):214–233, 1995.
[42] S. Mac Lane. *Categories for the Working Mathematician*. Springer, Berlin, 2nd edition, 1998.
[43] P.S. Mulry. Lifting theorems for Kleisli categories. In *Mathematical Foundations of Programming Semantics (MFPS IX)*, pages 304–319, London, UK, 1994. Springer-Verlag.
[44] A. Pardo. Fusion of recursive programs with computational effects. *Theor. Comp. Sci.*, 260(1–2):165–207, 2001.
[45] D. Pavlović, M. Mislove, and J.B. Worrell. Testing semantics: connecting processes and process logics. In M. Johnson and V. Vene, editors, *Algebraic Methodology and Software Technology (AMAST 2006)*, volume 4019 of *Lect. Notes Comp. Sci.* Springer, 2006.
[46] J. Power and D. Turi. A coalgebraic foundation for linear time semantics. In *Category Theory and Computer Science*, volume 29 of *Elect. Notes in Theor. Comp. Sci.* Elsevier, Amsterdam, 1999.
[47] J.J.M.M. Rutten. Universal coalgebra: a theory of systems. *Theor. Comp. Sci.*, 249:3–80, 2000.
[48] R. Segala. *Modeling and verification of randomized distributed real-time systems*. PhD thesis, MIT, 1995.

[49] R. Segala. A compositional trace-based semantics for probabilistic automata. In *International Conference on Concurrency Theory (CONCUR '95)*, pages 234–248. Springer-Verlag, 1995.
[50] A.K. Simpson. Recursive types in Kleisli categories. Unpublished paper, available at `http://homepages.inf.ed.ac.uk/als/Research/`, 1992.
[51] M.B. Smyth and G.D. Plotkin. The category theoretic solution of recursive domain equations. *SIAM Journ. Comput.*, 11:761–783, 1982.
[52] A. Sokolova. *Coalgebraic Analysis of Probabilistic Systems*. PhD thesis, Techn. Univ. Eindhoven, 2005.
[53] M. Stoelinga and F.W. Vaandrager. A testing scenario for probabilistic automata. In J.C.M. Baeten, J.K. Lenstra, J. Parrow, and G.J. Woeginger, editors, *ICALP*, volume 2719 of *Lect. Notes Comp. Sci.*, pages 464–477. Springer, 2003.
[54] R. Tix, K. Keimel, and G.D. Plotkin. Semantic domains for combining probability and non-determinism. *Elect. Notes in Theor. Comp. Sci.*, 129:1–104, 2005.
[55] D. Turi and G. Plotkin. Towards a mathematical operational semantics. In *Logic in Computer Science*, pages 280–291. IEEE, Computer Science Press, 1997.
[56] D. Turi and J.J.M.M. Rutten. On the foundations of final semantics: non-standard sets, metric spaces and partial orders. *Math. Struct. in Comp. Sci.*, 8(5):481–540, 1998.
[57] R.J. van Glabbeek. The linear time–branching time spectrum I; the semantics of concrete, sequential processes. In J.A. Bergstra, A. Ponse, and S.A. Smolka, editors, *Handbook of Process Algebra*, chapter 1, pages 3–99. Elsevier, 2001. Available at `http://boole.stanford.edu/pub/spectrum1.ps.gz`.
[58] R.J. van Glabbeek, S.A. Smolka, and B. Steffen. Reactive, generative, and stratified models of probabilistic processes. *Inf. & Comp.*, 121:59–80, 1995.
[59] D. Varacca and G. Winskel. Distributing probabililty over nondeterminism. *Math. Struct. in Comp. Sci.*, 16(1):87–113, 2006.
[60] M.Y. Vardi. Automatic verification of probabilistic concurrent finite-state programs. In *FOCS '85*, pages 327–338, 1985.

Appendix A. Preliminaries

A.1. **Initial/final sequences.** Here we recall the standard construction [2] of the initial algebra (or the final coalgebra) via the initial (or final) sequence. Notice that the base category need not be **Sets**.

Let $\mathbb{C}$ be a category with initial object 0, and $F : \mathbb{C} \to \mathbb{C}$ be an endofunctor. The *initial sequence*[10] of F is a diagram

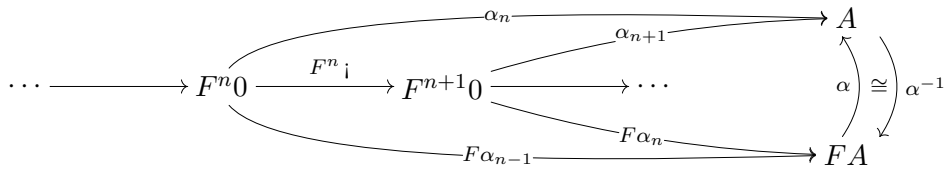
$$0 \xrightarrow{\ ¡\ } F0 \xrightarrow{\ F¡\ } \cdots \xrightarrow{\ F^{n-1}¡\ } F^n0 \xrightarrow{\ F^n¡\ } \cdots$$

where $¡ : 0 \to X$ is the unique arrow.

Now assume that:

- the initial sequence has an ω-colimit[11] $(F^n0 \xrightarrow{\alpha_n} A)_{n<\omega}$;
- the functor F preserves that ω-colimit.

Then we have two cocones $(\alpha_n)_{n<\omega}$ and $(F\alpha_{n-1})_{n<\omega}$ over the initial sequence. Moreover, the latter is again a colimit: hence we have mediating isomorphisms between these cones.



[10] In this paper we consider only initial/final sequences of length ω.
[11] An ω-colimit is a colimit of a diagram whose shape is the ordinal ω.

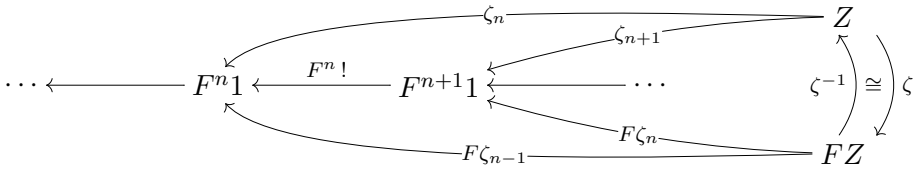
Proposition A.1. *The F-algebra $\alpha : FA \xrightarrow{\cong} A$ is initial.*

Proof. For future reference we prove the dual result: see Proposition A.2. □

The dual of this construction yields a final F-coalgebra. Assume that the base category $\mathbb{C}$ has a terminal object 1. The *final sequence* of F is

$$1 \xleftarrow{\ !\ } F1 \xleftarrow{\ F!\ } \cdots \xleftarrow{\ F^{n-1}!\ } F^n1 \xleftarrow{\ F^n!\ } \cdots,$$

where $! : X \to 1$ is the unique arrow. Assume that it has an ω^{op}-limit $(Z \xrightarrow{\zeta_n} F^n1)_{n<\omega}$, and also that F preserves that ω^{op}-limit. We have the following situation.



Proposition A.2. *The coalgebra $\zeta : Z \xrightarrow{\cong} FZ$ is final.*

Proof. Any F-coalgebra $c : X \to FX$ induces a cone $(X \xrightarrow{\beta_n} F^n1)_{n<\omega}$ over the final sequence in the following way.

$$\beta_0 = \ ! : X \longrightarrow 1 \ , \qquad \beta_{n+1} = F\beta_n \circ c \ .$$

Now we can prove the following: for an arrow $f : X \to Z$, f is a morphism of coalgebras from c to ζ if and only if f is a mediating arrow from the cone $(\beta_n)_{n<\omega}$ to the limit $(\zeta_n)_{n<\omega}$. Hence such a morphism of coalgebras uniquely exists. □

It is easy to see that every shapely functor in **Sets** preserves ω-colimits and ω^{op}-limits. Hence we have the following.

Lemma A.3. *A shapely functor F has both an initial algebra and a final coalgebra in* **Sets**. □

A.2. **limit-colimit coincidence.** We recall some relevant notions and results from [51]. The idea is that in a suitable order-enriched setting, (co)limits are equivalently described as an order-theoretic notion of **O**-*(co)limits*. Due to the inherent coincidence between **O**-limits and **O**-colimits, we also obtain the so-called *limit-colimit coincidence*.

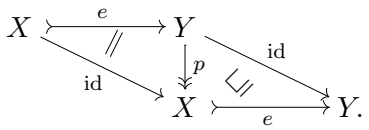
$$\begin{array}{ccc} \text{limit} & & \text{colimit} \\ \| & & \| \\ \mathbf{O}\text{-limit} & \xrightarrow[\text{obvious coincidence}]{} & \mathbf{O}\text{-colimit} \end{array}$$

The notions of **O**-(co)limits are stated in terms of *embedding-projection pairs* which we can define in an order-enriched category. In the sequel we assume the **Cppo**-enriched structure.

Definition A.4 (Embedding-projection pairs)**.** Let $\mathbb{C}$ be a **Cppo**-enriched category. A pair of arrows

$$X \underset{p}{\overset{e}{\rightleftarrows}} Y$$

in $\mathbb{C}$ is said to be an *embedding-projection pair* if we have $p \circ e = \mathrm{id}$ and $e \circ p \sqsubseteq \mathrm{id}$. Diagrammatically presented,



By $p \circ e = \mathrm{id}$ we automatically have that e is a mono and p is an epi. Both split.

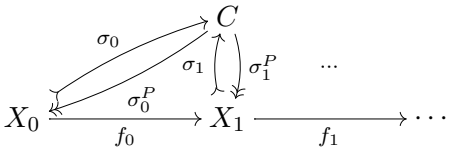
Proposition A.5. *Let $(e, p), (e', p') : X \rightleftarrows Y$ be two embedding-projection pairs with the same (co)domains. Then $e \sqsubseteq e'$ holds if and only if $p' \sqsubseteq p$. As a consequence, one component of an embedding-projection pair determines the other.* □

This proposition justifies the notation e^P for the projection corresponding to a given embedding e, and p^E for the embedding corresponding to a given projection p. It is easy to check that

$$(e \circ f)^P = f^P \circ e^P \qquad \text{and} \qquad (p \circ q)^E = q^E \circ p^E \ .$$

Definition A.6 (**O**-(co)limits)**.** Let $X_0 \stackrel{f_0}{\to} X_1 \stackrel{f_1}{\to} \cdots$ be an ω-chain in a **Cppo**-enriched $\mathbb{C}$. A cocone $(X_n \stackrel{\sigma_n}{\to} C)_{n<\omega}$ over this chain is said to be an **O**-*colimit* if:

- each σ_n is an embedding;
- the sequence of arrows $(C \stackrel{\sigma_n^P}{\twoheadrightarrow} X_n \stackrel{\sigma_n}{\rightarrowtail} C)_{n<\omega}$ is increasing. Moreover its join taken in the cpo $\mathbb{C}(C, C)$ is id_C.



Dually, a cone $(C \stackrel{\gamma_n}{\to} Y_n)_{n<\omega}$ over an ω^{op}-chain $Y_0 \stackrel{g_0}{\leftarrow} Y_1 \stackrel{g_1}{\leftarrow} \cdots$ is an **O**-*limit* if: each γ_n is a projection, and the sequence $(\gamma_n^E \circ \gamma_n : C \to C)_{n<\omega}$ is increasing and its join is id_C.

The following proposition establishes the equivalence between (co)limits and **O**-(co)limits. For its full proof the reader is referred to [51].

Proposition A.7 (Propositions A, B, C, D in [51])**.** *Let $X_0 \stackrel{e_0}{\to} X_1 \stackrel{e_1}{\to} \cdots$ be an ω-chain where each e_n is an embedding.*

(1) Let $(X_n \stackrel{\sigma_n}{\to} C)_{n<\omega}$ be the colimit over the chain. Then each σ_n is also an embedding. Moreover, $(\sigma_n)_{n<\omega}$ is an **O**-*colimit.*

(2) Conversely, an **O**-*colimit $(X_n \stackrel{\sigma_n}{\to} C)_{n<\omega}$ over the chain is a colimit.*

Dually, let $X_0 \stackrel{p_0}{\leftarrow} X_1 \stackrel{p_1}{\leftarrow} \cdots$ be an ω^{op}-chain where each p_n is a projection.

(3) Let $(D \stackrel{\tau_n}{\to} X_n)_{n<\omega}$ be a limit over the chain. Then each τ_n is also a projection. Moreover $(\tau_n)_{n<\omega}$ is an **O**-*limit.*

(4) Conversely, an **O**-*limit $(D \stackrel{\tau_n}{\to} X_n)_{n<\omega}$ over the chain is a limit.*

Proof. For later reference we present the proof of (4). Let $(B \stackrel{\beta_n}{\to} X_n)_{n<\omega}$ be an arbitrary cone over the chain $X_0 \stackrel{p_0}{\leftarrow} X_1 \stackrel{p_1}{\leftarrow} \cdots$. First we prove the uniqueness of a mediating map

$f : B \to D$.

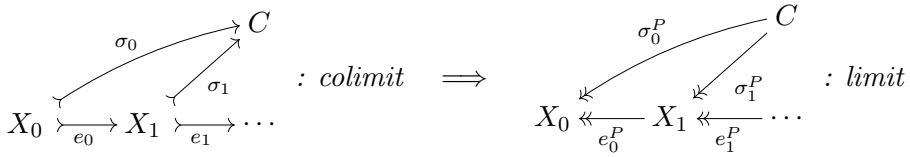
$$\begin{aligned} f = \mathrm{id}_D \circ f &= \big(\bigsqcup_{n<\omega}(\tau_n^E \circ \tau_n)\big) \circ f && (\tau_n)_{n<\omega} \text{ is an } \mathbf{O}\text{-limit} \\ &= \bigsqcup_{n<\omega}(\tau_n^E \circ \tau_n \circ f) && \text{composition is continuous} \\ &= \bigsqcup_{n<\omega}(\tau_n^E \circ \beta_n) && f \text{ is mediating .} \end{aligned}$$

We conclude the proof by showing that the sequence $(\tau_n^E \circ \beta_n)_{n<\omega}$ is increasing, hence such f indeed exists.

$$\tau_n^E \circ \beta_n \;=\; \tau_n^E \circ p_n \circ \beta_{n+1} \;=\; \tau_{n+1}^E \circ p_n^E \circ p_n \circ \beta_{n+1} \;\sqsubseteq\; \tau_{n+1}^E \circ \beta_{n+1}$$

The last inequality holds because $p_n^E \circ p_n \sqsubseteq \mathrm{id}$ from the definition of embedding-projection pairs. □

Theorem A.8 (Limit-colimit coincidence)**.** *Let $X_0 \overset{e_0}{\to} X_1 \overset{e_1}{\to} \cdots$ be an ω-chain where each e_n is an embedding, and $(X_n \overset{\sigma_n}{\to} C)_{n<\omega}$ be the colimit over the chain. Then each σ_n is an embedding, and the cone $(C \overset{\sigma_n^P}{\to} X_n)_{n<\omega}$ is a limit over the ω^{op}-chain $X_0 \overset{e_0^P}{\leftarrow} X_1 \overset{e_1^P}{\leftarrow} \cdots$.*



Dually, the limit of an ω^{op}-chain of projections consists of projections. By taking the corresponding embeddings we obtain a colimit of an ω-chain of embeddings.

Proof. We prove the first statement. By Proposition A.7 each σ_n is an embedding, and moreover $(\sigma_n)_{n<\omega}$ is an $\mathbf{O}$-colimit. Now obviously $(\sigma_n^P)_{n<\omega}$ is a cone over $X_0 \overset{e_0^P}{\leftarrow} X_1 \overset{e_1^P}{\leftarrow} \cdots$. Here we use the inherent coincidence of $\mathbf{O}$-(co)limits: namely, the condition that $(\sigma_n)_{n<\omega}$ is an $\mathbf{O}$-colimit is exactly the same as that $(\sigma_n^P)_{n<\omega}$ is an $\mathbf{O}$-limit. We use Proposition A.7 to conclude the proof. □

The Microcosm Principle and Concurrency in Coalgebra

Ichiro Hasuo[1,3,4], Bart Jacobs[1,⋆], and Ana Sokolova[2,⋆⋆]

[1] Radboud University Nijmegen, The Netherlands
[2] University of Salzburg, Austria
[3] RIMS, Kyoto University, Japan
[4] PRESTO Research Promotion Program, Japan Science and Technology Agency

Abstract. Coalgebras are categorical presentations of state-based systems. In investigating parallel composition of coalgebras (realizing *concurrency*), we observe that the same algebraic theory is interpreted in two different domains in a nested manner, namely: in the category of coalgebras, and in the final coalgebra as an object in it. This phenomenon is what Baez and Dolan have called the *microcosm principle*, a prototypical example of which is "a monoid in a monoidal category." In this paper we obtain a formalization of the microcosm principle in which such a nested model is expressed categorically as a suitable lax natural transformation. An application of this account is a general compositionality result which supports modular verification of complex systems.

1 Introduction

Design of systems with *concurrency* is nowadays one of the mainstream challenges in computer science [19]. Concurrency is everywhere: with the Internet being the biggest example and multi-core processors the smallest; also in a modular, component-based architecture of a complex system its components collaborate in a concurrent manner. However, numerous difficulties have been identified in getting concurrency right. For example, a system's exponentially growing complexity is one of the main obstacles. One way to cope with it is a *modular* verification method in which correctness of the whole system $\mathcal{C}_1 \parallel \cdots \parallel \mathcal{C}_n$ is established using correctness of each component $\mathcal{C}_i$. *Compositionality*—meaning that the behavior of $\mathcal{C} \parallel \mathcal{D}$ is determined by the behavior of $\mathcal{C}$ and that of $\mathcal{D}$—is an essential property for such a modular method to work.

Coalgebras as systems. This paper is a starting point of our research program aimed at better understanding of the mathematical nature of concurrency. In its course we shall use *coalgebras* as presentations of systems to be run in parallel. The use of coalgebras as an appropriate abstract model of state-based systems is increasingly established [26,11]; the notion's mathematical simplicity and clarity provide us with a sound foundation

⋆ Also part-time at Technical University Eindhoven, The Netherlands.
⋆⋆ Supported by the Austrian Science Fund (FWF) project no. P18913-N15.

R. Amadio (Ed.): FOSSACS 2008, LNCS 4962, pp. 246–260, 2008.

for our exploration. The following table summarizes how ingredients of the theory of systems are presented as coalgebraic constructs.

	system	behavior-preserving map	behavior
		morphism of coalgebras	by coinduction
coalgebraically	coalgebra $\begin{array}{c} FX \\ \uparrow \\ X \end{array}$	$\begin{array}{ccc} FX & \xrightarrow{Ff} & FY \\ \uparrow & & \uparrow \\ X & \xrightarrow[f]{} & Y \end{array}$	$\begin{array}{ccc} FX & - - \to & FZ \\ {\scriptstyle c}\uparrow & & {\scriptstyle \text{final}}\uparrow{\scriptstyle \cong} \\ X & \underset{\mathsf{beh}(c)}{- - \to} & Z \end{array}$ (1)

This view of "coalgebras as systems" has been successfully applied in the category $\mathbf{Sets}$ of sets and functions, in which case the word "behavior" in (1) refers (roughly) to bisimilarity. Our recent work [6, 5] has shown that "behavior" can also refer to trace semantics by moving from $\mathbf{Sets}$ to a suitable Kleisli category.

Compositionality in coalgebras. We start with the following question: what is "compositionality" in this coalgebraic setting? Conventionally compositionality is expressed as: $\mathcal{C} \sim \mathcal{C}'$ and $\mathcal{D} \sim \mathcal{D}'$ implies $\mathcal{C} \parallel \mathcal{D} \sim \mathcal{C}' \parallel \mathcal{D}'$, where the relation $\sim$ denotes the behavioral equivalence of interest. If this is the case the relation $\sim$ is said to be a *congruence*, with its oft-heard instance being "bisimilarity is a congruence."

When we interpret "behavior" in compositionality as the coalgebraic behavior induced by coinduction (see (1)), the following equation comes natural as a coalgebraic presentation of compositionality.

$$\mathsf{beh}\left(\begin{array}{c} FX \\ c\uparrow \\ X \end{array} \;\middle\|\; \begin{array}{c} FY \\ d\uparrow \\ Y \end{array}\right) = \mathsf{beh}\left(\begin{array}{c} FX \\ c\uparrow \\ X \end{array}\right) \;\middle\|\; \mathsf{beh}\left(\begin{array}{c} FY \\ d\uparrow \\ Y \end{array}\right) \tag{2}$$

But a closer look reveals that the two "parallel composition operators" $\parallel$ in the equation have in fact different types: the first one $\mathbf{Coalg}_F \times \mathbf{Coalg}_F \to \mathbf{Coalg}_F$ combines systems (as coalgebras) and the second one $Z \times Z \to Z$ combines behavior (as states of the final coalgebra).[1] Moreover, the two domains are actually nested: the latter one $Z \xrightarrow{\cong} FZ$ is an object of the former one $\mathbf{Coalg}_F$.

The microcosm principle. What we have just observed is one instance—probably the first one explicitly claimed in computer science—of the *microcosm principle* as it is called by Baez and Dolan [1]. It refers to a phenomenon that the same algebraic theory (or algebraic "specification," consisting of operations and equations) is interpreted twice in a nested manner, once in a category $\mathbb{C}$ and the other time in its object $X \in \mathbb{C}$. This is not something very unusual, because "a monoid in a monoidal category" constitutes a prototypical example.

[1] At this stage the presentation remains sloppy for the sake of simplicity. Later in technical sections the first composition operator will be denoted by $\otimes$; and the second composition operator will have the type $Z \otimes Z \to Z$ instead of $Z \times Z \to Z$.

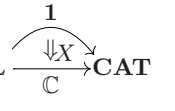
monoidal category $\mathbb{C}$		monoid $X \in \mathbb{C}$
$\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$	multiplication	$X \otimes X \xrightarrow{\mu} X$
$I \in \mathbb{C}$	unit	$I \xrightarrow{\eta} X$
$I \otimes X \cong X \cong X \otimes I$	unit law	$X \to X \otimes X \leftarrow X$, $X \otimes I \to X \otimes X \leftarrow I \otimes X$, $X \to X$
$X \otimes (Y \otimes Z) \cong (X \otimes Y) \otimes Z$	associativity law	$X \otimes X \otimes X \to X \otimes X$, $X \otimes X \to X$

(3)

Notice here that the outer operation $\otimes$ appears in the formulation of the inner operation μ. Moreover, to be precise, in the inner "equations" the outer isomorphisms should be present in suitable places. Hence this monoid example demonstrates that, in such nested algebraic structures, the inner structure depends on the outer. What is a mathematically precise formalization of such nested models? Answering this question is a main goal of this paper.

Such a formalization has been done in [1] when algebraic structures are specified in the form of *opetopes*. Here instead we shall formalize the microcosm principle for *Lawvere theories* [18], whose role as categorical representation of algebraic theories has been recognized in theoretical computer science.

As it turns out, our formalization looks like the situation on the right. Here $\mathbb{L}$ is a category (a Lawvere theory) representing an algebraic theory; an outer model $\mathbb{C}$ is a product-preserving functor; and an inner model X is a lax natural transformation. The whole setting is 2-categorical: 2-categories (categories in categories) serve as an appropriate basis for the microcosm principle (algebras in algebras).



Applications to coalgebras: Parallel composition via sync. The categorical account we have sketched above shall be applied to our original question about parallel composition of coalgebras. As a main application we prove a *generic compositionality theorem*. For an arbitrary algebraic theory $\mathbb{L}$, compositionality like (2) is formulated as follows: the "behavior" functor $\mathsf{beh} : \mathbf{Coalg}_F \to \mathbb{C}/Z$ via coinduction preserves an $\mathbb{L}$-structure. This general form of compositionality holds if: $\mathbb{C}$ has an $\mathbb{L}$-structure and $F : \mathbb{C} \to \mathbb{C}$ *lax*-preserves the $\mathbb{L}$-structure.

Turning back to the original setting of (2), these general assumptions read roughly as follows: the base category $\mathbb{C}$ has a binary operation $\|$; and the endofunctor F comes with a natural transformation $\mathsf{sync} : FX \parallel FY \to F(X \parallel Y)$. Essentially, this sync is what lifts $\|$ on $\mathbb{C}$ to $\|$ on $\mathbf{Coalg}_F$, hence "parallel composition via sync." It is called a *synchronization* because it specifies the way two systems synchronize with each other. In fact, for a fixed functor F there can be different choices of sync (such as CSP-style vs. CCS-style), which in turn yield different "parallel composition" operators on the category $\mathbf{Coalg}_F$.

Related work. Our interest is pretty similar to that of studies of *bialgebraic structures* in computer science (such as [27, 3, 15, 14, 12, 16]), in the sense that we are also concerned about algebraic structures on coalgebras as systems. Our current framework is distinguished in the following aspects.

First, we handle *equations* in an algebraic theory as an integral part of our approach. Equations such as associativity and commutativity appear explicitly as commutative diagrams in a Lawvere theory $\mathbb{L}$. We benefit from this explicitness in e.g. spelling out a condition for the generic associativity result (Theorem 2.4). In contrast, in the bialgebraic studies an algebraic theory is presented either by an endofunctor $X \mapsto \coprod_{\sigma\in\Sigma} X^{|\sigma|}$ or by a monad T. In the former case equations are simply not present; in the latter case equations are there but only implicitly.

Secondly and more importantly, by considering higher-dimensional, nested algebraic structures, we can now compose different coalgebras as well as different states of the same coalgebra. In this way the current work can be seen as a higher-dimensional extension of the existing bialgebraic studies (which focus on "inner" algebraic structures).

Organization of the paper. We shall not dive into our 2-categorical exploration from the beginning. In Section 2, we instead focus on one specific algebraic theory, namely the one for parallel composition of systems. Our emphasis there is on the fact that the sync natural transformation essentially gives rise to parallel composition $\|$, and the fact that equational properties of $\|$ (such as associativity) can be reduced to the corresponding equational properties of sync.

These concrete observations will provide us with intuition for abstract categorical constructs in Section 3, where we formalize the microcosm principle for an arbitrary Lawvere theory $\mathbb{L}$. Results on coalgebras such as compositionality are proved here in their full generality and abstraction.

In this paper we shall focus on *strict* algebraic structures on categories in order to avoid complicated coherence issues. This means for example that we only consider *strict* monoidal categories for which the isomorphisms in (3) are in fact equalities. However, we have also obtained some preliminary observations on relaxed ("pseudo" or "strong") algebraic structures: see Section 3.3.

2 Parallel Composition of Coalgebras

2.1 Parallel Composition Via sync Natural Transformation

Let us start with the equation (2), a coalgebraic representation of compositionality. The operator $\|$ on the left is of type $\mathbf{Coalg}_F \times \mathbf{Coalg}_F \to \mathbf{Coalg}_F$. It is natural to require functoriality of this operation, making it a *bifunctor*. A bifunctor—especially an associative one which we investigate in Section 2.3—plays an important role in various applications of category theory. Usually such an (associative) bifunctor is called a *tensor* and denoted by $\otimes$, a convention that we also follow. Therefore the "compositionality" statement now looks as follows.

$$\mathsf{beh}\begin{pmatrix} FX \\ c\!\uparrow \\ X \end{pmatrix} \otimes \begin{matrix} FY \\ d\!\uparrow \\ Y \end{matrix} = \mathsf{beh}\begin{pmatrix} FX \\ c\!\uparrow \\ X \end{pmatrix} \Big\| \mathsf{beh}\begin{pmatrix} FY \\ d\!\uparrow \\ Y \end{pmatrix} \tag{4}$$

The first question is: when do we have such a tensor $\otimes$ on $\mathbf{Coalg}_F$? In many applications of coalgebras, it is obtained by lifting a tensor $\otimes$ on the base category $\mathbb{C}$ to $\mathbf{Coalg}_F$.[2] Such a lifting is possible in presence of a natural transformation

$$FX \otimes FY \overset{\mathsf{sync}_{X,Y}}{\longrightarrow} F(X \otimes Y), \quad \text{used in} \quad \begin{array}{c} FX \\ c\uparrow \\ X \end{array} \otimes \begin{array}{c} FY \\ d\uparrow \\ Y \end{array} := \begin{array}{c} F(X \otimes Y) \\ \uparrow \mathsf{sync}_{X,Y} \\ FX \otimes FY \\ \uparrow c \otimes d \\ X \otimes Y \end{array} . \tag{5}$$

We shall call this sync a *synchronization* because its computational meaning is indeed a specification of the way two systems synchronize. This will be illustrated in the coming examples.

Once we have an outer parallel composition $\otimes$, an inner operator $\|$ which composes behavior (i.e. states of the final coalgebra) is also obtained immediately by coinduction as on the right. Compositionality (4) is also straightforward by finality: both sides of the equation are the unique coalgebra morphism from $c \otimes d$ to the final ζ. The following theorem summarizes the observations so far.

$$\begin{array}{ccc} F(Z \otimes Z) & \dashrightarrow & FZ \\ {\scriptstyle \zeta \otimes \zeta}\uparrow & & \uparrow{\scriptstyle \zeta} \\ Z \otimes Z & \overset{\|}{\dashrightarrow} & Z \end{array}$$

Theorem 2.1 (Coalgebraic compositionality). *Assume that a category $\mathbb{C}$ has a tensor $\otimes : \mathbb{C} \times \mathbb{C} \to \mathbb{C}$ and an endofunctor $F : \mathbb{C} \to \mathbb{C}$ has a natural transformation $\mathsf{sync}_{X,Y} : FX \otimes FY \to F(X \otimes Y)$. If moreover there exists a final F-coalgebra, then:*

1. *The tensor $\otimes$ on $\mathbb{C}$ lifts to an "outer" composition operator $\otimes : \mathbf{Coalg}_F \times \mathbf{Coalg}_F \to \mathbf{Coalg}_F$.*
2. *We obtain an "inner" composition operator $\| : Z \otimes Z \to Z$ by coinduction.*
3. *Between the two composition operators the compositionality property (4) holds.*

□

We can put the compositionality property (4) in more abstract terms as "the functor $\mathsf{beh} : \mathbf{Coalg}_F \to \mathbb{C}/Z$ preserves a tensor," meaning that the diagram below left commutes. Here a tensor $\underline{\otimes}$ on the slice category $\mathbb{C}/Z$ is given as on the right, using the inner composition $\|$.

$$\begin{array}{ccc} \mathbf{Coalg}_F \times \mathbf{Coalg}_F & \xrightarrow{\mathsf{beh} \times \mathsf{beh}} & \mathbb{C}/Z \times \mathbb{C}/Z \\ \otimes\downarrow & & \downarrow\underline{\otimes} \\ \mathbf{Coalg}_F & \xrightarrow[\mathsf{beh}]{} & \mathbb{C}/Z \end{array} \qquad \begin{pmatrix} X & Y \\ \downarrow f & \downarrow g \\ Z & Z \end{pmatrix} \overset{\underline{\otimes}}{\longmapsto} \begin{array}{c} X \otimes Y \\ \downarrow f \otimes g \\ Z \otimes Z \\ \downarrow \| \\ Z \end{array} \tag{6}$$

The point of Theorem 2.1 is as follows. Those parallel composition operators which are induced by sync are well-behaved ones: good properties like compositionality come for free. We shall present some examples in Section 2.2.

Remark 2.2. The view of parallel composition of systems as a tensor structure on $\mathbf{Coalg}_F$ has been previously presented in [13]. The interest there is on categorical

[2] Note that we use boldface $\otimes$ for a tensor on $\mathbf{Coalg}_F$ to distinguish it from $\otimes$ on $\mathbb{C}$.

structures on $\mathbf{Coalg}_F$ rather than on properties of parallel composition such as compositionality. In [13] and other literature an endofunctor F with sync (equipped with some additional compatibility) is called a *monoidal endofunctor*.[3]

2.2 Examples

In Sets: Bisimilarity is a congruence. We shall focus on LTSs and bisimilarity as their process semantics. For this purpose it is appropriate to take $\mathbf{Sets}$ as our base category $\mathbb{C}$ and $\mathcal{P}_\omega(\Sigma \times \_)$ as the functor F. We use Cartesian products as a tensor on $\mathbf{Sets}$. This means that a composition of two coalgebras has the product of the two state spaces as its state space, which matches our intuition. The functor $\mathcal{P}_\omega$ in F is the finite powerset functor; the finiteness assumption is needed for existence of a final F-coalgebra. It is standard (see e.g. [26]) that a final F-coalgebra captures bisimilarity via coinduction.

In considering parallel composition of LTSs, the following two examples are well-known ones.[4]

- *CSP-style* [7]: $a.P \parallel a.Q \xrightarrow{a} P \parallel Q$. For the whole system to make an a-action, each component has to make an a-action.
- *CCS-style* [21]: $a.P \parallel \overline{a}.Q \xrightarrow{\tau} P \parallel Q$, assuming $\Sigma = \{a, b, \dots\} \cup \{\overline{a}, \overline{b}, \dots\} \cup \{\tau\}$. When one component outputs on a channel a and another inputs from a, then the whole system makes an internal τ move.

In fact, each of these different ways of synchronization can be represented by a suitable sync natural transformation.

$$\begin{array}{ccc} \mathcal{P}_\omega(\Sigma \times X) \times \mathcal{P}_\omega(\Sigma \times Y) & \xrightarrow{\quad} & \mathcal{P}_\omega\big(\Sigma \times (X \times Y)\big) \\ (u, v) & \overset{\mathsf{sync}^{\mathrm{CSP}}_{X,Y}}{\longmapsto} & \big\{\, (a, (x, y)) \mid (a, x) \in u \wedge (a, y) \in v \,\big\} \\ (u, v) & \overset{\mathsf{sync}^{\mathrm{CCS}}_{X,Y}}{\longmapsto} & \big\{\, (\tau, (x, y)) \mid (a, x) \in u \wedge (\overline{a}, y) \in v \,\big\} \end{array}$$

By Theorem 2.1, each of these gives (different) $\otimes$ on $\mathbf{Coalg}_F$, and $\parallel$ on Z; moreover the behavior functor beh satisfies compositionality. In other words: bisimilarity is a congruence with respect to both CSP-style and CCS-style parallel composition.

Remark 2.3. As mentioned in the introduction, in some ways this paper can be seen as an extension of the bialgebraic studies started in [27]. However there is also a drawback, namely the limited expressive power of sync : $FX \otimes FY \to F(X \otimes Y)$.

Our sync specifies the way an algebraic structure interacts with a coalgebraic one. In this sense it is a counterpart of a distributive law $\Sigma F \Rightarrow F\Sigma$ in [27] representing operational rules, where Σ is a functor induced by an algebraic signature. However there are many common operational rules which do not allow representation of the

[3] Later in Section 3 we will observe that a functor F with sync is a special case of a *lax* $\mathbb{L}$-functor, by choosing a suitable algebraic theory $\mathbb{L}$. Such a functor F with sync is usually called a monoidal functor (as opposed to a *lax* monoidal functor), probably because it preserves (inner) monoid objects; see Proposition 3.8.1.

[4] Here we focus on synchronous interaction. Both CSP and CCS have an additional kind of interaction, namely an "interleaving" one; see Remark 2.3.

form $\Sigma F \Rightarrow F\Sigma$; therefore in [27] the type of such a distributive law is eventually extended to $\Sigma(F \times \mathrm{id}) \Rightarrow F\Sigma^*$. The class of rules representable in this form coincides with the class of so-called *GSOS-rules*.

At present it is not clear how we can make a similar extension for our sync; consequently there are some operational rules which we cannot model by sync. One important example is an *interleaving* kind of interaction—such as $a.P \parallel Q \xrightarrow{a} P \parallel Q$ which leaves the second component unchanged. This is taken care of in [27] by the identity functor (id) appearing on the left-hand side of $\Sigma(F \times \mathrm{id}) \Rightarrow F\Sigma^*$. For our sync to be able to model such interleaving, we can replace F by the cofree comonad on it, as is done in [13, Example 3.11]. This extension should be straightforward but detailed treatment is left as future work.

In $\mathcal{K}\ell(T)$: Trace equivalence is a congruence. In our recent work [6] we extend earlier observations in [25, 10] and show that trace semantics—including trace *set* semantics for non-deterministic systems and trace *distribution* semantics for probabilistic systems—is also captured by coinduction when it is employed in a Kleisli category $\mathcal{K}\ell(T)$. Applying the present composition framework, we can conclude that trace semantics is compositional with respect to well-behaved parallel composition. The details are omitted here due to lack of space.

2.3 Equational Properties of Parallel Composition

Now we shall investigate equational properties—associativity, commutativity, and so on—of parallel composition $\otimes$, which we have ignored deliberately for simplicity of argument. We present our result in terms of associativity; it is straightforward to transfer the result to other properties like commutativity. The main point of the following theorem is as follows: if $\otimes$ is associative and sync is "associative," then the lifting $\bar{\otimes}$ is associative. The proof is straightforward.

Theorem 2.4. *Let $\mathbb{C}$ be a category with a strictly associative tensor $\otimes$,*[5] *and $F : \mathbb{C} \to \mathbb{C}$ be a functor with* $\mathsf{sync} : FX \otimes FY \to F(X \otimes Y)$. *If the diagram*

$$\begin{array}{ccccc} FX \otimes (FY \otimes FZ) & \xrightarrow{FX \otimes \mathsf{sync}} & FX \otimes F(Y \otimes Z) & \xrightarrow{\mathsf{sync}} & F(X \otimes (Y \otimes Z)) \\ \downarrow \mathrm{id} & & & & \downarrow \mathrm{id} \\ (FX \otimes FY) \otimes FZ & \xrightarrow[\mathsf{sync} \otimes FZ]{} & F(X \otimes Y) \otimes FZ & \xrightarrow[\mathsf{sync}]{} & F((X \otimes Y) \otimes Z) \end{array} \tag{7}$$

commutes, then the lifted tensor $\bar{\otimes}$ on $\mathbf{Coalg}_F$ is strictly associative. □

The two identity arrows in (7) are available due to strict associativity of $\otimes$. In the next section we shall reveal the generic principle behind the commutativity condition of (7), namely a coherence condition on a lax natural transformation.

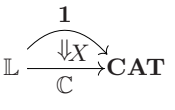
As an example, $\mathsf{sync}^{\mathrm{CSP}}$ and $\mathsf{sync}^{\mathrm{CCS}}$ in Section 2.2 are easily seen to be "associative" in the sense of the diagram (7). Therefore the resulting tensors $\bar{\otimes}$ are strictly associative.

[5] As mentioned already, in this paper we stick to *strict* algebraic structures.

3 Formalizing the Microcosm Principle

In this section we shall formalize the microcosm principle for an arbitrary algebraic theory presented as a Lawvere theory $\mathbb{L}$. This and the subsequent results generalize the results in the previous section. In particular, we will obtain a general compositionality result which works for an arbitrary algebraic theory.

As we sketched in the introduction, an outer model will be a product-preserving functor $\mathbb{C} : \mathbb{L} \to \mathbf{CAT}$; an inner model inside will be a lax natural transformation $X : \mathbf{1} \Rightarrow \mathbb{C}$. Here $\mathbf{1} : \mathbb{L} \to \mathbf{CAT}$ is the constant functor which maps everything to the category $\mathbf{1}$ with one object and one arrow (which is a special case of an outer model). Mediating 2-cells for the *lax* natural transformation X play a crucial role as inner interpretation of algebraic operations. In this section we heavily rely on 2-categorical notions, about which detailed accounts can be found in [4].



3.1 Lawvere Theories

Lawvere theories are categorical presentations of algebraic theories. The notion is introduced in [18] (not under this name, though) aiming at a categorical formulation of "theories" and "semantics." An accessible introduction to the notion can be found in [17]. Lawvere theories are known to be equivalent to *finitary monads*. These two ways of presenting algebraic theories have been widely used in theoretical computer science, e.g. for modeling computation with effect [22, 8]. Recent developments (such as [24]) utilize the increased expressive power of *enriched* Lawvere theories.

In the sequel, by an *FP-category* we refer to a category with (a choice of) finite products. An *FP-functor* is a functor between FP-categories which preserves finite products "on-the-nose," that is, up-to-equality instead of up-to-isomorphism.

Definition 3.1 (Lawvere theory). By $\mathbf{Nat}$ we denote the category of natural numbers (as sets) and functions between them. Therefore every arrow in $\mathbf{Nat}$ is a (cotuple of) coprojection; an arrow in $\mathbf{Nat}^{\mathrm{op}}$ is a (tuple of) projection.[6]

A *Lawvere theory* is a small FP-category $\mathbb{L}$ equipped with an FP-functor $H : \mathbf{Nat}^{\mathrm{op}} \to \mathbb{L}$ which is bijective on objects. We shall denote an object of $\mathbb{L}$ by a natural number k, identifying $k \in \mathbf{Nat}^{\mathrm{op}}$ and $Hk \in \mathbb{L}$.

The category $\mathbf{Nat}^{\mathrm{op}}$—which is a free FP-category on the trivial category $\mathbf{1}$—is there in order to specify the choice of finite products in $\mathbb{L}$. For illustration, we make some remarks on $\mathbb{L}$'s objects and arrows.

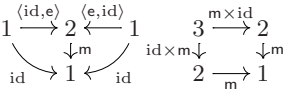
- An object $k \in \mathbb{L}$ is a k-fold product $1 \times \cdots \times 1$ of 1.
- An arrow in $\mathbb{L}$ is intuitively understood as an algebraic operation. That is, $k \to 1$ as a k-ary operation; and $k \to n$ as an n-tuple $\langle \mathsf{f}_1, \ldots, \mathsf{f}_n \rangle$ of k-ary operations. To be precise, arrows in $\mathbb{L}$ also include projections (such as $\pi_1 : 2 \to 1$) and *terms* made up of operations and projections (such as $\mathsf{m} \circ \langle \pi_1, \pi_2 \rangle : 3 \to 1$).

[6] An arrow $f : n \to k$ in $\mathbf{Nat}$ can be written as a cotuple $[\kappa_{f(1)}, \ldots, \kappa_{f(n)}]$ where $\kappa_i : 1 \to k$ is the coprojection into the i-th summand of $1 + \cdots + 1$ (k times).

Conventionally in universal algebra, an algebraic theory is presented by an *algebraic specification* (Σ, E)—a pair of a set Σ of operations and a set E of equations. A Lawvere theory $\mathbb{L}$ arises from such (Σ, E) as its so-called *classifying category* (see e.g. [18, 9]). An arrow $k \to n$ in the resulting Lawvere theory $\mathbb{L}$ is an n-tuple $([t_1(\overrightarrow{x})], \dots, [t_n(\overrightarrow{x})])$ of Σ-terms with k variables $\overrightarrow{x}$, where $[\_]$ denotes taking an equivalence class modulo equations in E. An equivalent way to describe this construction is via *sketches*: (Σ, E) is identified with an FP-sketch, which in turn induces $\mathbb{L}$ as a free FP-category. See [2] for details.

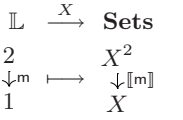
Our leading example is the Lawvere theory $\mathbf{Mon}$ for monoids.[7] It arises as a classifying category from the well-known algebraic specification of monoids. This specification has a nullary operation e and a binary one m; subject to the equations $\mathsf{m}(x, \mathsf{e}) = x$, $\mathsf{m}(\mathsf{e}, x) = x$, and $\mathsf{m}(x, \mathsf{m}(y, z)) = \mathsf{m}(\mathsf{m}(x, y), z)$.

Equivalently, $\mathbf{Mon}$ is the freely generated FP-category by arrows $0 \xrightarrow{\mathsf{e}} 1$ and $2 \xrightarrow{\mathsf{m}} 1$ subject to the commutativity on the right. These data (arrows and commutative diagrams) form an FP-sketch (see [2]).



3.2 Outer Models: $\mathbb{L}$-Categories

We start by formalizing an outer model. It is a category with an $\mathbb{L}$-structure, hence called an $\mathbb{L}$-*category*. It is standard that a (set-theoretic) model of $\mathbb{L}$—a *set* with an $\mathbb{L}$-structure—is identified with an FP-functor $\mathbb{L} \xrightarrow{X} \mathbf{Sets}$. Concretely, let $X = X1$ be the image of $1 \in \mathbb{L}$. Then $k \in \mathbb{L}$ must be sent to X^k due to preservation of finite products. Now the functor's action on arrows is what interprets $\mathbb{L}$'s operations in X, as illustrated above right. Equations (expressed as commutative diagrams in $\mathbb{L}$) are satisfied because a functor preserves commutativity.



Turning back to $\mathbb{L}$-*categories*, what we have to do here is to just replace $\mathbf{Sets}$ by the category $\mathbf{CAT}$ of (possibly large and locally small) categories.

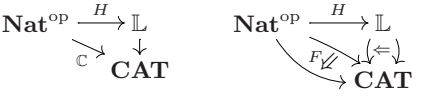
Definition 3.2 ($\mathbb{L}$-categories, $\mathbb{L}$-functors). A (strict) $\mathbb{L}$-*category* is an FP-functor $\mathbb{L} \xrightarrow{\mathbb{C}} \mathbf{CAT}$. In the sequel we denote the image $\mathbb{C}1$ of $1 \in \mathbb{L}$ by $\mathbb{C}$; and the image $\mathbb{C}(\mathsf{f})$ of an arrow f by $[\![\mathsf{f}]\!]$.

An $\mathbb{L}$-*functor* $F : \mathbb{C} \to \mathbb{D}$—a functor preserving an $\mathbb{L}$-structure—is a natural transformation $\mathbb{L} \overset{\mathbb{C}}{\underset{\mathbb{D}}{\Downarrow F}} \mathbf{CAT}$.

Another way to look at the previous definition is to view an $\mathbb{L}$-structure as "factorization through $\mathbf{Nat}^{\mathrm{op}} \to \mathbb{L}$." We can identify a category $\mathbb{C} \in \mathbf{CAT}$ with a functor $\mathbf{1} \to \mathbf{CAT}$, which is in turn identified with an FP-functor $\mathbf{Nat}^{\mathrm{op}} \to \mathbf{CAT}$, because $\mathbf{Nat}^{\mathrm{op}}$ is the free FP-category on $\mathbf{1}$. We say that $\mathbb{C}$ has an $\mathbb{L}$-structure, if this FP-functor factors through $H : \mathbf{Nat}^{\mathrm{op}} \to \mathbb{L}$ (as below left). Note that the factorization is not necessarily

[7] The Lawvere theory $\mathbf{Mon}$ for the theory of monoids should not be confused with the category of (set-theoretic) monoids and monoid homomorphisms (which is often denoted by $\mathbf{Mon}$ as well).

unique, because there can be different ways of interpreting the algebraic theory $\mathbb{L}$ in $\mathbb{C}$. Similarly, a functor $\mathbb{C} \xrightarrow{F} \mathbb{D}$ is identified with a natural transformation $\mathbf{1} \overset{\Downarrow F}{\rightrightarrows} \mathbf{CAT}$; and then with $\mathbf{Nat}^{\mathrm{op}} \overset{\Downarrow F}{\rightrightarrows} \mathbf{CAT}$ due to the 2-universality of $\mathbf{Nat}^{\mathrm{op}}$ as a free object. We say that this F preserves an $\mathbb{L}$-structure, if the last natural transformation factors through $H : \mathbf{Nat}^{\mathrm{op}} \to \mathbb{L}$ (as below right).

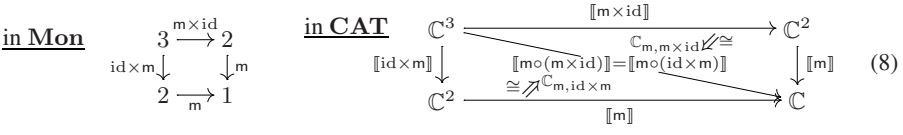


Example 3.3. The usual notion of strictly monoidal categories coincides with $\mathbb{L}$-categories for $\mathbb{L} = \mathbf{Mon}$. A tensor $\otimes$ and a unit I on a category arise as interpretation of the operations $2 \xrightarrow{\mathsf{m}} 1$ and $0 \xrightarrow{\mathsf{e}} 1$; commuting diagrams in $\mathbf{Mon}$ such as $\mathsf{m} \circ \langle \mathrm{id}, \mathsf{e} \rangle = \mathrm{id}$ yield equational properties of $\otimes$ and I.

3.3 Remarks on "Pseudo" Algebraic Structures

As we mentioned in the introduction, in this paper we focus on *strict* algebraic structures. This means that monoidal categories (in which associativity holds only up-to-isomorphism, for example) fall out of our consideration. Extending our current framework to such "pseudo" algebraic structures is one important direction of our future work. Such an extension is not entirely obvious; we shall sketch some preliminary observations in this direction.

The starting point is to relax the definition of $\mathbb{L}$-categories from (strict) functors $\mathbb{L} \to \mathbf{CAT}$ to *pseudo* functors, meaning that composition and identities are preserved only up-to-isomorphism. Then it is not hard to see that a pseudo functor $\mathbf{Mon} \xrightarrow{\mathbb{C}} \mathbf{CAT}$ (which preserves finite products in a suitable sense) gives rise to a monoidal category. Indeed, let us denote a mediating iso-2-cell for composition by $\mathbb{C}_{\mathsf{g},\mathsf{f}} : [\![\mathsf{g}]\!] \circ [\![\mathsf{f}]\!] \overset{\cong}{\Rightarrow} [\![\mathsf{g} \circ \mathsf{f}]\!]$. The associativity diagram (below left) gives rise to the two iso-2-cells on the right.



The composition $\mathbb{C}^{-1}_{\mathsf{m},\mathrm{id}\times\mathsf{m}} \bullet \mathbb{C}_{\mathsf{m},\mathsf{m}\times\mathrm{id}}$ is what gives us a natural isomorphism $\alpha : X \otimes (Y \otimes Z) \overset{\cong}{\Rightarrow} (X \otimes Y) \otimes Z$. Moreover, the coherence condition on such isomorphisms in a monoidal category (like the famous pentagon diagram; see [20]) follows from the coherence condition on mediating 2-cells of a pseudo functor (see [4]).

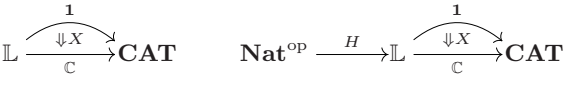
So far so good. However, at this moment it is not clear what is a canonical construction the other way round, i.e. from a monoidal category to a pseudo functor.[8] In the present paper we side-step these 2-categorical subtleties by restricting ourselves to strict, non-pseudo functors.

[8] For example, given a monoidal category $\mathbb{C}$, we need to define a functor $[\![\mathsf{m} \circ (\mathsf{m} \times \mathrm{id})]\!] = [\![\mathsf{m} \circ (\mathrm{id} \times \mathsf{m})]\!]$ in (8). It's not clear whether it should carry (X, Y, Z) to $X \otimes (Y \otimes Z)$, or to $(X \otimes Y) \otimes Z$.

3.4 Inner Models: $\mathbb{L}$-Objects

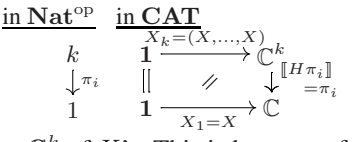
We proceed to formalize an inner model. It is an object in an $\mathbb{L}$-category which itself carries an (inner) $\mathbb{L}$-structure, hence is called an $\mathbb{L}$-*object*. A monoid object in a monoidal category is a prototypical example. We first present an abstract definition; some illustration follows afterwards.

Definition 3.4 ($\mathbb{L}$-objects). An $\mathbb{L}$-*object* X in an $\mathbb{L}$-category $\mathbb{C}$ is a lax natural transformation $X : \mathbf{1} \Rightarrow \mathbb{C}$ (below left) which is "product-preserving": this means that the composition $X \circ H$ (below right) is strictly, non-lax natural. Here $\mathbf{1} : \mathbb{L} \to \mathbf{CAT}$ denotes the constant functor to the trivial one-object category $\mathbf{1}$.

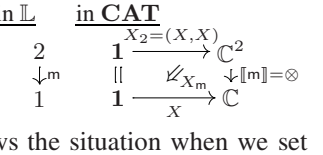


Such a nested algebraic structure—formalized as an $\mathbb{L}$-object in an $\mathbb{L}$-category—shall be called a *microcosm model* for $\mathbb{L}$.

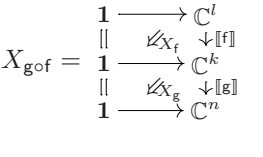
Let us now illustrate the definition. First, X's component at $1 \in \mathbb{L}$ is a functor $\mathbf{1} \xrightarrow{X_1} \mathbb{C}$ which is identified with an object $X \in \mathbb{C}$. This is the "carrier" object of this inner algebra. Moreover, any other component $\mathbf{1} \xrightarrow{X_k} \mathbb{C}^k$ must be the k-tuple $(X, \ldots, X) \in \mathbb{C}^k$ of X's. This is because of (strict) naturality of $X \circ H$ (see above right): for any $i \in [1, k]$ the composite $\pi_i \circ X_k$ is required to be X_1.



The (inner) algebraic structure on X arises in the form of mediating 2-cells of the *lax* natural transformation. For each arrow $k \xrightarrow{\mathsf{f}} n$ in $\mathbb{L}$, lax naturality of X requires existence of a mediating 2-cell $X_{\mathsf{f}} : [\![\mathsf{f}]\!] \circ X_k \Rightarrow X_n$. The diagram (above right) shows the situation when we set $\mathsf{f} = \mathsf{m}$, a binary operation. The natural transformation X_{m} can be identified with an arrow $X \otimes X \xrightarrow{\mu} X$ in $\mathbb{C}$, which gives an inner binary operation on X.



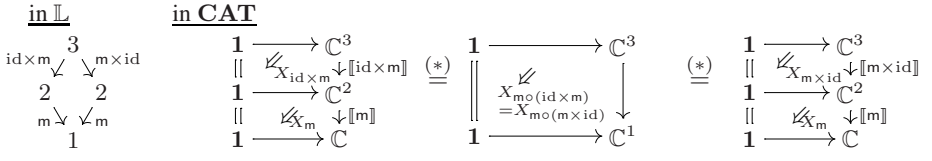
How do such inner operations on X satisfy equations as specified in $\mathbb{L}$? The key is the coherence condition[9] on mediating 2-cells: it requires $X_{\mathrm{id}} = \mathrm{id}$ concerning identities; and $X_{\mathsf{g}\circ\mathsf{f}} = X_{\mathsf{g}} \bullet ([\![\mathsf{g}]\!] \circ X_{\mathsf{f}})$ concerning composition (as on the right). The following example illustrates how such coherence induces equational properties.



Example 3.5. A monoid object in a strictly monoidal category is an example of an $\mathbb{L}$-object in an $\mathbb{L}$-category. Here we take $\mathbb{L} = \mathbf{Mon}$, the theory of monoids.

[9] This is part of the notion of lax natural transformations; see [4].

For illustration, let us here derive associativity of multiplication $X \otimes X \xrightarrow{\mu} X$. In the current setting the multiplication μ is identified with a mediating 2-cell X_{m} as above. The coherence condition yields the two equalities $(*)$ below.



Now it is not hard to see that: the composed 2-cell on the left corresponds to $X^3 \xrightarrow{X\times\mu} X^2 \xrightarrow{\mu} X$; and the one on the right corresponds to $X^3 \xrightarrow{\mu\times X} X^2 \xrightarrow{\mu} X$. The equalities $(*)$ above prove that these two arrows $X^3 \rightrightarrows X$ are identical.

3.5 Microcosm Structures in Coalgebras

In this section we return to our original question and apply the framework we just introduced to coalgebraic settings. First we present some basic results, which are used later in our main result of general compositionality. The constructs in Section 2 (such as sync) will appear again, now in their generalized form. Some details and proofs are omitted here due to lack of space. They will appear in the forthcoming extended version of this paper, although the diligent reader will readily work them out.

Let $\mathbb{C}$ be an $\mathbb{L}$-category, and $F : \mathbb{C} \to \mathbb{C}$ be a functor. We can imagine that, for the category $\mathbf{Coalg}_F$ to carry an $\mathbb{L}$-structure, F needs to be somehow compatible with $\mathbb{L}$; it turns out that the following condition is sufficient. It is weaker than F's being an $\mathbb{L}$-functor (see Definition 3.2).

Definition 3.6 (Lax $\mathbb{L}$-functor). A functor $F : \mathbb{C} \to \mathbb{D}$ between $\mathbb{L}$-categories is said to be a *lax* $\mathbb{L}$*-functor* if it is identified with[10] some lax natural transformation $\mathbb{L} \xrightarrow[\mathbb{D}]{\mathbb{C}} \mathbf{CAT}$ which is product-preserving (i.e. $F \circ H$ is strictly natural; see Definition 3.4).

Lax $\mathbb{L}$-endofunctors are natural generalization of functors with sync as in Section 2. To illustrate this, look at the lax naturality diagram on the right for a binary operation m. Here we denote the outer interpretation $[\![\mathsf{m}]\!]$ by $\otimes$. The 2-component is $F_2 = (F, F)$ because the lax natural transformation F is product-preserving. The mediating 2-cell F_{m} can be identified with a natural transformation $FX \otimes FY \to F(X \otimes Y)$; this is what we previously called sync. Moreover, F_{m} (as generalized sync) is automatically compatible with equational properties (as in Theorem 2.4); this is because of the coherence condition on mediating 2-cells like "$F_{\mathsf{g}\circ\mathsf{f}}$ is a suitable composition of F_{g} after F_{f}."

The following results follow from a more general result concerning the notion of *inserters*, namely: when G is an oplax $\mathbb{L}$-functor and F is a lax $\mathbb{L}$-functor, then the inserter $Ins(G, F)$ is an $\mathbb{L}$-category.

[10] Meaning: $F : \mathbb{C} \to \mathbb{D}$ is the 1-component of such a lax natural transformation $\mathbb{C} \Rightarrow \mathbb{D}$.

Proposition 3.7. *1. Let $\mathbb{C}$ be an $\mathbb{L}$-category and $F : \mathbb{C} \to \mathbb{C}$ be a lax $\mathbb{L}$-functor. Then $\mathbf{Coalg}_F$ is an $\mathbb{L}$-category; moreover the forgetful functor $\mathbf{Coalg}_F \xrightarrow{U} \mathbb{C}$ is a (strict, non-lax) $\mathbb{L}$-functor.*

2. Given a microcosm model $X \in \mathbb{C}$ for $\mathbb{L}$, the slice category $\mathbb{C}/X$ is an $\mathbb{L}$-category; moreover the functor $\mathbb{C}/X \xrightarrow{\mathrm{dom}} \mathbb{C}$ is an $\mathbb{L}$-functor. □

Note that $\mathbf{Coalg}_F$ being an $\mathbb{L}$-category means not only that operations are interpreted in $\mathbf{Coalg}_F$ but also that all the equational properties specified in $\mathbb{L}$ are satisfied in $\mathbf{Coalg}_F$. Therefore this result generalizes Theorem 2.4.

Concretely, an operation $\mathsf{f} : k \to 1$ in $\mathbb{L}$ is interpreted in $\mathbf{Coalg}_F$ and $\mathbb{C}/X$ as follows, respectively.

$$\begin{pmatrix} FX_1 & & FX_k \\ \uparrow c_1 & \cdots, & \uparrow c_k \\ X_1 & & X_k \end{pmatrix} \mapsto \begin{array}{c} F[\![\mathsf{f}]\!](\overrightarrow{X}) \\ \uparrow(F_{\mathsf{f}})_{\overrightarrow{X}} \\ [\![\mathsf{f}]\!](\overrightarrow{FX}) \\ \uparrow[\![\mathsf{f}]\!](\overrightarrow{c}) \\ [\![\mathsf{f}]\!](\overrightarrow{X}) \end{array} \qquad \begin{pmatrix} Y_1 & & Y_k \\ \downarrow y_1 & \cdots, & \downarrow y_k \\ X & & X \end{pmatrix} \mapsto \begin{array}{c} [\![\mathsf{f}]\!](\overrightarrow{Y}) \\ \downarrow[\![\mathsf{f}]\!](\overrightarrow{y}) \\ [\![\mathsf{f}]\!](\overrightarrow{X}) \\ \downarrow X_{\mathsf{f}} \\ X \end{array}$$

Compare these with (5) and (6); these make an essential use of F_{f} and X_{f} which generalize sync and $\|$ in Section 2, respectively.

Proposition 3.8. *1. A lax $\mathbb{L}$-functor preserves $\mathbb{L}$-objects. Hence so does an $\mathbb{L}$-functor.*

2. A final object of an $\mathbb{L}$-category $\mathbb{C}$, if it exists, is an $\mathbb{L}$-object. The inner $\mathbb{L}$-structure is induced by finality. □

We can now present our main result. It generalizes Theorem 2.1, hence is a generalized version of the "coalgebraic compositionality" equation (4).

Theorem 3.9 (General compositionality). *Let $\mathbb{C}$ be an $\mathbb{L}$-category and $F : \mathbb{C} \to \mathbb{C}$ be a lax $\mathbb{L}$-functor. Assume further that $\zeta : Z \xrightarrow{\cong} FZ$ is the final coalgebra. Then the functor* $\mathsf{beh} : \mathbf{Coalg}_F \to \mathbb{C}/Z$ *is a (non-lax) $\mathbb{L}$-functor. It makes the following diagram of $\mathbb{L}$-functors commute.*

$$\begin{array}{ccc} \mathbf{Coalg}_F & \xrightarrow{\mathsf{beh}} & \mathbb{C}/Z \\ & {\scriptstyle U}\searrow \quad \mathbb{C} \quad \swarrow{\scriptstyle \mathrm{dom}} & \end{array}$$

□

The proof is straightforward by finality. Here $\mathbf{Coalg}_F$ is an $\mathbb{L}$-category (Proposition 3.7.1). So is $\mathbb{C}/Z$ because: $\zeta \in \mathbf{Coalg}_F$ is an $\mathbb{L}$-object (Proposition 3.8.2); $Z = U\zeta$ is an $\mathbb{L}$-object (Propositions 3.8.1 and 3.7.1); hence $\mathbb{C}/Z$ is an $\mathbb{L}$-category (Proposition 3.7.2).

We have also observed some facts which look interesting but are not directly needed for our main result (Theorem 3.9). They include: the category $\mathbb{L}\text{-}\mathbf{obj}_{\mathbb{C}}$ of $\mathbb{L}$-objects in $\mathbb{C}$ and morphisms between them forms the lax limit of a diagram $\mathbb{C} : \mathbb{L} \to \mathbf{CAT}$; the simplicial category Δ is the "universal" microcosm model for $\mathbf{Mon}$ (cf. [20, Proposition VII.5.1]). The details will appear in the forthcoming extended version.

4 Conclusions and Future Work

In this paper we have observed that the microcosm principle (as called by Baez and Dolan) brings new mathematical insights into computer science. Specifically, we have looked into parallel composition of coalgebras, which would serve as a mathematical basis for the study of concurrency. As a purely mathematical expedition, we have presented a 2-categorical formalization of the microcosm principle, where an algebraic theory is presented by a Lawvere theory. Turning back to our original motivation, the formalization was applied to coalgebras and yielded some general results which ensure compositionality and equational properties such as associativity.

There are many questions yet to be answered. Some of them have been already mentioned, namely: extending the expressive power of sync (Remark 2.3), and a proper treatment of "pseudo" algebraic structures (Section 3.3).

On the application side, one direction of future work is to establish a relationship between sync and *(syntactic) formats* for process algebras. Our sync represents a certain class of operational rules; formats are a more syntactic way to do the same. Formats which guarantee certain good properties (such as commutativity, see [23]) have been actively studied. Such a format should be obtained by translating e.g. a "commutative" sync into a format.

On the mathematical side, one direction is to identify more instances of the microcosm principle. Mathematics abounds with the (often implicit) idea of nested algebraic structures. To name a few: a topological space in a topos which is itself a "generalized topological space"; a category of domains which itself carries a "structure as a domain." We wish to turn such an informal statement into a mathematically rigorous one, by generalizing the current formalization of the microcosm principle. As a possible first step towards this direction, we are working on formalizing the microcosm principle for finitary monads which are known to be roughly the same thing as Lawvere theories.

Another direction is a search for n-folded nested algebraic structures. In the current paper we have concentrated on two levels of interpretation; an example with more levels might be found e.g. in an internal category in an internal category.

Acknowledgments. Thanks are due to Kazuyuki Asada, John Baez, Masahito Hasegawa, Bill Lawvere, Duško Pavlović, John Power and the participants of CALCO-jnr workshop 2007 including Alexander Kurz for helpful discussions and comments.

References

1. Baez, J.C., Dolan, J.: Higher dimensional algebra III: n-categories and the algebra of opetopes. Adv. Math. 135, 145–206 (1998)
2. Barr, M., Wells, C.: Toposes, Triples and Theories. Springer, Berlin (1985)
3. Bartels, F.: On generalised coinduction and probabilistic specification formats. Distributive laws in coalgebraic modelling. PhD thesis, Free Univ. Amsterdam (2004)
4. Borceux, F.: Handbook of Categorical Algebra. Encyclopedia of Mathematics, vol. 50, 51, 52. Cambridge Univ. Press, Cambridge (1994)
5. Hasuo, I.: Generic forward and backward simulations. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 406–420. Springer, Heidelberg (2006)

6. Hasuo, I., Jacobs, B., Sokolova, A.: Generic trace semantics via coinduction. Logical Methods in Comp. Sci. 3(4–11) (2007)
7. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall, Englewood Cliffs (1985)
8. Hyland, M., Power, A.J.: Discrete Lawvere theories and computational effects. Theor. Comp. Sci. 366(1–2), 144–162 (2006)
9. Jacobs, B.: Categorical Logic and Type Theory. North Holland, Amsterdam (1999)
10. Jacobs, B.: Trace semantics for coalgebras. In: Adámek, J., Milius, S. (eds.) Coalgebraic Methods in Computer Science. Elect. Notes in Theor. Comp. Sci., vol. 106, Elsevier, Amsterdam (2004)
11. Jacobs, B.: Introduction to coalgebra. Towards mathematics of states and observations (2005) Draft of a book, `www.cs.ru.nl/B.Jacobs/PAPERS/index.html`
12. Jacobs, B.: A bialgebraic review of deterministic automata, regular expressions and languages. In: Futatsugi, K., Jouannaud, J.-P., Meseguer, J. (eds.) Algebra, Meaning, and Computation. LNCS, vol. 4060, pp. 375–404. Springer, Heidelberg (2006)
13. Johnstone, P.T., Power, A.J., Tsujishita, T., Watanabe, H., Worrell, J.: An axiomatics for categories of transition systems as coalgebras. In: Logic in Computer Science, IEEE, Computer Science Press, Los Alamitos (1998)
14. Kick, M., Power, A.J., Simpson, A.: Coalgebraic semantics for timed processes. Inf. & Comp. 204(4), 588–609 (2006)
15. Klin, B.: From bialgebraic semantics to congruence formats. In: Workshop on Structural Operational Semantics (SOS 2004). Elect. Notes in Theor. Comp. Sci. 128, 3–37 (2005)
16. Klin, B.: Bialgebraic operational semantics and modal logic. In: Logic in Computer Science, pp. 336–345. IEEE Computer Society, Los Alamitos (2007)
17. Kock, A., Reyes, G.E.: Doctrines in categorical logic. In: Barwise, J. (ed.) Handbook of Mathematical Logic, pp. 283–313. North-Holland, Amsterdam (1977)
18. Lawvere, F.W.: Functorial Semantics of Algebraic Theories and Some Algebraic Problems in the Context of Functorial Semantics of Algebraic Theories. PhD thesis, Columbia University, 1963. Reprints in Theory and Applications of Categories 5, 1–121 (2004)
19. Lee, E.A.: Making concurrency mainstream, Invited talk at CONCUR 2006 (2006)
20. Mac Lane, S.: Categories for the Working Mathematician, 2nd edn. Springer, Berlin (1998)
21. Milner, R.: Communication and Concurrency. Prentice-Hall, Englewood Cliffs (1989)
22. Moggi, E.: Notions of computation and monads. Inf. & Comp. 93(1), 55–92 (1991)
23. Mousavi, M.R., Reniers, M.A., Groote, J.F.: A syntactic commutativity format for SOS. Inform. Process. Lett. 93(5), 217–223 (2005)
24. Nishizawa, K., Power, A.J.: Lawvere theories enriched over a general base. Journ. of Pure & Appl. Algebra (to appear, 2006)
25. Power, J., Turi, D.: A coalgebraic foundation for linear time semantics. In: Category Theory and Computer Science. Elect. Notes in Theor. Comp. Sci., vol. 29, Elsevier, Amsterdam (1999)
26. Rutten, J.J.M.M.: Universal coalgebra: a theory of systems. Theor. Comp. Sci. 249, 3–80 (2000)
27. Turi, D., Plotkin, G.: Towards a mathematical operational semantics. In: Logic in Computer Science, pp. 280–291. IEEE, Computer Science Press, Los Alamitos (1997)

2009 30th IEEE Real-Time Systems Symposium

Distributed, Modular HTL*

Thomas A. Henzinger
EPFL and IST Austria
tah@ist.ac.at

Christoph M. Kirsch
University of Salzburg
ck@cs.uni-salzburg.at

Eduardo R. B. Marques
University of Porto
edrdo@dcc.fc.up.pt

Ana Sokolova
University of Salzburg
anas@cs.uni-salzburg.at

***Abstract*—The Hierarchical Timing Language (HTL) is a real-time coordination language for distributed control systems. HTL programs must be checked for well-formedness, race freedom, transmission safety (schedulability of inter-host communication), and time safety (schedulability of host computation). We present a modular abstract syntax and semantics for HTL, modular checks of well-formedness, race freedom, and transmission safety, and modular code distribution. Our contributions here complement previous results on HTL time safety and modular code generation. Modularity in HTL can be utilized in easy program composition as well as fast program analysis and code generation, but also in so-called runtime patching, where program components may be modified at runtime.**

I. Introduction

The Hierarchical Timing Language (HTL) [1] is a real-time coordination language for distributed control systems.

HTL essentially consists of four building blocks called program, module, mode, and task. An HTL program is a hierarchical, tree-like structure whose root node must be a program block. The immediate successors of a program block can be any number of module blocks, which are executed in parallel. The immediate successors of a module block can be any number of mode blocks, with one mode identified as start mode and some mode switching logic. During execution only one mode per module can be active at any time. The immediate successors of a mode block can be a (refinement) program block and any number of task blocks, which are executed periodically in parallel. Non-concurrent, concrete tasks (with implementations) in a refinement program refine a single abstract task (without implementation) which is to be understood as a placeholder for schedulability. Upon invocation, the implementation of a concrete task, which is written in some language other than HTL, such as C, for example, computes new output from its input. Task input may come from output of other tasks running either in the same mode, implicitly creating a task precedence relation, or in other modules, with possibly different periods, but only through so-called communicators, which are periodically updated program-wide variables with their own periods independent of any task periods. The execution of HTL programs on multiple hosts may be distributed at the level of modules with all inter-host communication done through communicators. Duplicates of the same module may execute on multiple hosts for fault tolerance [2].

HTL programs must be checked for well-formedness (of syntax), race freedom (of communicator updates), transmission safety (schedulability of inter-host communication), and time safety (schedulability of host computation). An optional check of reliability of communicator updates on unreliable hardware [2] may also be performed but is not considered here. The key feature of HTL is that the race-free, transmission-safe, and time-safe execution of well-formed programs is time-deterministic, that is, the computed values and update times of communicators are input-determined and therefore predictable [1].

In this paper, we present (1) a modular abstract syntax and semantics for HTL, (2) modular checks of well-formedness, race freedom, and transmission safety, and (3) modular code distribution. The key to (3) is the modular transmission safety check, ensuring that each communicator value can be communicated within a single communicator period. The original definition of HTL does not provide (1)–(3). Only time safety checking of refinement programs [1] and HTL code generation [3] has already been done modularly. The key result of refinement in HTL is that well-formed concrete programs that refine time-safe abstract programs are also time-safe [1]. Our contributions here complete the distributed and modular design of HTL, except for time safety checking of top-level programs (for which there are no abstractions), which remains non-modular.

Modularity in HTL is important for design scalability (program composition, analysis, and code generation) but also enables efficient program modifications at runtime, called runtime patching, while maintaining predictable behavior [4]. Runtime patching is a semantically well-defined method to modify program components at runtime, and may eventually be used as software foundation for addressing uncertainty in control systems.

*Supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design, the EU project COMBEST, the Austrian Science Funds P18913-N15 and V00125, and Fundação para a Ciência e Tecnologia funds SFRH/BD/29461/2006 and PTDC/EIA/71462/2006.

1052-8725/09 $26.00 © 2009 IEEE
DOI 10.1109/RTSS.2009.9

We first introduce HTL informally by example in Section II. We then discuss HTL compilation and runtime patching in Section III. The modular abstract syntax and semantics for HTL are defined in Section IV, followed by a detailed discussion of modularity in HTL in Section V. Some related work is pointed out in Section VI. The conclusion is in Section VII.

II. Overview of HTL

A. An HTL example application

Figure 1 illustrates a three-tank system (3TS) [5] consisting of three tanks, Tank_1, Tank_2, and Tank_3, with respective evacuation taps Tap_1, Tap_2, and Tap_3, and tank inter-connecting taps $\text{Tap}_{1,3}$ and $\text{Tap}_{2,3}$. Two pumps Pump_1 and Pump_2 actuate on the system by controlling the flow of water into Tank_1 and Tank_2. The aim is to maintain the water level in the tanks both in the normal case with no perturbations (no water leaves the tanks through the tanks' taps) and in the case of perturbations (water leaks). To control the pumps, a proportional (P) controller is used in the absence of perturbations, and two proportional-integrative (PI) controllers are used when there are perturbations, one with slow integration speed for an estimated low control error, the other with faster integration speed. All the controllers work with a frequency of 2Hz.

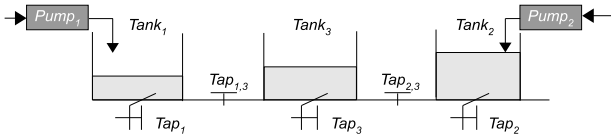


Fig. 1. Three-tank system [5]

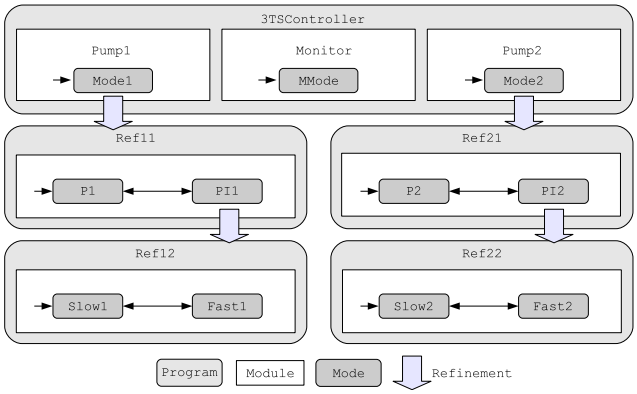


Fig. 2. HTL 3TS controller [3]

To control this system, an HTL program has been implemented [3] running on three hosts with its high-level structure shown in Figure 2. Two hosts have direct access to the two pumps, respectively. The remaining host serves as a monitoring interface to an operator. The HTL top-level program `3TSController` consists of three concurrently running modules `Pump1`, `Monitor`, and `Pump2`, each mapped to one of the mentioned hosts. Each module is organized in modes, which describe switchable configurations of operation in the module, with each mode defining the invocation of a set of real-time task invocations over a time period, some of which can be abstract placeholders for hierarchical refinement. In the example, the pump control modules `Pump1` and `Pump2` are symmetrical in structure. Each has a single mode that is further refined into a program with a single module, switching between modes of P-control and PI-control according to tap perturbations. The PI mode is further refined by a program that defines the "slow"-PI and "fast"-PI control modes.

Figure 3 illustrates a possible execution of `3TSController`. The control starts without water-level perturbations in the system, and modes `P1` and `P2` remain active for some time. Due to a perturbation in the water level of the first tank, there is a mode switch to mode `PI1` within `Ref11` at time 1, and, by refinement, program `Ref12` and mode `Slow1` within `Ref12` become active. Later, at time 2, due to a high control error, there is a mode switch from `Slow1` to `Fast1` within `Ref12`. A similar behavior of the controller of the second tank can be observed, with perturbations making `Ref21`/`PI2` become active at time 1.5, and `Slow2` switching to `Fast2` within `Ref22` at time 3. As perturbations in the first tank decrease, there is a mode switch from `Fast1` to `Slow1` at time 3.5, and at higher level, from mode `PI1` to `P1` at time 4.5.

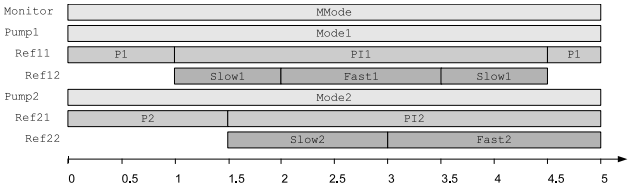


Fig. 3. Sample `3TSController` execution

B. HTL programs, components, and blocks

An HTL program is a tree-like structure of HTL blocks (program, module, mode, task), where the root node is a program block. A component of an HTL program is a subtree of the program. We now describe in more detail the four HTL blocks, and in addition, explain the hierarchical refinement relation, discuss the advantage of hierarchical programs over flat programs, and present the definition of a platform on which an HTL program executes.

Task. A task T in HTL, depicted in Figure 4, is defined by a sequential code procedure `f` without internal synchronization and with isolated memory space, and sets `In`, `Out`, and `Priv` of input, output, and private persistent variables called ports. Each port p is characterized by a set $\texttt{Type}(p)$, defining the domain of p, and an initial value $\texttt{init}(p) \in \texttt{Type}(p)$ that is undefined (equals $\perp$) for non-private ports. The code procedure `f` is implemented in an external language (e.g. C or Java), and its flow of execution comprises an assignment of the task's input ports (`In`) and the actual execution of the procedure code that, upon completion, produces values written to the task's output ports (`Out`) and updates the task's private ports (`Priv`). The private ports enable storing state across task invocations. This task scheme resembles port-based objects [6].

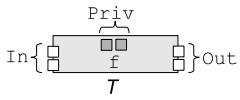


Fig. 4. An HTL task

The HTL task model is a generalization of the Giotto task model [7], based on the notion of logical execution time (LET) of tasks, which is defined by release and termination events. The release time is the latest of logical times for task inputs to become available and the termination time is the earliest of logical times when task outputs may be made available. The LET of a task defines a logical, platform-independent interval for execution of the task's code procedure. All task inputs and outputs can be understood as logically available at the release and termination time, respectively. In an actual execution on a concrete platform, the task procedure may start later than the release event, or complete before the termination event, and be preempted any number of times (if preemption is supported by the platform), as illustrated in Figure 5. The key property of the LET model is that the functional aspects and some key non-functional aspects of programs such as I/O and memory management are preserved across different platforms and workloads as long as there are sufficient computational resources [8], [7], [9].

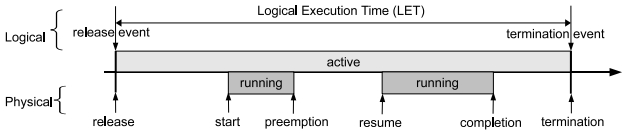


Fig. 5. Logical and physical execution of a task in HTL [1]

A task in HTL is invoked periodically on the timeline. It gets its inputs (resp. produces outputs) from (for) other tasks ports that run in the same period, or from (to) variables called communicators. Figure 6 illustrates the invocation of a task that interfaces with three communicators and other tasks. The inputs and outputs fed from and to other tasks ports define task precedences. A communicator c is specified by a period $\Pi(c)$, a set $\texttt{Type}(c)$ that defines the domain of c, and an initial value $\texttt{init}(c) \in \texttt{Type}(c)$. Communicators can be read or written periodically according to their own periods, and provide means for tasks invoked with different periods to communicate, or an interface with the external runtime environment. A task invocation can read or write a communicator at logical times that fall within the task's invocation period, and match (are divisible by) the communicator's own period. Communicators and ports are not detailed in Figure 2 for the 3TS example, but also there they provide the communication mechanism for task interaction and interaction with the external environment (eg. water-level sensor readings or pump actuation are defined through communicators).

Mode. A mode m in HTL specifies the invocation of a set of tasks, `Tasks`, in a period of time Δ, the mode period. The invocation of each task in a mode is completely defined by acyclic task precedences and read/write access to

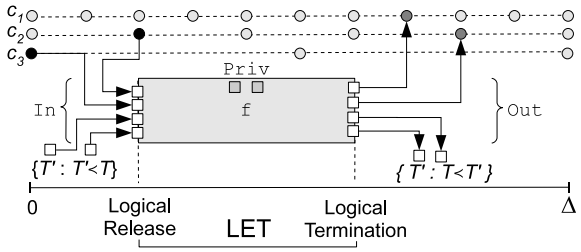


Fig. 6. A task invocation

communicators. The task invocations are established by the mode task invocation relation `Inv` composed of elements of the form (p, p') defining a dependency between an output port p of a task T in `Tasks` and an input port p' of a task T' also in `Tasks`, and of elements (p, c, t) defining a communicator read (resp. write) from (to) communicator c to input (from output) port p at time $t \in [0, \Delta]$ which is a factor of the communicator period. A mode can additionally be refined by a program, as we have already mentioned and will discuss in more detail below.

Figure 7 depicts an example mode with period $\Delta = 10$ and task set $\texttt{Tasks} = \{T_1, T_2, T_3, T_4\}$. We present some details on the task invocation relation for T_1 and T_2 involving the ports p_1 to p_7 and communicators c_1, c_2, and c_3 with periods 1, 2, and 5, respectively. This example also illustrates that the LET of a task is affected both by communicator access and task dependencies: the logical termination time of T_1 is no later than the logical termination of T_2 since there is a dependency from T_1 to T_2, and for the same reason, the logical release time of T_4 is not earlier than the release time of T_3.

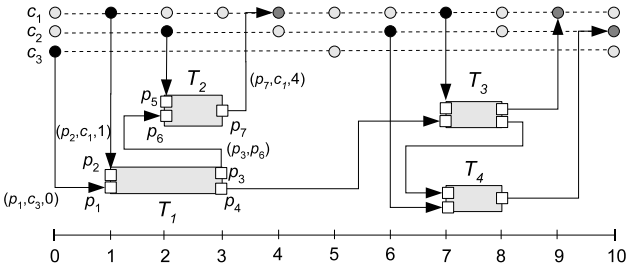


Fig. 7. Mode example

Module. A module M in HTL consists of a set of modes, `Modes`, a starting mode `start` in `Modes`, a mode switching function `switch`, and a set of ports, `Ports`, defining the scope for the ports of tasks in `Modes`. The execution of M corresponds to an execution of its modes in a sequence, starting with `start`, with mode switching taking place at the end of the active mode's period by the evaluation of `switch`. The mode switching function decides the next mode to execute, considering the last active mode and the current values of ports and communicators accessed by tasks in the scope of the module. Modules declared at the top level can have modes with distinct periods that are harmonic with the communicator periods accessed by any mode in the module.

Figure 8 illustrates the structure and a possible execution of an example module M, with 3 modes, m_1, m_2, and m_3 with periods 2, 4, and 3, respectively, with m_1 being the starting mode of M.

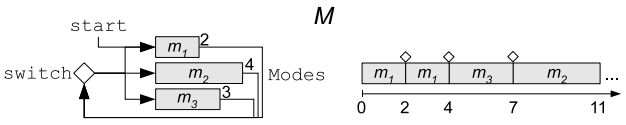


Fig. 8. A module with different mode periods

Program. A program P in HTL consists of a set of modules, `Modules`. If a program P is a top-level program, then it also has a set of communicators, `Comms`. The modules in a program execute concurrently, as illustrated in Figure 3 for the 3TS system, interacting through the program's communicators.

Hierarchical refinement. Hierarchical refinement is enabled if a subset of the tasks of a mode are declared as abstract. Abstract tasks in a mode have no associated code procedure, but are instantiated by other (concrete or abstract) refinement tasks in the mode's refinement program. A mode m with abstract task set `ATasks` must have a defined refinement program `RefP` such that the implementation of tasks in `ATasks` is partitioned across the modules of `RefP`. The other basic syntactic constraints in task refinement require that the LET of a refinement task R must be larger than that of the abstract task A it refines, as illustrated by Figure 9. This means that the refinement task is less constrained in its execution, and in that sense the abstract task is a convenient conservative abstraction for the refinement task.

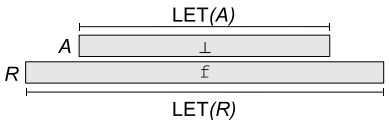


Fig. 9. LET refinement constraint

Hierarchical vs. flat programs. Hierarchical refinement does not add expressiveness to HTL, as it is possible to "flatten" any hierarchical program into a "flat" program without refinement. Hierarchical refinement does however offer the flexibility of hierarchical encapsulation, which in turn can leverage the effort in compiling a program. In general, even beyond HTL, the flat version of a hierarchical program is typically larger and takes more effort to analyze, eventually leading to state-space explosion during compilation. Also, in general, flattening a program may be an unfeasible option if components are "black boxes" with only known "public interface" (see e.g. [10]), or the internal behavior of components is too heterogeneous, requiring an agreed interface to compose them together (see e.g. [11]).

In Figure 10 we depict the structure of a flat HTL program equivalent to the hierarchical 3TS controller of Figure 2, illustrating some issues concerning flat vs. hierarchical programs. The flat program contains "flat modes", accounting

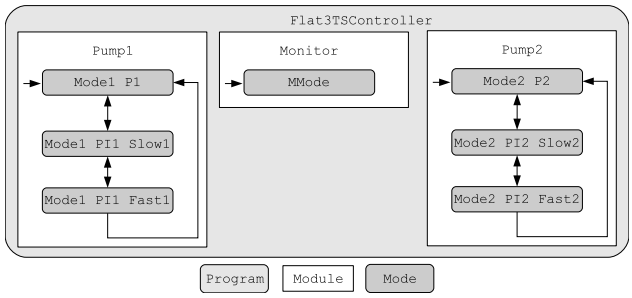


Fig. 10. Flat 3TS controller

for all possible combinations of modes in the refinement level. Furthermore, "flatness" may turn into "fatness" for each flattened component: an abstract mode like `Mode1` in Figure 2 contains abstract tasks but also concrete tasks, so in that case its concrete functionality needs to be repeated three times in the flat program. Finally, and crucially, the program becomes harder to check at the top level, as the number of possible combinations of concurrent top-level mode invocations grows exponentially. We have only one possible top-level combination in the hierarchical 3TS controller (`Mode1`/`Mode2`/`Monitor`), in contrast to nine combinations in the flat program.

C. HTL platform characterization

An HTL program, by definition, does not include a description of the platform on which it executes. The platform definition is encoded separately, or conveyed through programmer annotations.

A platform for the execution of an HTL program P is defined by: (1) A set of hosts, `Hosts`, each with a uniprocessor maintaining a mirror image of the values of all communicators, employing a preemptive EDF scheduling policy, and communicating over a reliable, time-synchronized broadcast network. (2) A distribution $\texttt{dmap} : \texttt{Modules}(P) \rightarrow \texttt{Hosts}$ mapping modules of P to hosts in the platform. Each concrete task T, declared in some module $M \in \texttt{Modules}(P)$, runs on host $\texttt{dmap}(M)$. (3) A WCET mapping $\texttt{wcet} : \texttt{Tasks}(P) \rightarrow \mathbb{Q}_{>0}$ indicating the WCET estimate for all abstract or concrete tasks declared in P. The WCET is a bound on the execution of a task procedure in the absence of any concurrency. If T is an abstract task, then $\texttt{wcet}(T)$ should be understood as an abstract annotation for the upper bound on the WCET for any refinements of T. (4) A WCTT mapping $\texttt{wctt} : \texttt{Comms}(P) \rightarrow \mathbb{Q}_{>0}$. For each communicator c, $\texttt{wctt}(c)$ is a bound on the worst-case time the network needs to transmit a single communicator datum, in the absence of concurrency.

This characterization of platform differs in the formulation of the WCTTs from the original characterization [1]. The difference is that WCTTs are characterized here as raw time needed to transmit a datum, in the absence of concurrency, rather than in the presence of worst-case concurrency. Instead of abstracting the complete network with a worst-case characterization, we consider it here in a form that allows modular transmission safety checking and modular code distribution.

III. Compilation and Runtime Patching

The desired key property of HTL programs is time-determinism, meaning that their functional and temporal behavior is repeatable, i.e., for every timed sequence of inputs there is a unique timed sequence of outputs. HTL compilation is the process of checking whether an HTL program is time-deterministic on a given, possibly distributed target platform, and of generating code that runs on that platform. HTL runtime patching is the process of modifying components of an HTL program at runtime using HTL compilation to re-check correctness and re-generate code.

A. HTL compilation

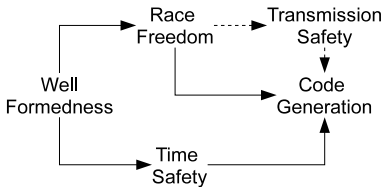


Fig. 11. HTL compilation stages

HTL compilation can be seen as proceeding in stages to check if a program is time-deterministic, as illustrated in Figure 11, before generating code. First, any program specification needs to be checked for well-formedness with respect to the syntactic constraints of the language. Then, race freedom must be established, meaning that no communicator is assigned different values at the same time. The existence of a race clearly leads to non-determinism. Along with race freedom, time safety must also be checked, in the traditional sense of real-time systems, meaning that the computation of a program on each host of the platform is schedulable, taking into account the mapping of modules to hosts and the WCET of tasks. For distributed platforms, a transmission safety check is also necessary, verifying that there exists a schedule for the networked communication between different hosts, based on the WCTT of communicators. In particular, the check verifies that race-free communicator updates can be broadcast during the respective communicators' periods before the update. Time and transmission safety are checked independently, in contrast to [1], by requiring tasks to complete so that their outputs are available at least one communicator period early.

Well-formed, race-free, time-safe, and transmission-safe programs are time-deterministic [1]. Code generation starts after checking these properties. There is a non-modular, flattening HTL compiler generating code that may be exponentially larger than the input program [1], and a modular, hierarchy-preserving HTL compiler [3] generating code that is linear in the size of the input program. The modular compiler can also compile HTL programs on the level of individual components. Below top level, the modular compiler can even check time-determinism and generate code incrementally by only considering a limited context of the component or even no context at all. Section V introduces a framework to quantify the degree of compositionality in modular compilation and applies the framework to HTL.

B. HTL runtime patching

Modular and incremental compilation in particular is not just a prerequisite for scalable software development but may also lead to interesting applications at runtime. Uncertainty in control systems, for example, makes it difficult if not impossible to design control software at compile time that is able to address infinitely many environment states at runtime. Run-time code modifications provide a way to accomplish this. However, semantics and efficiency of code modifications at runtime requires a robust foundation. Runtime patching is the process of modifying code incrementally (for efficiency) at runtime such that the resulting system behavior could be reproduced (for semantics) by a program containing complete copies of the unpatched and the patched program, switching from the unpatched to the patched copy at the time of the patch [4]. Therefore, the semantics of the language in which the patched programs are written also defines the semantics of code modifications through runtime patching.

Runtime patching can be applied to HTL programs using mode switching as a means to express the switch from unpatched to patched versions. HTL components can be patched at runtime as long as there exists a time-deterministic HTL program that can mimick the patch through mode switching. Sufficient conditions for runtime patching cold (non-executing) and even hot (executing) HTL components have been discussed elsewhere [4]. An implementation of HTL runtime patching is future work. The modularity framework and analysis presented in this paper provide a foundation for it.

IV. Modular syntax and semantics

$$
\begin{aligned}
\langle \text{Top Program} \rangle &\to \texttt{Comms} : \langle \text{Communicator} \rangle^{+} , \\
&\quad \langle \text{Program} \rangle_{\texttt{Comms},\emptyset} \cdot \\
\langle \text{Program} \rangle_{\texttt{Comms},\texttt{Ports}} &\to P : \text{Name} , \\
&\quad \texttt{Modules} : \langle \text{Module} \rangle^{+}_{\texttt{Comms},\texttt{Ports}} \cdot \\
\langle \text{Communicator} \rangle &\to c : \text{Name} , \\
&\quad \texttt{Type} : \text{Type} , \\
&\quad \texttt{init} : \texttt{Type}, \\
&\quad \Pi : \mathbb{Q}_{>0} \cdot \\
\langle \text{Module} \rangle_{\texttt{Comms},\texttt{Ports}_0} &\to M : \text{Name} , \\
&\quad \texttt{Ports} : \langle \text{Ports} \rangle^{+} , \\
&\quad \texttt{Modes} : \langle \text{Mode} \rangle^{+}_{\texttt{Comms},\texttt{Ports}_0 \cup \texttt{Ports}} , \\
&\quad \texttt{start} : \texttt{Modes}, \\
&\quad \texttt{switch} : \texttt{Modes}^{\texttt{Modes} \times \mathcal{V}(\texttt{Comms} \cup \texttt{Ports})} . \\
\langle \text{Port} \rangle &\to p : \text{Name} , \\
&\quad \texttt{Type} : \text{Type} , \\
&\quad \texttt{init} : \texttt{Type} . \\
\langle \text{Mode} \rangle_{\texttt{Comms},\texttt{Ports}} &\to m : \text{Name} , \\
&\quad \Delta : \mathbb{Q}_{>0} , \\
&\quad \texttt{Tasks} : \langle \text{Task} \rangle^{+}_{\texttt{Ports}} , \\
&\quad \texttt{Inv} \subseteq (\texttt{Ports} \times \texttt{Ports}) \cup (\texttt{Ports} \times \texttt{Comms} \times \mathbb{Q}_{\geq 0}) , \\
&\quad \texttt{RefP} : \langle \text{Program} \rangle_{\texttt{Comms},\texttt{Ports}} \mid \bot . \\
\langle \text{Task} \rangle_{\texttt{Ports}} &\to T : \text{Name} , \\
&\quad \texttt{In} \subseteq \texttt{Ports}, \\
&\quad \texttt{Out} \subseteq \texttt{Ports}, \\
&\quad \texttt{Priv} \subseteq \texttt{Ports}, \\
&\quad \texttt{f} : \mathcal{V}(\texttt{Out} \cup \texttt{Priv})^{\mathcal{V}(\texttt{In} \cup \texttt{Priv})} \mid \bot .
\end{aligned}
$$

Fig. 12. HTL abstract syntax grammar

A. Syntax

HTL programs are formally defined via an abstract syntax [1], which is implemented by compilers in a textual or

Top level semantics

$$\frac{}{s_0 = (0, v_0(P), \texttt{invoke}(P, 0))}\ (\texttt{INIT})$$

$$\texttt{E} = \{\texttt{complete}, \texttt{write}, \texttt{switch}, \texttt{read}, \texttt{time}\}$$

$$\frac{s \overset{\texttt{complete}}{\Longrightarrow}{}^{\bullet}\ s_1 \overset{\texttt{write}}{\Longrightarrow}{}^{\bullet}\ s_2 \overset{\texttt{switch}}{\Longrightarrow}{}^{\bullet}\ s_3 \overset{\texttt{read}}{\Longrightarrow}{}^{\bullet}\ s_4 \overset{\texttt{time}}{\Longrightarrow} s'}{s \longrightarrow s'}\ (\texttt{TOP})$$

$$\frac{e \in \texttt{E} \setminus \{\texttt{time}\} \quad s \overset{e}{\underset{l_1}{\Longrightarrow}} s_1 \overset{e}{\underset{l_2}{\Longrightarrow}} \ldots \overset{e}{\underset{l_n}{\Longrightarrow}} s_n \overset{e}{\not\Longrightarrow} \cdot \quad n \geq 0 \quad \forall i,\ l_i = \varepsilon \vee \forall i \neq j,\ l_i \neq l_j}{s \overset{e}{\Longrightarrow}{}^{\bullet}\ s_n}\ (\texttt{MICRO})$$

Invocation and transitions for program $(P, \texttt{Modules})$

$$\texttt{invoke}(P, t) = \{\langle M, \texttt{invoke}(M, t)\rangle \mid M \in \texttt{Modules}\}$$

$$\frac{e \in \texttt{E} \setminus \{\texttt{time}\} \quad \langle M, A_M\rangle \in A \quad (t, v, A_M) \overset{e}{\underset{l}{\Longrightarrow}} (t, v', A'_M)}{(t, v, A) \overset{e}{\underset{l}{\Longrightarrow}} (t, v', (A \setminus \{\langle M, A_M\rangle\}) \cup \{\langle M, A'_M\rangle\})}\ (\texttt{COMP1})$$

$$\frac{\forall \langle M, A_M\rangle \in A, (t, v, A_M) \overset{\texttt{time}}{\Longrightarrow} (t', v, A_M)}{(t, v, A) \overset{\texttt{time}}{\Longrightarrow} (t', v, A)}\ (\texttt{TIME1})$$

Invocation and transitions for a module $(M, \texttt{Ports}, \texttt{Modes}, \texttt{start}, \texttt{switch})$
$(\texttt{PrivPorts} = \{p \in \texttt{Priv}(T) \mid T \in \texttt{Tasks}(m) \wedge m \in \texttt{Modes}\})$

$$\texttt{invoke}(M, t) = \langle \texttt{start}, \texttt{invoke}(\texttt{start}, t)\rangle$$

$$\frac{e \in \texttt{E} \setminus \{\texttt{switch}\} \quad A = \langle m, A_m\rangle \quad (t, v, A_m) \overset{e}{\underset{l}{\Longrightarrow}} (t', v', A_m)}{(t, v, A) \overset{e}{\underset{l}{\Longrightarrow}} (t', v', A)}\ (\texttt{COMP2})$$

$$\frac{\begin{array}{c}(t, v, A_m) \overset{\texttt{switch}}{\Longrightarrow} (t, v', A'_m) \\ m_{new} = \texttt{switch}(m, v(\texttt{Comms} \cup \texttt{Ports})) \\ (m_{new} = m \wedge A_{new} = A'_m) \vee A_{new} = \texttt{invoke}(m_{new}, t)\end{array}}{(t, v, \langle m, A_m\rangle) \overset{\texttt{switch}}{\Longrightarrow} (t, [\texttt{Ports} \setminus \texttt{PrivPorts} := \bot]\, v', \langle m_{new}, A_{new}\rangle)}\ (\texttt{SWITCH1})$$

Invocation and transitions for a mode $(m, \Delta, \texttt{Tasks}, \texttt{Inv}, \texttt{RefP})$
$(\texttt{CTasks} = \{T \in \texttt{Tasks} \mid \texttt{f}(T) \neq \bot\})$

$$\texttt{invoke}(m, t) = \begin{cases} \langle t, \bot\rangle & , \texttt{RefP} = \bot \\ \langle t, \texttt{invoke}(\texttt{RefP}, t)\rangle & , \texttt{RefP} \neq \bot \end{cases}$$

$$\frac{e \in \texttt{E} \setminus \{\texttt{switch}, \texttt{time}\} \quad (t, v, A_r) \overset{e}{\underset{l}{\Longrightarrow}} (t, v', A_r)}{(t, v, \langle a, A_r\rangle) \overset{e}{\underset{l}{\Longrightarrow}} (t, v', \langle a, A_r\rangle)}\ (\texttt{REFINE})$$

$$\frac{(\texttt{RefP} = A'_r = \bot) \vee (t, v, A_r) \overset{\texttt{switch}}{\Longrightarrow} (t, v, A'_r)}{(t, v, \langle t - \Delta, A_r\rangle) \overset{\texttt{switch}}{\Longrightarrow} (t, v, \langle t, A'_r\rangle)}\ (\texttt{SWITCH2})$$

$$\frac{\begin{array}{c}t < t' \leq a + \min_{>t-a}\{\delta \mid (p, \cdot, \delta) \in \texttt{Inv} \vee \delta = \Delta\} \\ A = \langle a, A_r\rangle \quad \texttt{RefP} = \bot \vee (t, v, A_r) \overset{\texttt{time}}{\Longrightarrow} (t', v, A_r)\end{array}}{(t, v, A) \overset{\texttt{time}}{\Longrightarrow} (t', v, A)}\ (\texttt{TIME2})$$

$$\frac{\begin{array}{c}A = (a, \cdot) \qquad \forall p \in \texttt{In}(T), v(p) \neq \bot \\ T \in \texttt{CTasks} \quad t - a \leq \texttt{Termination}(T) \\ v' = [\texttt{Out}(T) \cup \texttt{Priv}(T) := \texttt{f}(T)(v(\texttt{Priv}(T) \cup \texttt{In}(T))]\, v\end{array}}{(t, v, A) \overset{\texttt{complete}}{\underset{T}{\Longrightarrow}} (t, v', A)}\ (\texttt{COMPLETE1})$$

$$\frac{\begin{array}{c}A = (a, \cdot) \qquad \forall p \in \texttt{In}(T), v(p) \neq \bot \\ T \in \texttt{CTasks} \quad t - a < \texttt{Termination}(T)\end{array}}{(t, v, A) \overset{\texttt{complete}}{\underset{T}{\Longrightarrow}} (t, v, A)}\ (\texttt{COMPLETE2})$$

$$\frac{(p, p') \in \texttt{Inv} \quad v(p) \neq \bot \quad v(p') = \bot}{(t, v, A) \overset{\texttt{read}}{\Longrightarrow} (t, [p' := v(p)]\, v, A)}\ (\texttt{DEP})$$

$$\frac{\begin{array}{cc}A = (a, \cdot) & (c, p, t - a) \in \texttt{Inv} \\ p \in \texttt{In}(T) \quad v(p) = \bot & T \in \texttt{CTasks}\end{array}}{(t, v, A) \overset{\texttt{read}}{\Longrightarrow} (t, [p := v(c)]\, v, A)}\ (\texttt{READ})$$

$$\frac{\begin{array}{cc}A = (a, \cdot) & (c, p, t - a) \in \texttt{Inv} \\ p \in \texttt{Out}(T) \quad v(p) \neq \bot & T \in \texttt{CTasks}\end{array}}{(t, v, A) \overset{\texttt{write}}{\underset{p,c}{\Longrightarrow}} (t, [c := v(p)]\, v, A)}\ (\texttt{WRITE})$$

Fig. 13. HTL semantics

visual representation [8], [12]. Figure 12 depicts the abstract syntax in grammar-like notation, defining the various blocks and the scope of ports and communicators within a program. In the syntax presentation we use the symbol $+$ as in regular expressions, for the set of non-empty sequences of elements in a given set. In addition to the syntax rules (and simple scope and naming constraints detailed in [1]), a well-formed HTL program must conform to a set of constraints regarding the LET of tasks and hierarchical refinement, discussed below.

LET constraints. In a mode m, with period Δ, a task invocation may access (read/write) communicators only at logical times in $[0, \Delta]$ that are factors of the communicators' periods. No read may happen at time Δ, and no write at time 0. Also, Δ must be a factor of every communicator period referenced by other modes in the same module, in order that mode switching to any mode induces an aligned time window for all communicators. In addition, the task precedence relation $\prec_m$ for the tasks in m must be acyclic.

Each task $T \in \texttt{Tasks}(m)$ must have a well-defined LET interval $\texttt{LET}(T) = [\texttt{Release}(T), \texttt{Termination}(T)]$ such that $\texttt{Release}(T)$ is the maximum of times t such that $(p, c, t) \in \texttt{Inv}(m)$ for $p \in \texttt{In}(T)$ or such that $t = \texttt{Release}(T_0)$ for $T_0 \prec_m T$, and $\texttt{Termination}(T)$ is the minimum of times t such that $(p, c, t + \Pi(c)) \in \texttt{Inv}(m)$ for $p \in \texttt{Out}(T)$ or such that $t = \texttt{Termination}(T_0)$ for $T \prec_m T_0$. Note that for communicator writes we take into account the transmission time, which takes at most one communicator period $\Pi(c)$. Hence, if $(p, c, t) \in \texttt{Inv}(m)$ for $p \in \texttt{Out}(T)$, then at time t the corresponding p-output of T is both written to the communicator c and has already been transmitted via the network.

Refinement constraints. A mode m is refined if and only if it contains at least one abstract task, i.e., $A \in \texttt{Tasks}(m)$ with $\texttt{f}(A) = \bot$. In that case, each refinement mode m' within $\texttt{RefP}(m)$ must have the same period as m, and each of its tasks must be a proper refinement of a unique abstract task A.

A task T is a proper refinement of A if and only if the following four conditions are met: (1) The LET of T contains that of A, $\texttt{LET}(T) \supseteq \texttt{LET}(A)$; (2) Any precedence of T is a precedence of A, directly or by refinement: if $T_0 \prec_m T$ then T_0 is either a refinement of a precedence of A, or a concrete task in the parent mode that is a precedence of A; (3) T writes to all of the output ports of A, $\texttt{Out}(T) \supseteq \texttt{Out}(A)$; and (4) T only writes to communicators to which A also writes. We note that condition (4) is not present in [1], but it certainly characterizes a well-defined refinement and is necessary for establishing race freedom in the simple way hinted in [1]. Whenever an execution platform is specified for the program,

the following condition must also be satisfied: (5) The WCET of T is less than or equal to the WCET of A. Note that an abstract task is not an interface for communication of refinement tasks, but merely a placeholder for schedulability ensuring race freedom.

B. Semantics

An operational semantics was given for HTL in [1], in close relation to the time-triggered execution of the E machine [9]. Here we briefly present a more abstract modular semantics defined compositionally in terms of the HTL blocks.

A state of a top program P is a triple $s = (t, v, A)$ where $t \in \mathbb{Q}_{\geq 0}$ is the current time, v is a valuation of the variables of P, and A is an activation of P. Activations are defined compositionally: for a program P, an activation $A \in \texttt{Act}(P)$ is a set

$$A = \{\langle M, A_M\rangle \mid M \in \texttt{Modules}(M), A_M \in \texttt{Act}(M)\}$$

where $A_M = \langle m, A_m\rangle \in \texttt{Act}(M)$ is an activation for the module M, with $m \in \texttt{Modes}(M)$ being the active mode, and $A_m \in \texttt{Act}(m)$ its activation. A mode activation for m is another pair $A_m = \langle a, A_r\rangle$, with activation time $a \in \mathbb{Q}_0^+$, and a refinement activation $A_r = \bot$ if $\texttt{RefP}(m) = \bot$, or $A_r \in \texttt{Act}(\texttt{RefP}(m))$ otherwise.

A program trace $s_0 \longrightarrow s_1 \longrightarrow s_2 \longrightarrow \cdots$ is defined by a two-layer operational semantics shown in Figure 13. The initial program state of P (INIT rule) is $s_0 = (0, v_0(P), \texttt{invoke}(P, 0))$, where $v_0(P)$ assigns the initial value $\texttt{init}(\texttt{x})$ to each communicator or port x in P. In the initial activation $\texttt{invoke}(P, 0)$, the active mode of each module is set to be the specified start mode. A successor state s' of s is reached after a sequence of task completions, writes, reads, and mode switches, defining "micro"-successors, before an elapse of time (TOP rule). We write e for any of these events (comprising the set of events E). Each e-"micro"-successor, except for the time elapse, comprises a closed sequence of e-events with distinct labels or no labels at all (MICRO rule). The labels are a convenience artifact that allows us to model that any (complete or write) event transition is enabled only once at a time instant.

The handling of program events is modeled by the second layer of the semantics, modularly. A program event is either an independent event in one of its modules (COMP1) or a synchronized elapse of time in all of its modules (TIME1). The behavior of a module reduces to the behavior of the current active mode (COMP2), until mode switching is due and the active mode may change (SWITCH1). At the level of modes, the behavior is defined by the concrete tasks of the mode, and compositionally by the refinement program (REFINE). Switching is enabled at the end of the mode's period (SWITCH1), and time elapses with an offset up to the next logical event in the mode (TIME2). Both switching and time elapsing are synchronized events with the mode's refinement program. In the lowest level of concrete task invocations in a mode, each task completion is modeled as completing non-deterministically at an instant that is within the LET of the task (COMPLETE1 and COMPLETE2). Other than that, the state may change by appropriate updates of ports and communicators (DEP, READ, and WRITE) due to task precedences and communicator reads/writes. Note that the semantics defines only the time-deterministic behavior of the program and not an actual execution on a platform. In this respect it qualifies as "abstract".

V. Modularity

A. Modularity framework

We consider the following framework, illustrated in Figure 14. Given a top-level program P_o for which property or aspect φ holds, and a component C_o within P_o that is updated to a component (patch) C yielding top-level program P, we wish to establish φ for P. Let $\mathcal{A}$ be the algorithm used for establishing φ. The modularity of φ given $\mathcal{A}$, in regard to P and C, is a pair

$$\big(\mathcal{D}_\varphi^{\mathcal{A}}(C, P), \mathcal{C}_\varphi^{\mathcal{A}}(C, P)\big)$$

where $\mathcal{D}_\varphi^{\mathcal{A}}(C, P)$ stands for the part of P that needs to be re-analyzed in order to establish φ, called the modular dependency context and $\mathcal{C}_\varphi^{\mathcal{A}}(C, P)$ is the complexity function measuring the effort in doing so in terms of the size of the dependency context, called the modular complexity.

For comparison, we also denote by $\overline{\mathcal{C}}_\varphi^{\mathcal{A}}(P)$ the complexity of the total effort of checking φ for P using $\mathcal{A}$. Note that, in the modularity framework, we exclude the case $C_o = P_o$, i.e., $C = P$, since then the modular complexity equals $\overline{\mathcal{C}}_\varphi^{\mathcal{A}}(P)$.

Hence, our framework is one for incremental compilation, applicable when a certain component of a program is updated, and thus the program requires re-compilation. The framework also subsumes plain incremental compilation, in case when the old component C_o is the "empty component".

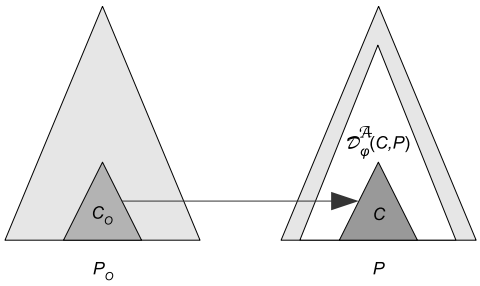


Fig. 14. Modularity

The modular dependency context $\mathcal{D}_\varphi^{\mathcal{A}}(C, P)$ alone is already a good measure of modularity, and is in close relation to the modular complexity. In the best-case scenario, we will have $\mathcal{D}_\varphi^{\mathcal{A}}(C, P) = C$, meaning that only C needs to be re-checked. This is the case when φ (and $\mathcal{A}$) is "fully modular" for C with regard to P, or C can be checked to be a refinement of C_o in regard to φ. In the worst-case scenario, we will have $\mathcal{D}_\varphi^{\mathcal{A}}(C, P) = P$ and $\mathcal{C}_\varphi^{\mathcal{A}}(C, P) = \overline{\mathcal{C}}_\varphi^{\mathcal{A}}(P)$, meaning that P needs to be fully re-checked, i.e., φ (or $\mathcal{A}$) is "non-modular" for C in the context of P.

From now on, if φ, $\mathcal{A}$, C, and P are fixed and clear, we write $\mathcal{D}$ and $\mathcal{C}$, for the dependency context and the complexity, respectively. Similarly, we write $\overline{\mathcal{C}}$ for the total complexity.

B. HTL modularity aspects

We characterize HTL modularity for all cases of interest. For each property φ, from the compilation stages of Figure 11, we discuss an algorithm $\mathcal{A}$ used to validate φ on P. Before presenting the modularity of the property, i.e., $\mathcal{D}$ and $\mathcal{C}$ for all possible components, we first discuss $\overline{\mathcal{C}}$, the total complexity of checking φ by $\mathcal{A}$. Note that in case of a dependency between properties, i.e., an arrow in Figure 11, we only consider the effort of checking the property of interest and assume that the needed predecessor property has already been checked. For example, in order to establish transmission safety a program needs to be race free (and in order to be race free, it must be well-formed). When discussing the complexity of the transmission-safety check, we assume the program is already proven race free (and hence also well-formed), and consider only the complexity that is essential to the transmission safety check.

Our results are summarized in Table I, showing that top-level time safety is non-modular, whereas all other properties are modular. Moreover, refinement-level race freedom, transmission safety, refinement-level time safety, and code generation are fully modular properties. In the rest of the paper we discuss the results presented in Table I. Let us start by introducing the needed notation.

A program P has a refinement height r_P, and the following upper bounds on its size: n_c communicators, n_M top modules, n_m modes per top module, n_T tasks per mode, n_w communicator writes per task, n_a communicator accesses per task, n_p ports per task, and periods of modes with maximum value $\Delta_{\max}$. We also use the notation $n^P_{m\downarrow}$ for the bound on the total number of modes in P calculated as $n^P_{m\downarrow}(r_P) = (n_M \cdot n_m)^{r_P+1}$, and $n^P_{T\uparrow}$ for the bound of the total number of top-level tasks in P calculated as $n^P_{T\uparrow} = n_M n_m n_T$.

We distinguish between top-level and refinement-level components. A top-level component can have a root node module, mode, or task. A refinement-level component can have a root node program, module, mode, or task. Similar as for programs, r_C denotes the refinement height of a component C, $n^C_{m\downarrow}$ an upper bound on the total number of modes in it, and $n^C_{T\uparrow}$ an upper bound on the number of top-level tasks. These two notions can be calculated for each type of a component. For example, if C is a refinement program, then $n^C_{m\downarrow} = (n_M \cdot n_m)^{r_C+1}$ and $n^C_{T\uparrow} = 0$; whereas if C is a top-level component with root node a module, then $n^C_{m\downarrow} = (n_M \cdot n_m)^{r_C} \cdot n_m$ and $n^C_{T\uparrow} = n_m\, n_T$.

For simplicity, we assume that the mode switching function of each module induces a fully connected mode-switching graph. This is not a restriction but actually creates a worst-case scenario since it subsumes all possible mode-switching behaviors.

Well-formedness. The verification of well-formedness of a program P with respect to the LET and refinement constraints discussed in Section IV is performed inductively on the structure of the program and it can be achieved in linear time (one pass) depending on the product of the total number of modes, number of tasks per mode, and number of ports per task, i.e. $\overline{\mathcal{C}} = \mathrm{O}(n^P_{m\downarrow}(r)\ n_T\ n_p)$. Checking other syntactic constraints is performed in less time, and therefore subsumed by this complexity bound.

For modular compilation, the well-formedness check is also linear, now in the size of the component. The dependency context is C itself (although each refinement task needs to be compared also to its parent task) and the modular complexity is $\mathcal{C} = \mathrm{O}(n^C_{m\downarrow}(r_C)\ n_T\ n_p)$.

Race freedom. A race happens in the execution of a program when two concurrent task invocations write the same communicator at the same time, compromising time-determinism.

We consider a simple race-free sufficiency algorithm. It uses the fact that if no two concurrent tasks write to the same communicators, regardless of the time when the communicators are written, then race freedom is ensured. Moreover, using condition (4) of the refinement constraints in Section IV-A, we observe that two concurrent tasks in an HTL program write to the same communicators if and only if there exist two top-level concurrent tasks in the program that write the same communicators. Hence, for race freedom it is sufficient to check that no two top-level concurrent tasks write to the same communicators. At top level, concurrent tasks are any two tasks within the same mode, or any two tasks in different top-level modules.

More precisely, let $\texttt{write}(T)$ be the set of communicators to which task T writes, and let $\texttt{write}(M)$ be the set of communicators written by any task in any mode of M. We need to ensure that

1. for all top-level modules M_1 and M_2 we have $\texttt{write}(M_1) \cap \texttt{write}(M_2) = \emptyset$, and
2. for every top level mode m, and each two tasks T_1 and T_2 of m, it holds that $\texttt{write}(T_1) \cap \texttt{write}(T_2) = \emptyset$.

The check of 1. can be done in linear time in the number of modules and number of communicators, whereas the check of 2. is linear in the total number of writes of all top-level task. Hence, $\overline{\mathcal{C}} = \mathrm{O}(n^P_{T\uparrow}\ n_w + n_M\ n_c)$.

As for the modularity cases of interest, if C is a well-formed refinement-level component, then there is nothing to check: the dependency context is C, and the complexity is constant time, actually zero time. If C is a top-level component, then re-checking race freedom requires checking 2. for the tasks in C and re-checking 1. Hence, the dependency context is the whole program P, but not all of P needs to be re-checked. There is still some modularity involved, the top-level tasks outside C need not be re-checked. As a result, we get that $\mathcal{C} = \mathrm{O}(n^C_{T\uparrow}\ n_w + n_M\ n_c)$.

Transmission safety. Transmission safety requires that the networked communication of a program, i.e., the broadcast of communicator values, is schedulable, meaning that any transmission of a value of any communicator fits into the communicator's period instance.

In the worst-case transmission scenario of a race-free program, every communicator is updated at every period. Since transmitting an update takes time at most the WCTT of the communicator, scheduling the network communication amounts to scheduling a set of periodic tasks (one per each

φ	C	$\mathcal{D}^{\mathcal{A}}_{\varphi}(C,P)$	$\mathcal{C}^{\mathcal{A}}_{\varphi}(C,P)$	$\overline{\mathcal{C}}^{\mathcal{A}}_{\varphi}(P)$
Well-formedness	any	C	$n^C_{m\downarrow}\ n_T\ n_p$	$n^P_{m\downarrow}\ n_T\ n_p$
Race freedom	top	P	$n^C_{T\uparrow}\ n_w + n_M\ n_c$	$n^P_{T\uparrow}\ n_w + n_M\ n_c$
	ref.	C	1	
Transmission safety	any	C	1	n_c
Time safety	top	P	$(n_m\ \Delta_{max})^{n_M}$	$(n_m\ \Delta_{max})^{n_M}$
	ref.	C	1	
Code generation	any	C	$n^C_{m\downarrow}\ (n_T\ n_a + n_m)$	$n^P_{m\downarrow}\ (n_T\ n_a + n_m)$

n_a	number of communicator accesses per task	n_c	number of communicators	n_M	number of modules per program
n_m	number of modes per module	n_p	number of ports per task	n_T	number of tasks per mode
n_w	number of communicator writes per task	$n^C_{m\downarrow}$	total number of modes in C	$n^P_{m\downarrow}$	total number of modes in P
$n^C_{T\uparrow}$	number of top-level tasks in C	$n^P_{T\uparrow}$	number of top-level tasks in P	$\Delta_{\max}$	maximal value of mode periods

TABLE I
MODULARITY ASPECTS OF HTL

communicator) with execution times equal to the corresponding communicator WCTTs and periods equal to the corresponding communicator periods.

Hence, for network architectures that are flexible with respect to the choice of a scheduling algorithm, a simple utilization based scheduling test and algorithm (e.g. EDF or rate-monotonic scheduling), linear in the number of communicators in the system, suffices to establish transmission safety, $\overline{\mathcal{C}} = \mathrm{O}(n_c)$. An example of such a flexible architecture is an FFT-CAN bus [13]. Also statically scheduled networks like the simplest forms of TDMA buses may allow for a simple utilization-based schedulability check.

Transmission safety is fully modular although the communicators in a program are global: if the original program P_o is transmission safe, then all communicators have already been taken into account in P_o and a change of a component does not require any additional check. So, the dependency context is always C and $\mathcal{C} = \mathrm{O}(1)$.

Time safety. For time safety, a program needs to be checked for schedulability of computation, taking into account the target platform (WCETs) and the program specification (LETs). Time safety can be checked independently from the network communication (the transmission-safety check). The main HTL schedulability result, enabled by the refinement constraints, is that time safety of any program P is ensured by schedulability of all (abstract and concrete) top-level tasks. In such a case, any schedule for the top-level tasks is robust and sustainable (in the sense of e.g. [14], [15]) to a replacement of an abstract task by any of its refinements.

For schedulability at the top level, standard periodic-task scheduling techniques can be used (c.f. [16]). In the absence of mode switching, with a single mode per module, the top level of P is a periodic-task system with task precedences and the complexity of checking top-level schedulability is exponential in the mode periods: $\mathrm{O}(\Delta^{n_M}_{max})$ [17]. This can be improved for special cases, for which there are tractable schedulability tests and algorithms, e.g. synchronous periodic-task systems with deadlines equal to or less than periods. In the general case with mode switching, the top-level schedulability check must consider in the worst case $\mathrm{O}({n_m}^{n_M})$ combinations of concurrent modes at the top level. There is no need to check anything but well-formedness for refinement programs. Hence, the total complexity for checking time safety of a program P is $\overline{\mathcal{C}} = \mathrm{O}((n_m\ \Delta_{max})^{n_M})$.

Let us consider the two cases of interest for modularity analysis. If C is a well-formed refinement-level component, then $\mathcal{D} = C$ and the modular complexity is $\mathcal{C} = \mathrm{O}(1)$ since there is nothing to check. If C is a top-level component, the top-level time-safety analysis of P must be fully re-checked, i.e., time safety is non-modular in this case with dependency context $\mathcal{D} = P$ and modular complexity $\mathcal{C} = \mathrm{O}((n_m\ \Delta_{max})^{n_M})$.

Code generation. There is a flattening HTL compiler targeting the E machine [1] and a modular, hierarchy-preserving HTL compiler targeting the HE machine [3]. A detailed comparison of both compilers can be found in [5]. We highlight the differences, and characterize the complexity and modularity for the HE-compiler.

The more recent HE-compiler allows modular code generation. The worst-case code size per mode generated by the HE-compiler depends linearly on the number of communicator accesses (n_a) for each task and the number of modes in the same module (for the evaluation of mode switching), c.f. [3] for more details. Hence, the size of the generated code per mode is $\mathrm{O}(n_T\ n_a + n_m)$ and the overall complexity of generating code for P is $\overline{\mathcal{C}} = \mathrm{O}(n^P_{m\downarrow}(r_P)\,(n_T\,n_a + n_m))$. The code for each component C can be generated independently from the code for the rest of the program. So, we have that for any component C, $\mathcal{D} = C$ and $\mathcal{C} = \mathrm{O}(n^C_{m\downarrow}(r_C)(n_T n_a + n_m))$. To be precise, this holds for any component that contains at least one mode. For the corner case of a component with root node task, the modular complexity simplifies to $\mathcal{C} = \mathrm{O}(n_a)$.

Beyond code generation, there is the aspect of code distribution across multiple hosts. In the HTL characterization in [1] different modules exchanged values of ports, rather than values of communicators. As a consequence of exchanging values of ports, the code for (almost) the entire program had to be

Performance Evaluation 66 (2009) 435–452

Contents lists available at ScienceDirect

Performance Evaluation

journal homepage: www.elsevier.com/locate/peva

Compositionality for Markov reward chains with fast and silent transitions

J. Markovski [a,*], A. Sokolova [b], N. Trčka [a], E.P. de Vink [a]

[a] Formal Methods Group, Eindhoven University of Technology, Den Dolech 2, 5612AZ Eindhoven, The Netherlands

[b] Computational Systems Group, University of Salzburg, Jakob-Haringer-Straße 2, 5020 Salzburg, Austria

ARTICLE INFO

Article history:
Received 25 February 2008
Received in revised form 13 November 2008
Accepted 7 January 2009
Available online 15 January 2009

Keywords:
Markov reward chains
Fast transitions
Silent transitions
Parallel composition
Aggregation

ABSTRACT

A parallel composition is defined for Markov reward chains with stochastic discontinuity, and with fast and silent transitions. In this setting, compositionality with respect to the relevant aggregation preorders is established. For Markov reward chains with fast transitions the preorders are τ-lumping and τ-reduction. Discontinuous Markov reward chains are 'limits' of Markov reward chains with fast transitions, and have related notions of lumping and reduction. Markov reward chains with silent transitions are equivalence classes of Markov reward chains with fast transitions and come equipped with the lifted preorders $\tau_\sim$-lumping and $\tau_\sim$-reduction. In total, six compositionality results are presented. Additionally, the parallel operators involved are related by a continuity result.

1. Introduction

Compositionality is a central issue in the theory of concurrent processes. Discussing compositionality requires three ingredients: (1) a class of processes or models; (2) an operation to compose processes; and (3) a notion of behaviour, usually given by a semantic preorder or equivalence relation on the class of processes. For the purpose of this paper, we will have semantic preorders and the parallel composition as operation. Therefore, the compositionality result can be stated as

$$\mathsf{P}_1 \geqslant \overline{\mathsf{P}}_1, \qquad \mathsf{P}_2 \geqslant \overline{\mathsf{P}}_2 \Longrightarrow \mathsf{P}_1 \parallel \mathsf{P}_2 \geqslant \overline{\mathsf{P}}_1 \parallel \overline{\mathsf{P}}_2,$$

where P_1, P_2, $\overline{\mathsf{P}}_1$, and $\overline{\mathsf{P}}_2$ are arbitrary processes and $\parallel$ and $\geqslant$ denote their parallel composition and the semantic preorder relation, respectively. Hence, compositionality enables the narrowing of a parallel composition by composing simplifications of its components, thus avoiding the construction of the actual parallel system. In this paper, we study compositionality for augmented types of Markov chains.

Homogeneous continuous-time Markov chains, Markov chains for short, are among the most important and wide-spread analytical performance models. A Markov chain is given by a graph with nodes representing states and outgoing arrows labelled by exponential rates determining the stochastic behaviour of each state. An initial probability vector indicates which states may act as starting ones. Markov chains often come equipped with rewards that are used to measure their performance, such as throughput, utilisation, etc. (cf. [1]). In this paper, we focus on state rewards only, and refer to a Markov chain with rewards as a Markov reward chain. A state reward is a number associated to a state, representing the rate at which gain is received while the process resides in the state. Transition (impulse) rewards [1] can similarly be dealt with.

* Corresponding author. Tel.: +31 641920544; fax: +31 492475361.
E-mail addresses: j.markovski@tue.nl (J. Markovski), anas@cs.uni-salzburg.at (A. Sokolova), n.trcka@tue.nl (N. Trčka), evink@win.tue.nl (E.P. de Vink).

0166-5316/$ – see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.peva.2009.01.001

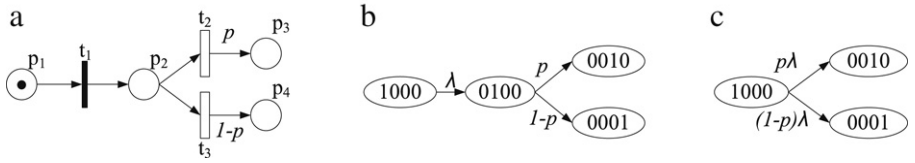


Fig. 1. (a) A generalised stochastic Petri net, (b) its extended reachability graph, and (c) the derived Markov chain.

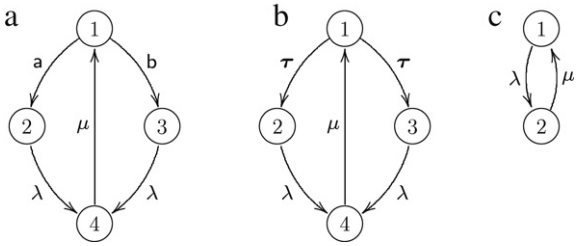


Fig. 2. (a) An Interactive Markov Chain, (b) the intermediate model with τ-transitions, and (c) the induced Markov chain.

To cope with the ever growing complexity of systems, several performance modelling techniques have been developed to support the compositional generation of Markov reward chains. This includes stochastic process algebras [2,3], (generalised) stochastic Petri nets [4,5], probabilistic I/O automata [6,7], stochastic automata networks [8], etc. The compositional modelling enables composing a bigger system from several smaller components. The size of the state space of the resulting system is in the range of the product of the sizes of the constituent state spaces. Hence, compositional modelling usually suffers from state space explosion.

In the process of compositional modelling, performance evaluation techniques produce intermediate constructs that are typically extensions of Markov chains featuring transitions with communication labels [2–8]. In the final modelling phase, all labels are discarded and communication transitions are assigned instantaneous behaviour. The models that we consider here are intended to support direct performance analysis of systems that exhibit both stochastic and instantaneous behaviour. We elaborate this for two simple modelling examples using generalised stochastic Petri nets [4] and Interactive Markov Chains [2].

Example 1. Fig. 1(a) depicts an example of a generalised stochastic Petri net with its corresponding reachability graph in Fig. 1(b). The graph contains the markings of the only token placed initially in p_1. The vanishing marking is 0100 because of the enabled immediate transitions t_2 and t_3 with probabilities p and $1 - p$, respectively. The Markov chain in Fig. 1(c) is obtained by a reduction procedure that splits the incoming normal rate λ according the probabilities of the vanishing marking into two rates $p\lambda$ and $(1 - p)\lambda$ that reach the final markings 0010 and 0001, respectively.

Next, consider the Interactive Markov Chain depicted in Fig. 2(a). Assuming that the system is closed, the transitions labelled by a and b are renamed into the instantaneous transition τ and an equivalent model is obtained. This intermediate transition system is depicted in Fig. 2(b). Now, assume that the process in Fig. 2(b) starts from state 1. There it exhibits classical non-determinism, i.e., the probability of taking the τ-transitions is unspecified. Note, however, that the process has the same behaviour in states 2 and 3. No matter which transition is taken from state 1, after performing an instantaneous τ-transition and delaying exponentially with rate λ, the process enters state 4. According to a bisimulation-based reduction procedure the instantaneous transitions are eliminated and the performance of the process in Fig. 2(b) is considered to be the performance of the Markov chain in Fig. 2(c).

In both cases in Example 1 the intermediate models depicted in Fig. 1(b) and Fig. 2(b) are treated only on syntactic level and their performance is defined to be the performance of the final Markov chains from Fig. 1(c) and Fig. 2(c). Here, we wish to treat the intermediate models as stochastic processes with well-defined notion of performance, which corresponds to the performance of the resulting Markov chain. Markov reward chains with fast transitions were introduced in [9–11] to formalise stochastically the elimination of immediate transitions and the vanishing markings of Fig. 1(b). They are extensions of the standard Markov reward chains with transitions decorated with a real-valued linear parameter. To capture the intuition that the labelled transitions are instantaneous, a limit for the parameter to infinity is taken. The resulting process is a generalisation of the standard Markov reward chain that can perform infinitely many transitions in a finite amount of time. This model was initially studied in [12,13] without rewards, and it is called a (stochastically) discontinuous Markov reward chain. The process exhibits stochastic discontinuity and it is often considered pathological. However, as shown in [13,14,5], it proves very useful for the explanation of results. To deal with the non-deterministic case of Fig. 2(b), Markov reward chains with silent transitions were introduced, which generalise Markov reward chains with fast transitions by leaving the linear parameter unspecified.

Here, we consider discontinuous Markov reward chains, Markov reward chains with fast transitions, and Markov reward chains with silent transitions. To summarise, these three models are intimately related: Markov reward chains with fast

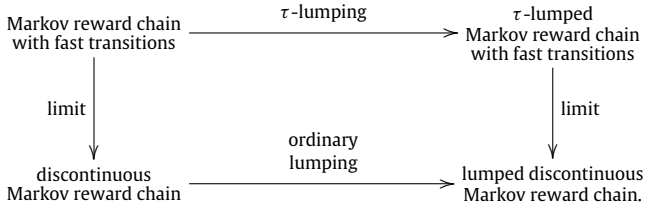
and silent transitions are used for modelling, but some notions for these processes are expressed asymptotically in terms of discontinuous Markov reward chains. A limiting process of a Markov reward chain with fast transitions is a discontinuous Markov reward chain; a Markov reward chain with silent transitions is identified with an equivalence class of a relation $\sim$ on Markov reward chains with fast transitions relating chains with the 'same shape of fast transitions'. We define the parallel composition of all models in the vein of standard Markov reward chains using Kronecker products and sums [15].

As already mentioned, compositional modelling may lead to state space explosion. Current analytical and numerical methods can efficiently handle Markov reward chains with millions of states [16,17]. However, they only alleviate the problem and many real world problems still cannot be feasibly solved. Several aggregation techniques have been proposed to reduce the state space of Markov reward chains. Ordinary lumping is the most prominent one [18,15]. The method partitions the state space into partition classes. In each class, the states exhibit equivalent behavior for transiting to other classes, i.e., the cumulative probability of transiting to another class is the same for every state of the class. If non-trivial lumping exists, i.e., at least one partition contains more than one state, then the method produces a smaller Markov chain that retains the performance characteristics of the original one. For example, the expected reward rate at a given time is the same for the original as for the reduced, so-called lumped, process.

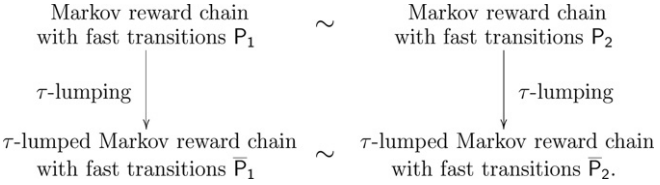
Another lumping-based method is exact lumping [19,15]. This method requires that each partition class of states has the same cumulative probability of transiting to every state of another class and, moreover, each state in the class has the same initial probability. The gain of exact lumping is that the probabilities of the original process can be computed for a special class of initial probability vectors by using the lumped Markov reward chain only.

A preliminary treatment of relational properties of lumping-based aggregations of Markov chains has been given in [20]. It has been shown that the notion of exact lumping is not transitive, i.e., there are processes which have exactly lumped versions that can be non-trivially exactly lumped again, but the original process cannot be exactly lumped directly to the resulting process. On the other hand, ordinary lumping of Markov reward chains is transitive and, moreover, it has a property of strict confluence. Strict confluence means that whenever a process can be lumped using two different partitions, there is always a smaller process to which the lumped processes can lump to.

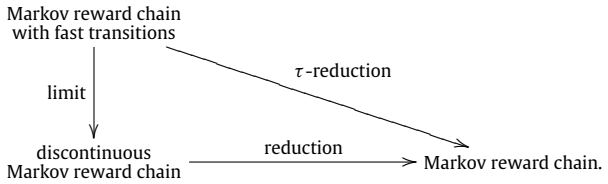
Coming back to our models of interest, ordinary lumping is defined for discontinuous Markov reward chains in [9–11]. It is an extension of the standard lumping method that accounts for instantaneous states as well. A lumping method is also defined for Markov reward chains with fast transitions, referred to as τ-lumping [9–11]. The general idea is that two states can be lumped together whenever they can be lumped in the asymptotic discontinuous Markov reward chain, when the parameter tends to infinity. The following commuting diagram gives the agreement of ordinary and τ-lumping. The diagram also shows that τ-lumping is a proper extension of ordinary lumping for standard Markov reward chains.



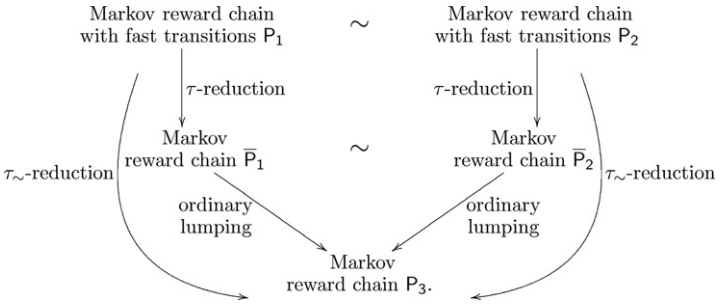
For Markov reward chains with silent transitions, a lifting of τ-lumping to the $\sim$ -equivalence classes is proposed, referred to as $\tau_\sim$-lumping [9–11]. More precisely, a partition is a $\tau_\sim$-lumping of a Markov reward chain with silent transitions, if it is a τ-lumping for every Markov reward chain with fast transitions in its $\sim$-equivalence class and the τ-lumped process does not depend on the choice of the representative. This situation is depicted in the following figure.



In addition, [10,11] study an aggregation method by reduction that eliminates the stochastic discontinuity and reduces a discontinuous Markov reward chain to a Markov reward chain. The reduction method is an extension of a well-known method in perturbation theory [21,22,13]. Its advantage is the ability to split states. The lumping method, in contrast, provides more flexibility: also states that do not exhibit discontinuous behavior can be aggregated. The reduction-based aggregation straightforwardly extends to τ-reduction of Markov reward chains with fast transitions [10,11]:



In the case of Markov reward chains with silent transitions, a direct lifting of the τ-reduction to equivalence classes does not aggregate many processes, as most of the time the reduced process depends on the actual fast transitions [10,11]. In an attempt to remedy the effect of the fast transitions, we combine τ-reduction and standard ordinary lumping for Markov reward chains to obtain $\tau_\sim$-reduction as depicted below.[1]



Hence, a Markov reward chain with silent transitions can be $\tau_\sim$-reduced if all Markov reward chains with fast transitions in its equivalence class τ-reduce to Markov reward chains that can be ordinary lumped to the same Markov reward chain.

Both the lumping-based and the reduction-based aggregation method induce semantic relations. Namely, for two processes P and $\overline{\mathsf{P}}$, we say that $\mathsf{P} \geqslant \overline{\mathsf{P}}$ if $\overline{\mathsf{P}}$ is an aggregated version of P. In our initial study [23], we investigated the properties of these relations for discontinuous Markov reward chains and Markov reward chains with fast transitions. Here we extend the approach to include Markov reward chains with silent transitions as well.

As already mentioned, compositionality is very important as it allows us to aggregate the smaller parallel components first, and then to combine them into the complete aggregated system. This approach can find its way into Markovian model-checkers [24–26] where a huge Markov chain can be (syntactically) decomposed to feasibly-sized components, which can then be aggregated to obtain the aggregated version of the original process. The technique also serves as a validation of existing bisimulation-based reduction methods that treat the intermediate performance models as transition systems. We show that the relations induced by the lumping and reduction methods are indeed preorders, i.e., reflexive and transitive relations. Having all the ingredients in place, we show the compositionality of the aggregation preorders with respect to the defined parallel composition(s). We also show continuity of the parallel composition(s). In short, the parallel operators preserve the diagrams above.

The structure of the rest of the paper is as follows. We start by recalling the three types of Markovian models in Section 2. Section 3 and Section 4 focus on the aggregation methods based on lumping and reduction for each of the models, respectively. Therefore, Section 2 to Section 4 can be understood as an overview of a fairly recent line of research and an introduction to the results that follow. In Section 5, we show that the aggregation methods define preorders on the models. Section 6 contains the main results of the paper: compositionality of the new parallel operators for each type of Markov chains with respect to both aggregation preorders. Section 7 wraps up with conclusions. Throughout the paper we present examples to illustrate the proposed approaches. Whenever we use results from (our) previous work, we state them as propositions with appropriate references. The novel results are stated as theorems.

Notation All vectors are column vectors if not indicated otherwise. By $\mathbf{1}^n$ we denote the vector of n 1's; by $\mathbf{0}^{n\times m}$ the $n \times m$ zero matrix; by I^n the $n \times n$ identity matrix. We omit the dimensions n and m when they are clear from the context. By $A[i, j]$ we denote an element of the matrix $A \in \mathbb{R}^{m\times n}$ assuming $1 \leqslant i \leqslant m$ and $1 \leqslant j \leqslant n$. We write $A \geqslant 0$ when all elements of A are non-negative. The matrix A is called stochastic if $A \geqslant 0$ and $A \cdot \mathbf{1} = \mathbf{1}$. By A^{T} we denote the transpose of A.

Let $\mathcal{S}$ be a finite set. A set $\mathcal{P} = \{S_1, \ldots, S_N\}$ of N subsets of $\mathcal{S}$ is called a partition of $\mathcal{S}$ if $\mathcal{S} = S_1 \cup \cdots \cup S_N$, $S_i \neq \emptyset$ and $S_i \cap S_j = \emptyset$ for all i, j, with $i \neq j$. The partitions $\{\mathcal{S}\}$ and $\Delta = \{\{i\} \mid i \in \mathcal{S}\}$ are the trivial partitions. Let $\mathcal{P}_1 = \{S_1, \ldots, S_N\}$ be a partition of $\mathcal{S}$ and $\mathcal{P}_2 = \{T_1, \ldots, T_M\}$, in turn, a partition of $\mathcal{P}_1$. The composition $\mathcal{P}_1 \circ \mathcal{P}_2$ of the partitions $\mathcal{P}_1$ and $\mathcal{P}_2$ is a partition of $\mathcal{S}$, given by $\mathcal{P}_1 \circ \mathcal{P}_2 = \{U_1, \ldots, U_M\}$, where $U_i = \bigcup_{C\in T_i} C$.

2. Markovian models

In this section we introduce the Markovian models studied in this paper: discontinuous Markov reward chains as generalisations of standard Markov reward chains where infinitely many transitions can be performed in a finite amount of time; Markov reward chains with fast transitions as Markov reward chains parametrised by a real variable τ; and Markov reward chains with silent transitions as equivalence classes of Markov reward chains with fast transitions with the same structure and unspecified 'speeds' of the fast transitions. The fast transitions explicitly model stochastic behavior, while the silent transitions model non-deterministic internal steps.

[1] The method is called total $\tau_\sim$-reduction in [10,11], since there more $\tau_\sim$-reduction methods are considered.

$$\Pi = \begin{pmatrix} \overline{\Pi}_1 & \dots & \overline{\Pi}_M & \mathbf{0} \\ \Pi_1 & \dots & \mathbf{0} & \mathbf{0} \\ \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \dots & \Pi_M & \mathbf{0} \end{pmatrix} \qquad L = \begin{pmatrix} \mu_1 & \dots & \mathbf{0} & \mathbf{0} \\ \vdots & \ddots & \vdots & \vdots \\ \mathbf{0} & \dots & \mu_M & \mathbf{0} \end{pmatrix} \qquad R = \begin{pmatrix} \delta_1 & \dots & \delta_M \\ \mathbf{1} & \dots & \mathbf{0} \\ \vdots & \ddots & \vdots \\ \mathbf{0} & \dots & \mathbf{1} \end{pmatrix}$$

Fig. 3. The ergodic form of Π.

2.1. Discontinuous Markov reward chains

In the standard theory (cf. [27,28,1]) Markov chains are assumed to be stochastically continuous. This means that the probability is 1 for the process occupying the same state at time t as at time 0, when $t \to 0$. As we include instantaneous transitions in our theory [13], this requirement must be dropped. Therefore, we work in the more general setting of discontinuous Markov chains originating from [12].

A discontinuous Markov reward chain is a time-homogeneous finite-state stochastic process with an associated (state) reward structure that satisfies the Markov property. It is completely determined by: (1) a stochastic initial probability row vector that gives the starting probabilities of the process for each state, (2) a transition matrix function $P : \mathbb{R}^+ \to \mathbb{R}^{n\times n}$ that defines the stochastic behavior of the transitions at time $t > 0$, and (3) a state reward rate vector that associates a number to each state representing the gain of the process while spending time in the state. The transition matrix function gives a stochastic matrix $P(t)$ at any time $t > 0$, and has the property $P(t + s) = P(t) \cdot P(s)$ [27,28]. It has a convenient characterisation independent of time [13,29], which allows for the following equivalent definition.

Definition 2. A discontinuous Markov reward chain $\mathsf{D} = (\sigma, \Pi, Q, \rho)$, where σ is a stochastic initial probability row vector, ρ is a state reward vector and $\Pi \in \mathbb{R}^{n\times n}$ and $Q \in \mathbb{R}^{n\times n}$ satisfy the following six conditions: (1) $\Pi \geqslant 0$, (2) $\Pi \cdot \mathbf{1} = \mathbf{1}$, (3) $\Pi^2 = \Pi$, (4) $\Pi Q = Q\Pi = Q$, (5) $Q \cdot \mathbf{1} = \mathbf{0}$, and (6) $Q + c\Pi \geqslant 0$, for some $c \geqslant 0$. The matrix function $P(t) = \Pi e^{Qt}$ is the transition matrix of D.

We note that the transition matrix uniquely determines the matrices Π and Q as given in Definition 2. It is continuous at zero if and only if $\Pi = I$. In this case, Q is a standard generator matrix [13,9]. Otherwise, the matrix Q might contain negative non-diagonal entries. We note that, unlike for standard Markov reward chains, a meaningful graphical representation of discontinuous Markov reward chains when $\Pi \neq I$ is not common. The intuition behind the matrix Π is that $\Pi[i, j]$ denotes the probability that a process occupies two states via an instantaneous transition. Therefore, in case of no instantaneous transitions, i.e., when $\Pi = I$, we get a standard (stochastically continuous) Markov reward chain denoted by $\mathsf{M} = (\sigma, Q, \rho)$.

For every discontinuous Markov reward chain $\mathsf{D} = (\sigma, \Pi, Q, \rho)$, Π gets an 'ergodic' form after a suitable renumbering of states [13] as depicted in Fig. 3. Here, for all $1 \leqslant k \leqslant M$, $\Pi_k = \mathbf{1}\cdot\mu_k$ and $\overline{\Pi}_k = \delta_k\cdot\mu_k$ for a row vector $\mu_k > 0$ such that $\mu_k\cdot\mathbf{1} = 1$ and a vector $\delta_k \geqslant 0$ such that $\sum_{k=1}^{M} \delta_k = \mathbf{1}$. Then the pair of matrices (L, R) depicted above forms a canonical product decomposition of Π (cf. Section 4.1), which is needed for the definition of the reduction-based method of aggregation.

The new numbering induces a partition $\mathcal{E} = \{E_1, \dots, E_M, T\}$ of the state space $\mathcal{S} = \{1, \dots, n\}$, where $E_1, \dots, E_M$ are the ergodic classes, determined by $\Pi_1, \dots, \Pi_M$, respectively, and T is the class of transient states, determined by any $\overline{\Pi}_i$, $1 \leqslant i \leqslant M$. The partition $\mathcal{E}$ is called the ergodic partition. For every ergodic class E_k, the vector μ_k is the vector of ergodic probabilities. If an ergodic class E_k contains exactly one state, then $\mu_k = (1)$ and the state is called regular. The vector δ_k contains the trapping probabilities from transient states to the ergodic class E_k.

We next discuss the behaviour of a discontinuous Markov reward chain $\mathsf{D} = (\sigma, \Pi, Q, \rho)$. It starts in a state with a probability given by the initial probability vector σ. In an ergodic class with multiple states the process spends a non-zero amount of time switching rapidly (infinitely many times) among the states. The probability that it is found in a specific state of the class is given by the vector of ergodic probabilities. The time the process spends in the class is exponentially distributed and determined by the matrix Q. In an ergodic class with a single state the row of Q corresponding to that state has the form of a row in a generator matrix, and $Q[i, j]$ for $i \neq j$ is interpreted as the rate from i to j. In a transient state the process spends no time (with probability one) and it immediately becomes trapped in some ergodic class. The process in $i \in T$ can be trapped in E_k if and only if the trapping probability $\delta_k[i] > 0$.

The expected reward (rate) at time $t > 0$, notation $R(t)$, is obtained as $R(t) = \sigma P(t)\rho$. It is required in the calculation of the expected accumulated reward up to time t, given by $\int_0^t R(s)\mathrm{d}s$. We have that the expected reward remains unchanged if the reward vector ρ is replaced by $\Pi\rho$. To see this, we use that $P(t) = P(t)\Pi$ (cf. [13,11]), so $\sigma P(t)\Pi\rho = \sigma P(t)\rho = R(t)$. Intuitively, the reward in a transient state can be replaced by the sum of the rewards of the ergodic states that it can get trapped in as the process gains no reward while transiting through transient states. The reward of an ergodic state is the sum of the rewards of all states inside its ergodic class weighted according to their ergodic probabilities. This alternative representation of the reward vector alleviates the presentation of some aggregation methods in later sections.

2.2. Markov reward chains with fast transitions

A Markov reward chain with fast transitions is obtained by adding parametrised, so-called fast, transitions to a standard Markov reward chain. The remaining standard transitions are referred to as slow. The behavior of a Markov reward chain

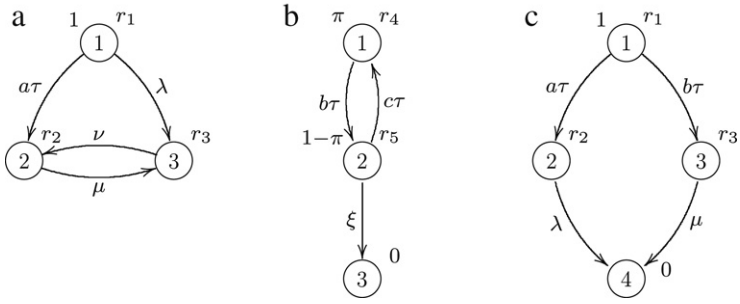


Fig. 4. Markov reward chains with fast transitions.

with fast transitions is determined by two generator matrices S and F, which represent the rates of the slow transitions and the rates (called speeds) of the fast transitions, respectively.

Definition 3. A Markov reward chain with fast transitions $\mathsf{F} = (\sigma, S, F, \rho)$ is a function assigning to each $\tau > 0$, the parametrised Markov reward chain

$$\mathsf{M}_\tau = (\sigma, S + \tau F, \rho)$$

where $\sigma \in \mathbb{R}^{1\times n}$ is an initial probability vector, $S, F \in \mathbb{R}^{n\times n}$ are two generator matrices, and $\rho \in \mathbb{R}^{n\times 1}$ is the reward vector.

By taking the limit when $\tau \to \infty$, fast transitions become instantaneous. Then, a Markov reward chain with fast transitions behaves as a discontinuous Markov reward chain [13,9–11].

Definition 4. Let $\mathsf{F} = (\sigma, S, F, \rho)$ be a Markov reward chain with fast transitions. The discontinuous Markov reward chain $\mathsf{D} = (\sigma, \Pi, Q, \Pi\rho)$ is the limit of F, where the matrix Π is the so-called ergodic projection at zero of F, that is $\Pi = \lim_{t\to\infty} \mathrm{e}^{Ft}$, and $Q = \Pi S \Pi$. We write $\mathsf{F} \to_\infty \mathsf{D}$.

The initial probability vector is not affected by the limit construction. We will later motivate the choice of using the reward vector $\Pi\rho$ instead of just ρ. In addition, we define the ergodic partition of a Markov reward chain with fast transitions to be the ergodic partition of its limit discontinuous Markov reward chain.

To illustrate the relation between the Markov reward chains with fast transitions and discontinuous Markov reward chains we give an example.

Example 5. We depict Markov reward chains with fast transitions as in Fig. 4. The initial probabilities are depicted left above, and the reward rates right above each state. Here, a, b, and c are speeds, whereas λ, μ, ν, and ξ are rates of slow transitions. As in the definition, τ denotes the real parameter.

The limit of the Markov reward chain with fast transitions in Fig. 4(a) is given by the discontinuous Markov reward chain $\mathsf{D}_1 = (\sigma_1, \Pi_1, Q_1, \rho_1)$ with:

$$\sigma_1 = \begin{pmatrix}1 & 0 & 0\end{pmatrix} \quad \Pi_1 = \begin{pmatrix}0 & 1 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix} \quad Q_1 = \begin{pmatrix}0 & -\mu & \mu\\ 0 & -\mu & \mu\\ 0 & \nu & -\nu\end{pmatrix} \quad \rho_1 = \begin{pmatrix}r_2\\ r_2\\ r_3\end{pmatrix}.$$

State 1 is transient, whereas states 2 and 3 are regular.

The limit of the Markov reward chain with fast transitions in Fig. 4(b) is given by $\mathsf{D}_2 = (\sigma_2, \Pi_2, Q_2, \rho_2)$ with:

$$\sigma_2 = \begin{pmatrix}\pi & 1-\pi & 0\end{pmatrix} \Pi_2 = \begin{pmatrix}p & q & 0\\ p & q & 0\\ 0 & 0 & 1\end{pmatrix} \quad Q_2 = \begin{pmatrix}-pq\xi & -q^2\xi & q\xi\\ -pq\xi & -q^2\xi & q\xi\\ 0 & 0 & 0\end{pmatrix} \quad \rho_2 = \begin{pmatrix}pr_4 + qr_5\\ pr_4 + qr_5\\ 0\end{pmatrix},$$

where $p = \frac{c}{b+c}$ and $q = \frac{b}{b+c}$. States 1 and 2 in Fig. 4(b) form an ergodic class and state 3 is regular.

Finally, the limit of the process depicted in Fig. 4(c) is given by the discontinuous Markov reward chain $\mathsf{D}_3 = (\sigma_3, \Pi_3, Q_3, \rho_3)$ with:

$$\sigma_3 = \begin{pmatrix}1 & 0 & 0 & 0\end{pmatrix} \quad \Pi_3 = \begin{pmatrix}0 & \frac{a}{a+b} & \frac{b}{a+b} & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{pmatrix} \quad Q_3 = \begin{pmatrix}0 & 0 & 0 & 0\\ 0 & -\lambda & 0 & \lambda\\ 0 & 0 & -\mu & \mu\\ 0 & 0 & 0 & 0\end{pmatrix} \quad \rho_3 = \begin{pmatrix}0\\ r_2\\ r_3\\ 0\end{pmatrix}.$$

State 1 in Fig. 4(c) is transient, whereas states 2, 3, and 4 are regular.

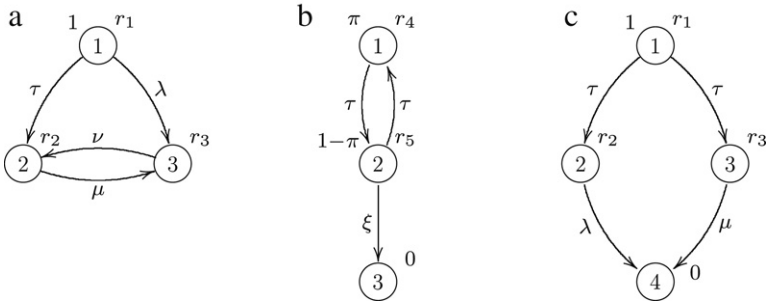


Fig. 5. Markov reward chains with silent transitions.

2.3. Markov reward chains with silent transitions

We define a Markov reward chain with silent transitions as a Markov reward chain with fast transitions in which the speeds of the fast transitions are left unspecified. To abstract away from the speeds of the fast transitions we introduce a suitable equivalence relation on Markov reward chains with fast transitions that is induced by the following equivalence relation of matrices.

Definition 6. Two matrices $A, B \in \mathbb{R}^{n\times n}$ have the same shape (also called grammar), notation $A \sim B$, if and only if they have zeros on the same positions. That is,

$$A \sim B \iff (\forall i, j)\,(A[i,j] = 0 \iff B[i,j] = 0)\,.$$

It is obvious that $\sim$ is an equivalence on matrices of the same order. The abstraction from speeds is achieved by identifying generator matrices of fast transitions with the same shape. Thus, silent transitions are modelled by equivalence classes of $\sim$.

Definition 7. A Markov reward chain with silent transitions S is a quadruple $\mathsf{S} = (\sigma, S, \mathcal{F}, \rho)$ where $\mathcal{F}$ is an equivalence class of $\sim$ and, for every $F \in \mathcal{F}$, $\mathsf{F} = (\sigma, S, F, \rho)$ is a Markov reward chain with fast transitions.

We write $\mathsf{F} \in \mathsf{S}$ if $\mathsf{S} = (\sigma, S, \mathcal{F}, \rho)$, and $\mathsf{F} = (\sigma, S, F, \rho)$ with $F \in \mathcal{F}$. Furthermore, we lift the relation $\sim$ to Markov reward chains with fast transitions and write $\mathsf{F} \sim \mathsf{F}'$ if $\mathsf{F}, \mathsf{F}' \in \mathsf{S}$. The notion of an ergodic partition is speed independent, i.e., if $\mathsf{F} \sim \mathsf{F}'$, then they have the same ergodic partition, since the ergodic partition depends only on the existence of fast transitions, but not on the actual speeds. Hence we can define the ergodic partition of a Markov reward chain with silent transitions S as the ergodic partition of any Markov reward chain with fast transitions F such that $\mathsf{F} \in \mathsf{S}$.

Example 8. We depict Markov reward chains with silent transitions as in Fig. 5, i.e., by omitting the speeds of the fast transitions. The depicted Markov reward chains with silent transitions are induced by the Markov reward chains with fast transitions in Fig. 4.

In Fig. 5, τ can be understood as a label of internal action transitions, as it is common in transition system modeling and process algebra [30,31]. In this way we formalise the notion of performance analysis for Markov reward chains with non-deterministic internal steps.

3. Aggregation by lumping

In this section we present lumping methods for the Markovian models of the previous section originally studied in [9–11]. First, we generalise ordinary lumping of [18] to discontinuous Markov reward chains. Then, we define τ-lumping for Markov reward chains with fast transitions based on ordinary lumping of discontinuous Markov reward chains. Finally, we lift the τ-lumping to $\tau_\sim$-lumping of Markov reward chains with silent transitions.

We define aggregation by lumping in terms of matrices. Every partition $\mathcal{P} = \{C_1, \ldots, C_N\}$ of $\mathcal{S} = \{1, \ldots, n\}$ can be associated with a so-called collector matrix $V \in \mathbb{R}^{n\times N}$ defined by $V[i,k] = 0$ if $i \notin C_k$, $V[i,k] = 1$ if $i \in C_k$, and vice versa. The k-th column of V has 1's for elements corresponding to states in C_k and has 0's otherwise. Note that $V \cdot \mathbf{1} = \mathbf{1}$. A distributor matrix $U \in \mathbb{R}^{N\times n}$ for $\mathcal{P}$ is defined as a matrix $U \geqslant 0$, such that $UV = I^N$. To satisfy these conditions, the elements of the k-th row of U, which correspond to states in the class C_k, sum up to one, whereas the other elements of the row are 0.

3.1. Ordinary lumping

An ordinary lumping is a partition of the state space of a discontinuous Markov reward chain into classes such that the states that are lumped together have equivalent behaviour for transiting to other classes and, additionally, have the same reward.

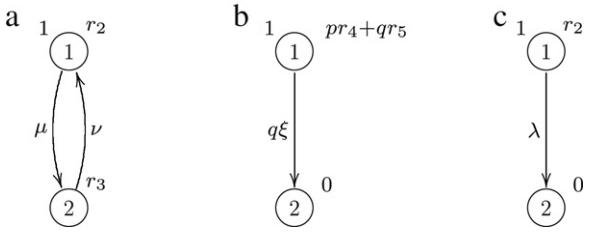


Fig. 6. Aggregated Markov reward chains with fast transitions of Fig. 4.

Definition 9. A partition $\mathcal{L}$ of $\{1, \ldots, n\}$ is an ordinary lumping, or lumping for short, of a discontinuous Markov reward chain $\mathsf{D} = (\sigma, \Pi, Q, \rho)$ if and only if the following hold: (1) $VU\Pi V = \Pi V$, (2) $VUQV = QV$, and (3) $VU\rho = \rho$, where V is the collector matrix and U is any distributor matrix for $\mathcal{L}$.

The lumping conditions only require that the rows of ΠV (respectively QV and ρ) that correspond to the states of the same partition class are equal. We have the following property.

Proposition 10 (*[9–11]*). *Let* $\mathsf{D} = (\sigma, \Pi, Q, \rho)$ *be a discontinuous Markov reward chain and let* $\mathcal{L}$ *be its ordinary lumping. Define* (1) $\overline{\sigma} = \sigma V$, (2) $\overline{\Pi} = U\Pi V$, (3) $\overline{Q} = UQV$ *and* (4) $\overline{\rho} = U\rho$, *for the collector matrix* V *of* $\mathcal{L}$ *and any distributor* U. *Then* $\overline{\mathsf{D}} = (\overline{\sigma}, \overline{\Pi}, \overline{Q}, \overline{\rho})$ *is a discontinuous Markov reward chain. Moreover,* $\overline{P}(t) = UP(t)V$ *where* $\overline{P}(t)$ *and* $P(t)$ *are the transition matrices of* $\overline{\mathsf{D}}$ *and* D, *respectively.* □

Definition 11. If the conditions of Proposition 10 are satisfied, then $\mathsf{D} = (\sigma, \Pi, Q, \rho)$ lumps to $\overline{\mathsf{D}} = (\overline{\sigma}, \overline{\Pi}, \overline{Q}, \overline{\rho})$, called the lumped discontinuous Markov reward chain with respect to $\mathcal{L}$. We write $\mathsf{D} \xrightarrow{\mathcal{L}} \overline{\mathsf{D}}$.

It can readily be seen that neither the definition of a lumping, nor the definition of the lumped process depends on the choice of a distributor matrix U. For example, if $VUQV = QV$, then $VU'QV = VU'VUQV = VUQV = QV$, for any other distributor U'. In the continuous case, when $\Pi = I$ we have $\overline{\Pi} = I$, so $\overline{Q}$ is a generator matrix and our notion of ordinary lumping coincides with the standard definition [18,32]. The expected reward is preserved by ordinary lumping, since:

$$\overline{R}(t) = \sigma VUP(t)VU\rho = \sigma P(t)VU\rho = \sigma P(t)\rho = R(t).$$

Similarly, as in [18], one can show that other performance measures are also preserved by lumping. To illustrate the method we lump the discontinuous Markov reward chains of Example 5.

Example 12. It is directly checked that the lumping condition holds for the discontinuous Markov reward chains D_1 and D_2 of Example 5 for the same partition $\{\{1, 2\}, \{3\}\}$. The discontinuous Markov reward chain D_3 can be lumped only if $\lambda = \mu$ and $r_2 = r_3$ using the partition $\{\{1, 2, 3\}, \{4\}\}$. For the lumped version $\overline{\mathsf{D}}_1 = (\overline{\sigma}_1, \overline{\Pi}_1, \overline{Q}_1, \overline{\rho}_1)$ of D_1 we obtain:

$$V_1 = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \overline{\sigma}_1 = \begin{pmatrix} 1 & 0 \end{pmatrix} \quad \overline{\Pi}_1 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \overline{Q}_1 = \begin{pmatrix} -\mu & \mu \\ \nu & -\nu \end{pmatrix} \quad \overline{\rho}_1 = \begin{pmatrix} r_2 \\ r_3 \end{pmatrix}.$$

The lumped process $\overline{\mathsf{D}}_1$ is a standard Markov reward chain as $\overline{\Pi}_1$ is the identity matrix.

For the lumped version $\overline{\mathsf{D}}_2 = (\overline{\sigma}_2, \overline{\Pi}_2, \overline{Q}_2, \overline{\rho}_2)$ of D_2 we obtain:

$$V_2 = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \overline{\sigma}_2 = \begin{pmatrix} 1 & 0 \end{pmatrix} \overline{\Pi}_2 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \overline{Q}_2 = \begin{pmatrix} -q\xi & q\xi \\ 0 & 0 \end{pmatrix} \quad \overline{\rho}_2 = \begin{pmatrix} pr_4 + qr_5 \\ 0 \end{pmatrix},$$

where $p = \frac{c}{b+c}$ and $q = 1 - p$. Again, the result is a standard Markov reward chain.

If we assume that $\lambda = \mu$ and $r_2 = r_3$, then the lumped version of D_3 is $\overline{\mathsf{D}}_3 = (\overline{\sigma}_3, \overline{\Pi}_3, \overline{Q}_3, \overline{\rho}_3)$ with:

$$V_3 = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \overline{\sigma}_3 = \begin{pmatrix} 1 & 0 \end{pmatrix} \quad \overline{\Pi}_3 = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \quad \overline{Q}_3 = \begin{pmatrix} -\lambda & \lambda \\ 0 & 0 \end{pmatrix} \quad \overline{\rho}_3 = \begin{pmatrix} r_2 \\ 0 \end{pmatrix},$$

which is again a standard Markov reward chain.

The lumped processes are depicted in Fig. 6.

3.2. τ-lumping

The notion of τ-lumping is based on ordinary lumping for discontinuous Markov reward chains. The aim is that the limit of a τ-lumped Markov reward chain with fast transitions is an ordinary lumped version of the limit of the original Markov reward chain with fast transitions.

Definition 13. A partition $\mathcal{L}$ of the state space of a Markov reward chain with fast transitions F is called a τ-lumping, if it is an ordinary lumping of its limiting discontinuous Markov reward chain D with $\mathsf{F} \to_\infty \mathsf{D}$.

Note that since we defined the reward of the limit by $\Pi\rho$, a τ-lumping may identify states with different rewards.

Like for ordinary lumping, we define the τ-lumped process by multiplying σ, S, F and ρ with a collector matrix and a distributor matrix. However, unlike for ordinary lumping, not all distributors are allowed. In [9–11] a class of special distributors, called τ-distributors, is provided that yields a τ-lumped process.

Definition 14. Let $\mathsf{D} = (\sigma, \Pi, Q, \rho)$ be a discontinuous Markov reward chain. Let V be a collector corresponding to a partition of the state space of this chain. A matrix W is a τ-distributor for V if and only if (1) it is a distributor for V, (2) $\Pi VW\Pi = \Pi VW$, and (3) the entries of W corresponding to states in classes of transient states are positive.

A τ-distributor for a partition of a Markov reward chain with fast transitions is any τ-distributor for the same partition of its limiting discontinuous Markov reward chain.

Remark 15. An alternative, explicit definition of the τ-distributors can be found in [9–11]. We note here that the class of τ-distributors given by Definition 14 depends on two sets of parameters. Namely, after suitable renumbering, any τ-distributor W can be written as $W = \begin{pmatrix} W(\alpha) & \mathbf{0} \\ \mathbf{0} & W(\beta) \end{pmatrix}$, where $W(\alpha)$ is a distributor for the classes containing ergodic states and $W(\beta)$ is a distributor for the classes of transient states. As the notation suggests, the distributor $W(\alpha)$ depends on a set of parameters α and the distributor $W(\beta)$ is determined by a set of parameters β. To explicitly state this dependence we may write $W_{\alpha,\beta}$ for a τ-distributor depending on the parameter sets α and β. By the alternative definition of τ-distributors we can also establish the existence of a τ-distributor for any τ-lumping.

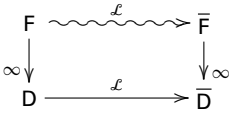
Having defined τ-distributors, we can now explicitly define a τ-lumped process.

Definition 16. Let $\mathsf{F} = (\sigma, S, F, \rho)$ and let $\mathcal{L}$ be a lumping with a collector matrix V, and a corresponding τ-distributor W. The τ-lumped Markov reward chain with fast transitions $\overline{\mathsf{F}} = (\overline{\sigma}, \overline{S}, \overline{F}, \overline{\rho})$ is defined as $\overline{\sigma} = \sigma V$, $\overline{S} = WSV$, $\overline{F} = WFV$, $\overline{\rho} = W\rho$. We say that $\mathsf{F}\tau$-lumps to $\overline{\mathsf{F}}$ with respect to W and write $\mathsf{F} \overset{\mathcal{L}}{\rightsquigarrow}_W \overline{\mathsf{F}}$. We write $\mathsf{F} \overset{\mathcal{L}}{\rightsquigarrow} \overline{\mathsf{F}}$ if $\mathsf{F} \overset{\mathcal{L}}{\rightsquigarrow}_W \overline{\mathsf{F}}$ for some τ-distributor W.

In general, when lumping F using a collector V and a distributor U, USV and UFV are not uniquely determined, i.e., they depend on the choice of the distributor. The restriction to τ-distributors does not change this. Subsequently, the τ-lumped process depends on the choice of the τ-distributor. In order to make the τ-distributor used explicit, we sometimes write $\mathsf{F} \overset{\mathcal{L}}{\rightsquigarrow}_{\alpha,\beta} \overline{\mathsf{F}}$ in order to emphasise the parameter sets such that $W = W_{\alpha,\beta}$.

The motivation for restricting to τ-distributors, despite that they do not ensure a unique τ-lumped process, is that all τ-lumped processes are equivalent in the limit. This is stated in the following proposition, which gives the precise connection of ordinary lumping and τ-lumping.

Proposition 17 (*[9–11]*). *The following diagram commutes*



that is, if $\mathsf{F} \overset{\mathcal{L}}{\rightsquigarrow} \overline{\mathsf{F}} \to_\infty \overline{\mathsf{D}}$ *and if* $\mathsf{F} \to_\infty \mathsf{D} \overset{\mathcal{L}}{\to} \overline{\mathsf{D}}'$, *then* $\overline{\mathsf{D}} = \overline{\mathsf{D}}'$, *for* F *and* $\overline{\mathsf{F}}$ *Markov reward chains with fast transitions, and* D, $\overline{\mathsf{D}}$, *and* $\overline{\mathsf{D}}'$ *discontinuous Markov reward chains.* □

Moreover, the τ-lumped processes that originate from the same Markov reward chain with fast transitions become exactly the same, once all fast transitions are eliminated [10,11].

Example 18. The τ-distributors (cf. [10,11] for the explicit form of the τ-distributors) for the Markov reward chains with fast transitions depicted in Fig. 4(a), Fig. 4(b), and Fig. 4(c) are

$$W_1 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad W_2 = \begin{pmatrix} \dfrac{c}{b+c} & \dfrac{b}{b+c} & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad W_3 = \begin{pmatrix} 0 & \alpha & 1-\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

for some $0 \leq \alpha \leq 1$, respectively. Again we assume that $\lambda = \mu$ and $r_2 = r_3$ for the Markov reward chain with fast transitions in Fig. 4(c) to be τ-lumpable. It is directly checked that the τ-lumped processes correspond to the lumped Markov reward chains from Example 12 depicted in Fig. 6 as expected by Proposition 17.

3.3. $\tau_\sim$-lumping

We lift τ-lumping to equivalence classes of $\sim$ to obtain $\tau_\sim$-*lumping* for Markov reward chains with silent transitions. Intuitively, a partition is a $\tau_\sim$-lumping of S, if it is a τ-lumping for every $\mathsf{F} \in \mathsf{S}$ and, moreover, the limit of the τ-lumped

process of F does not depend on the parameters chosen for the τ-distributor. Recall that the parameter set α affects ergodic states, whereas the parameter set β affects only transient states.

Definition 19. Let S be a Markov reward chain with silent transitions and let $\mathcal{L}$ be its partition. Then $\mathcal{L}$ is a $\tau_\sim$-lumping if and only if it is a τ-lumping for every Markov reward chain with fast transitions $\mathsf{F} \in \mathsf{S}$ and, moreover, for every $\mathsf{F}, \mathsf{F}' \in \mathsf{S}$ if $\mathsf{F} \overset{\mathcal{L}}{\rightsquigarrow}_{\alpha,\beta} \overline{\mathsf{F}}$ and $\mathsf{F}' \overset{\mathcal{L}}{\rightsquigarrow}_{\alpha',\beta} \overline{\mathsf{F}}'$, then $\overline{\mathsf{F}} \sim \overline{\mathsf{F}}'$.

The motivation behind the use of the same parameter set β in Definition 19 is that there may be slow transitions originating from transient states which will depend on β in the lumped process. If we do not restrict to the same parameter set β, then $\tau_\sim$-lumpings will only exist in rare cases in which transient states have no slow transitions. We refer to [10,11] for details.

Now we can define a $\tau_\sim$-lumped process which is unique for a given $\tau_\sim$-lumping $\mathcal{L}$ and a parameter set β.

Definition 20. Let S be a Markov reward chain with silent transitions and $\mathcal{L}$ its $\tau_\sim$-lumping. Let $\mathsf{F} \in \mathsf{S}$ be such that $\mathsf{F} \overset{\mathcal{L}}{\rightsquigarrow}_{\alpha,\beta} \overline{\mathsf{F}}$ and let $\overline{\mathsf{S}}$ be the Markov reward chain with silent transitions with $\overline{\mathsf{F}} \in \overline{\mathsf{S}}$. Then $\mathsf{S}\tau_\sim$-lumps to $\overline{\mathsf{S}}$, with respect to $\mathcal{L}$ and β, notation $\mathsf{S} \overset{\mathcal{L}}{\rightsquigarrow}_\beta \overline{\mathsf{S}}$. We write $\mathsf{S} \overset{\mathcal{L}}{\rightsquigarrow} \overline{\mathsf{S}}$ if $\mathsf{S} \overset{\mathcal{L}}{\rightsquigarrow}_\beta \overline{\mathsf{S}}$ for some parameter set β.

Example 21. From the lumped and τ-lumped versions of Example 12 and Example 18, respectively, we conclude that the Markov reward chain with silent transitions depicted in Fig. 5(a) can be $\tau_\sim$-lumped to the Markov reward chain $\overline{\mathsf{D}}_1$ of Example 12 depicted in Fig. 6(a) as every representative Markov reward chain with fast transitions lumps to this process. The Markov reward chain with silent transitions depicted in Fig. 5(b) cannot be $\tau_\sim$-lumped as the τ-lumped versions of the representative Markov reward chains with fast transitions always depend on the parameters b and c and no further lumping is possible. Finally, if we assume that $\lambda = \mu$ and $r_2 = r_3$ for the Markov reward chain with silent transitions depicted in Fig. 5(c), then it can be $\tau_\sim$-lumped to the Markov reward chain $\overline{\mathsf{D}}_3$ of Example 12 depicted in Fig. 6(c).

4. Aggregation by reduction

Reduction is a specific aggregation method for transforming a discontinuous Markov chain into a standard Markov chain, originally studied in [21,22,13]. Extended to reward processes, the method reduces a discontinuous Markov reward chain to a Markov reward chain by eliminating instantaneous states, while retaining the behaviour of the regular states. In the same spirit, we define reduction methods that reduce Markov reward chains with fast and silent transitions to Markov reward chains following [10,11], called τ-reduction and $\tau_\sim$-reduction, respectively.

4.1. Reduction

The reduction-based aggregation method masks the stochastic discontinuity of a discontinuous Markov reward chain and transforms it into a Markov reward chain [21,13,10,11]. The underlying idea is to abstract away from the behaviour of individual states in an ergodic class. The method is based on the notion of a canonical product decomposition.

Definition 22. Let $\mathsf{D} = (\sigma, \Pi, Q, \rho)$ and assume that $\mathrm{rank}(\Pi) = M$, i.e., that there are M ergodic classes. A canonical product decomposition of Π is a pair of matrices (L, R) with $L \in \mathbb{R}^{M\times n}$ and $R \in \mathbb{R}^{n\times M}$ such that $L \geqslant 0$, $R \geqslant 0$, $\mathrm{rank}(L) = \mathrm{rank}(R) = M$, $L \cdot \mathbf{1} = \mathbf{1}$, and $\Pi = RL$.

A canonical product decomposition always exists and it can be constructed from the ergodic form of Π (see Fig. 3). Moreover, it can be shown that any other canonical product decomposition is permutation equivalent to this one. Since a canonical product decomposition (L, R) of Π is a full-rank decomposition, and since Π is idempotent, we also have that $LR = I^M$. Thus, we have $L\Pi = LRL = L$ and $\Pi R = RLR = R$. Next, we present the reduction method.

Definition 23. For a discontinuous Markov reward chain $\mathsf{D} = (\sigma, \Pi, Q, \rho)$, the reduced Markov reward chain $\mathsf{M} = (\overline{\sigma}, \overline{Q}, \overline{\rho})$ is given by $\overline{\sigma} = \sigma R$, $\overline{Q} = LQR$ and $\overline{\rho} = L\rho$, where (L, R) is a canonical product decomposition of Π. We write $\mathsf{D} \rightarrow_r \mathsf{M}$.

If $\overline{P}(t)$ and $P(t)$ are the transition matrices of the reduced and the original chain, respectively, then one can show that $\overline{P}(t) = LP(t)R$, see [13,22].

The reduced process is unique up to a permutation of the states, since the canonical product decomposition is. The states of the reduced process are given by the ergodic classes of the original process, while the transient states are 'ignored'. Intuitively, the transient states are split probabilistically between the ergodic classes according to their trapping probabilities. In case a transient state is also an initial state, its initial probability is split according to its trapping probabilities. The reward rate is calculated as the sum of the individual reward rates of the states of the ergodic class weighted by their ergodic probabilities. Like lumping, the reduction also preserves the expected reward rate at time t:

$$\overline{R}(t) = \sigma RLP(t)RL\rho = \sigma \Pi P(t) \Pi \rho = \sigma P(t)\rho = R(t).$$

In case the original process has no stochastic discontinuity, i.e., $\Pi = I$, the reduced process is equal to the original.

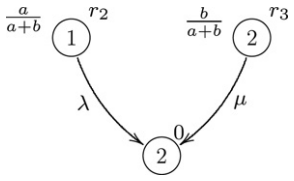


Fig. 7. τ-reduced Markov reward chain with fast transitions of Fig. 4(c).

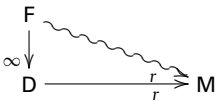
4.2. τ-reduction

We now define a reduction-based aggregation method called τ-reduction. It aggregates a Markov reward chain with fast transitions to an asymptotically equivalent Markov reward chain.

Definition 24. A Markov reward chain with fast transitions $\mathsf{F} = (\sigma, S, F, \rho)$ τ-reduces to the Markov reward chain $\mathsf{M} = (\overline{\sigma}, \overline{Q}, \overline{\rho})$, given by (1) $\overline{\sigma} = \sigma R$, (2) $\overline{Q} = LSR$, and (3) $\overline{\rho} = L\rho$, where $\mathsf{F} \to_\infty (\sigma, \Pi, Q, \Pi\rho)$ and (L, R) is a canonical product decomposition of Π. When F τ-reduces to M, we write $\mathsf{F} \leadsto_r \mathsf{M}$.

The following simple property relates τ-reduction to reduction. It holds since $LQR = L\Pi S\Pi R = LSR$ and $L\Pi\rho = L\rho$.

Proposition 25 (*[10,11]*). *The following diagram commutes*



that is, if $\mathsf{F} \leadsto_r \mathsf{M}$ *and* $\mathsf{F} \to_\infty \mathsf{D} \to_r \mathsf{M}'$, *then* $\mathsf{M} = \mathsf{M}'$, *for* F *a Markov reward chain with fast transitions,* D *a discontinuous Markov reward chain and* M *and* M' *(continuous) Markov reward chains.* □

Example 26. The aggregation by τ-reduction reduces the Markov reward chains with fast transitions of Fig. 4(a) and Fig. 4(b) to the Markov reward chains $\overline{\mathsf{D}}_1$ and $\overline{\mathsf{D}}_2$ of Example 12, respectively, as in the case of τ-lumping. However, by using τ-reduction the Markov reward chain with fast transitions depicted in Fig. 4(c) can be directly aggregated to a Markov reward chain without any assumptions. The canonical decompositions for the Markov reward chains with fast transitions depicted in Fig. 4(a), (b), and (c) are:

$$L_1 = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad R_1 = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \qquad L_2 = \begin{pmatrix} \frac{c}{b+c} & \frac{b}{b+c} & 0 \\ 0 & 0 & 1 \end{pmatrix} \qquad R_2 = \begin{pmatrix} 1 & 0 \\ 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$L_3 = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \qquad R_3 = \begin{pmatrix} \frac{a}{a+b} & \frac{b}{a+b} & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

The Markov reward chain with fast transitions of Fig. 4(c) reduces to the Markov reward chain depicted in Fig. 7. Note that the initial probability vector is split according to the transient probabilities.

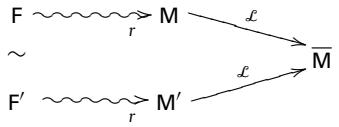
4.3. $\tau_\sim$-reduction

By combining τ-reduction with ordinary lumping of Markov reward chains, we can eliminate the effect of the speeds and obtain a reduction-like aggregation method for Markov reward chains with silent transitions. Here, we refer to this method as $\tau_\sim$-reduction.

One could define a reduction-based method for Markov reward chains with silent transitions by saying that a Markov reward chain with silent transitions S reduces to a Markov reward chain M if all Markov reward chains with fast transitions $\mathsf{F} \in \mathsf{S}$ will τ-reduce to M. However, such is not an efficient reduction method as it is applicable only in the few special cases when all Markov reward chains with fast transitions in a $\sim$ -equivalence class τ-reduce to the same Markov reward chain [10,11]. For this reason we combine τ-reduction and lumping.

Similarly as for $\tau_\sim$-lumping, the result of the $\tau_\sim$-reduction should not depend on the representative Markov reward chain with fast transitions. Therefore, a Markov reward chain with silent transitions can be $\tau_\sim$-reduced if all Markov reward chains with fast transitions in its equivalence class τ-reduce to Markov reward chains that can be ordinary lumped to the same

Markov reward chain, as depicted below:



Definition 27. Let S be a Markov reward chain with silent transitions. Let $\mathcal{E} = \{E_1, \ldots, E_M, T\}$ be its ergodic partition, and $\mathcal{L}$ a partition of $\{E_1, \ldots, E_M\}$. Then S can be $\tau_\sim$-reduced according to $\mathcal{L}$ if and only if there exists a Markov reward chain $\overline{\mathsf{M}}$, such that for every $\mathsf{F} \in \mathsf{S}$, we have that $\mathsf{F} \rightsquigarrow_r \mathsf{M} \xrightarrow{\mathcal{L}} \overline{\mathsf{M}}$ for some Markov reward chain M.

In the above situation, we write $\mathsf{S} \overset{\mathcal{L}}{\rightsquigarrow}_r \overline{\mathsf{M}}$ and $\mathsf{S} \rightsquigarrow_r \overline{\mathsf{M}}$ if a partition $\mathcal{L}$ exists such that $\mathsf{S} \overset{\mathcal{L}}{\rightsquigarrow}_r \overline{\mathsf{M}}$. We note that both $\tau_\sim$-lumping and $\tau_\sim$-reduction produce the same process when all silent transitions are eliminated, cf. [10,11] for details.

Example 28. The Markov reward chain with fast transitions depicted in Fig. 4(a) is $\tau_\sim$-reduced to the $\tau_\sim$-lumped version of the process from Fig. 6(a). As in the case of $\tau_\sim$-lumping above, there is no $\tau_\sim$-reduced process of the Markov reward chain with fast transitions in Fig. 4(b), as the τ-reduced process depends on the parameters and no further aggregation is possible. To obtain the $\tau_\sim$-reduced version of the Markov reward chain with fast transitions depicted in Fig. 4(c) we have to assume that $\lambda = \mu$ and $r_2 = r_3$ in order to lump the Markov reward chain from Fig. 7.

5. Relational properties

We investigate the relational properties of the lumping-based aggregation methods. For ordinary lumping, the combination of transitivity and strict confluence ensures that iterative application yields a uniquely determined process. In the case of τ-lumping, in view of Proposition 17, only the limit of the final reduced process is uniquely determined, unless the final process contains no fast transitions. Similarly, for $\tau_\sim$-lumping the reduced process is uniquely determined only if it does not contain any silent transitions.

There is no need to investigate the relational properties of the reduction-based methods, since they act in one step (no iteration is possible), in a unique way, between different types of models.

First, we investigate the properties of the relation $\geqslant$ on discontinuous Markov reward chains defined by

$$\mathsf{D}_1 \geqslant \mathsf{D}_2 \iff (\exists \mathcal{L})\, \mathsf{D}_1 \xrightarrow{\mathcal{L}} \mathsf{D}_2.$$

The above relation is clearly reflexive, since the trivial partition Δ is always an ordinary lumping, i.e., $\mathsf{D} \xrightarrow{\Delta} \mathsf{D}$ for any discontinuous Markov reward chain D. Transitivity enables replacement of repeated application of ordinary lumping by a single application using an ordinary lumping that is a composition of the individual lumpings.

Theorem 29. *Let* D *be a discontinuous Markov reward chain such that* $\mathsf{D} \xrightarrow{\mathcal{L}} \overline{\mathsf{D}}$ *and* $\overline{\mathsf{D}} \xrightarrow{\overline{\mathcal{L}}} \overline{\overline{\mathsf{D}}}$. *Then* $\mathsf{D} \xrightarrow{\mathcal{L} \circ \overline{\mathcal{L}}} \overline{\overline{\mathsf{D}}}$.

Proof. Let $\mathsf{D} = (\sigma, \Pi, Q, \rho)$, $\overline{\mathsf{D}} = (\overline{\sigma}, \overline{\Pi}, \overline{Q}, \overline{\rho})$, and $\overline{\overline{\mathsf{D}}} = (\overline{\overline{\sigma}}, \overline{\overline{\Pi}}, \overline{\overline{Q}}, \overline{\overline{\rho}})$. Let V and $\overline{V}$ denote the collector matrices for $\mathcal{L}$ and $\overline{\mathcal{L}}$, respectively. The collector matrix for $\mathcal{L} \circ \overline{\mathcal{L}}$ is $V\overline{V}$. The following lumping conditions hold: $VU\Pi V = \Pi V$, $VUQV = QV$ and $VU\rho = \rho$. Also $\overline{\Pi} = U\Pi V$, $\overline{Q} = UQV$ and $\overline{\rho} = U\rho$ for any distributor U for V. Similarly, it holds that: $\overline{V}\,\overline{U}\,\overline{\Pi}\,\overline{V} = \overline{\Pi}\,\overline{V}$, $\overline{V}\,\overline{U}\,\overline{Q}\,\overline{V} = \overline{Q}\,\overline{V}$ and $\overline{V}\,\overline{U}\,\overline{\rho} = \overline{\rho}$. Moreover $\overline{\overline{\Pi}} = \overline{U}\,\overline{\Pi}\,\overline{V}$, $\overline{\overline{Q}} = \overline{U}\,\overline{Q}\,\overline{V}$ and $\overline{\overline{\rho}} = \overline{U}\,\overline{\rho}$ for any distributor $\overline{U}$ for $\overline{V}$.

The iterative application of the ordinary lumping method can be replaced by the ordinary lumping given by the partition $\mathcal{L} \circ \overline{\mathcal{L}}$, that corresponds to the collector matrix $\overline{\overline{V}} = V\overline{V}$. A corresponding distributor is $\overline{\overline{U}} = \overline{U}\,U$, because $\overline{\overline{U}}\,\overline{\overline{V}} = \overline{U}UV\overline{V} = I$. That the partition is indeed an ordinary lumping follows from: $\overline{\overline{V}}\,\overline{\overline{U}}\,\Pi\,\overline{\overline{V}} = V\,\overline{V}\,\overline{U}\,U\Pi V\,\overline{V}\,V = V\,\overline{V}\,\overline{U}\,\overline{\Pi}\,\overline{V} = V\,\overline{\Pi}\,\overline{V} = VU\Pi V\,\overline{V} = \Pi V\,\overline{V} = \Pi\overline{\overline{V}}$. Similarly, one gets the condition for Q, and $\overline{\overline{V}}\,\overline{\overline{U}}\,\rho = V\,\overline{V}\,\overline{U}\,U\rho = V\,\overline{V}\,\overline{U}\,\overline{\rho} = V\,\overline{\rho} = VU\rho = \rho$. □

The relation $\geqslant$ on Markov reward chains with fast transitions, defined by

$$\mathsf{F}_1 \geqslant \mathsf{F}_2 \iff (\exists \mathcal{L})\, \mathsf{F}_1 \overset{\mathcal{L}}{\rightsquigarrow} \mathsf{F}_2$$

is a preorder as well. It is reflexive via the trivial lumping Δ. The following theorem shows the transitivity of the τ-lumping relation.

Theorem 30. *Let* F *be a Markov reward chain with fast transitions, such that* $\mathsf{F} \overset{\mathcal{L}}{\rightsquigarrow} \overline{\mathsf{F}}$ *and* $\overline{\mathsf{F}} \overset{\overline{\mathcal{L}}}{\rightsquigarrow} \overline{\overline{\mathsf{F}}}$. *Then* $\mathsf{F} \overset{\mathcal{L} \circ \overline{\mathcal{L}}}{\rightsquigarrow} \overline{\overline{\mathsf{F}}}$.

Proof. Let $\mathsf{F} = (\sigma, F, S, \rho)$ and $\overline{\mathsf{F}} = (\overline{\sigma}, \overline{F}, \overline{S}, \overline{\rho})$. Denote by V and $\overline{V}$ the collector matrices for $\mathcal{L}$ and $\overline{\mathcal{L}}$, respectively. The collector matrix for $\mathcal{L} \circ \overline{\mathcal{L}}$ is then $\overline{\overline{V}} = V\overline{V}$. Let W and $\overline{W}$ be the corresponding τ-distributors used for $\mathsf{F} \overset{\mathcal{L}}{\rightsquigarrow} \overline{\mathsf{F}}$ and

$\overline{\mathsf{F}} \overset{\overline{\mathcal{L}}}{\rightsquigarrow} \overline{\overline{\mathsf{F}}}$, respectively. Since τ-lumping is defined in terms of ordinary lumping, it is sufficient to show that $\overline{\overline{W}} = \overline{W}W$ is a τ-distributor. From Theorem 29 it is a distributor. The condition requiring positive entries corresponding to transient states that lump only with other transient states, can be checked using the explicit description of τ-distributors [11]. It remains to verify the third condition.

Let Π and $\overline{\Pi}$ be the ergodic projections of F and $\overline{F}$. Then, $\Pi V W \Pi = \Pi V W$ and $\overline{\Pi}\,\overline{V}\,\overline{W}\,\overline{\Pi} = \overline{\Pi}\,\overline{V}\,\overline{W}$. We have that:

$$\begin{aligned}\Pi\,\overline{\overline{V}}\,\overline{\overline{W}}\,\Pi &= \Pi V\,\overline{V}\,\overline{W}\,W\Pi = VW\Pi V\,\overline{V}\,\overline{W}W\Pi = V\,\overline{\Pi}\,\overline{V}\,\overline{W}\,W\Pi\\ &= V\,\overline{\Pi}\,\overline{V}\,\overline{W}\,\overline{\Pi}\,W\Pi = V\overline{\Pi}\,\overline{V}\,\overline{W}\,W\Pi VW\Pi = V\,\overline{\Pi}\,\overline{V}\,\overline{W}\,W\Pi VW\\ &= \text{(the same derivation steps backwards)}\\ &= \Pi V\,\overline{V}\,\overline{W}\,W = \Pi\,\overline{\overline{V}}\,\overline{\overline{W}}. \quad \square\end{aligned}$$

Similarly, $\tau_\sim$-lumping induces a preorder on Markov reward chains with silent transitions defined by

$$\mathsf{S}_1 \geqslant \mathsf{S}_2 \iff (\exists \mathcal{L})\, \mathsf{S}_1 \overset{\mathcal{L}}{\rightsquigarrow} \mathsf{S}_2.$$

Reflexivity again holds due to the trivial partition Δ, while transitivity is a direct consequence of Theorem 30 and the definition of $\tau_\sim$-lumping, Definition 19. Thus, we have the following theorem.

Theorem 31. *Let* S *be a Markov reward chain with silent transitions. Suppose* $\mathsf{S} \overset{\mathcal{L}}{\rightsquigarrow} \overline{\mathsf{S}}$ *and* $\overline{\mathsf{S}} \overset{\overline{\mathcal{L}}}{\rightsquigarrow} \overline{\overline{\mathsf{S}}}$. *Then* $\mathsf{S} \overset{\mathcal{L}\circ\overline{\mathcal{L}}}{\rightsquigarrow} \overline{\overline{\mathsf{S}}}$. $\square$

The lumping preorders also have the strict confluence property. In the case of lumping this means that if $\mathsf{P} \overset{\mathcal{L}_1}{\to} \mathsf{P}_1$ and $\mathsf{P} \overset{\mathcal{L}_2}{\to} \mathsf{P}_2$, then there exist two partitions $\overline{\mathcal{L}}_1$ and $\overline{\mathcal{L}}_2$ such that $\mathsf{P}_1 \overset{\mathcal{L}_1\circ\overline{\mathcal{L}}_1}{\to} \overline{\mathsf{P}}$ and $\mathsf{P}_2 \overset{\mathcal{L}_2\circ\overline{\mathcal{L}}_2}{\to} \overline{\mathsf{P}}$. One can prove the strict confluence property by adapting the proof for Markov reward chains, e.g., from [20].

6. Parallel composition and compositionality

In this section we define parallel composition for each of the models, and prove the compositionality results. The definitions are based on Kronecker products and sums, as for standard Markov reward chains [15,33]. The intuition behind this is that the Kronecker sum represents interleaving, whereas the Kronecker product represents synchronisation. Let us first recall the definition of Kronecker product and sum.

Definition 32. Let $A \in \mathbb{R}^{n_1\times n_2}$ and $B \in \mathbb{R}^{m_1\times m_2}$. The *Kronecker product* of A and B is a matrix $(A \otimes B) \in \mathbb{R}^{n_1m_1\times n_2m_2}$ defined by

$$(A \otimes B)[(i-1)m_1 + k, (j-1)m_2 + \ell] = A[i,j]B[k,\ell]$$

for $1 \leqslant i \leqslant n_1$, $1 \leqslant j \leqslant n_2$, $1 \leqslant k \leqslant m_1$ and $1 \leqslant \ell \leqslant m_2$.

The Kronecker sum of two square matrices $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{m\times m}$ is a matrix $(A \oplus B) \in \mathbb{R}^{nm\times nm}$ defined by $A \oplus B = A \otimes I^m + I^n \otimes B$.

Next, we list some basic properties of the Kronecker product and sum.

Proposition 33 (*[34]*). *The following equations hold:*

1. $(A \otimes B)(C \otimes D) = AC \otimes BD$,
2. $(A + B) \otimes (C + D) = A \otimes C + A \otimes D + B \otimes C + B \otimes D$,
3. $c(A \otimes B) = (cA \otimes B) = (A \otimes cB)$,
4. $c(A \oplus B) = (cA \oplus cB)$,
5. $\mathrm{e}^{A\oplus B} = \mathrm{e}^A \otimes \mathrm{e}^B$,
6. $\mathrm{rank}(A \otimes B) = \mathrm{rank}(A)\,\mathrm{rank}(B)$. $\square$

We also need the notion of a Kronecker product of two partitions. Let $\mathcal{L}_1$ and $\mathcal{L}_2$ be two partitions with corresponding collector matrices V_1 and V_2, respectively. Then $\mathcal{L}_1 \otimes \mathcal{L}_2$ denotes the partition corresponding to the collector matrix $V_1 \otimes V_2$.

6.1. Composing discontinuous Markov reward chains

First, we present the definition of parallel composition of discontinuous Markov reward chains. The intuition is that 'rates' interleave, and the probabilities of the instantaneous transitions synchronise, i.e., they are independent.

Definition 34. Let $\mathsf{D}_1 = (\sigma_1, \Pi_1, Q_1, \rho_1)$ and $\mathsf{D}_2 = (\sigma_2, \Pi_2, Q_2, \rho_2)$ be discontinuous Markov reward chains. Their parallel composition is defined as:

$$\mathsf{D}_1 \parallel \mathsf{D}_2 = (\sigma_1 \otimes \sigma_2, \Pi_1 \otimes \Pi_2, Q_1 \otimes \Pi_2 + \Pi_1 \otimes Q_2, \rho_1 \otimes \mathbf{1}^{|\rho_2|} + \mathbf{1}^{|\rho_1|} \otimes \rho_2).$$

The following theorem shows that the parallel composition of two discontinuous Markov reward chains is well defined.

Theorem 35. *Let* D_1 *and* D_2 *be two discontinuous Markov reward chains. Then* $\mathsf{D}_1 \parallel \mathsf{D}_2$ *is a discontinuous Markov reward chain.*

Proof. Let $\mathsf{D}_1 = (\sigma_1, \Pi_1, Q_1, \rho_1)$ and $\mathsf{D}_2 = (\sigma_2, \Pi_2, Q_2, \rho_2)$. The initial probability vector $\sigma_1 \otimes \sigma_2$ is a stochastic vector and the reward vector is well defined. Using Proposition 33(1)-(3), it is easy to check that the matrices $\Pi_1 \otimes \Pi_2$ and $Q_1 \otimes \Pi_2 + \Pi_1 \otimes Q_2$ satisfy the conditions of Definition 2, i.e., (1) $(\Pi_1 \otimes \Pi_2) \geq 0$, (2) $(\Pi_1 \otimes \Pi_2) \cdot \mathbf{1} = \mathbf{1}$, (3) $(\Pi_1 \otimes \Pi_2)^2 = \Pi_1 \otimes \Pi_2$, (4) $(\Pi_1 \otimes \Pi_2) \cdot (Q_1 \otimes \Pi_2 + \Pi_1 \otimes Q_2) = (Q_1 \otimes \Pi_2 + \Pi_1 \otimes Q_2) \cdot (\Pi_1 \otimes \Pi_2) = Q_1 \otimes \Pi_2 + \Pi_1 \otimes Q_2$, (5) $(Q_1 \otimes \Pi_2 + \Pi_1 \otimes Q_2) \cdot \mathbf{1} = \mathbf{0}$, and (6) $Q_1 \otimes \Pi_2 + \Pi_1 \otimes Q_2 + (c_1 + c_2) \cdot (\Pi_1 \otimes \Pi_2) = (Q_1 + c_1\Pi_1) \otimes \Pi_2 + \Pi_1 \otimes (Q_2 + c_2\Pi_2) \geq 0$ for $c_1, c_2 \geq 0$ such that $Q_1 + c_1\Pi_1 \geq 0$ and $Q_2 + c_2\Pi_2 \geq 0$. □

In the special case, when both discontinuous Markov reward chains are continuous, their parallel composition is again a Markov reward chain as defined in [15]. Moreover, the following property shows that the parallel composition of two discontinuous Markov reward chains has a transition matrix that is the Kronecker product of the individual transition matrices, corresponding to the intuition that the Kronecker product represents synchronisation. This justifies the definition of the parallel composition.

Theorem 36. *Let* D_1 *and* D_2 *be two discontinuous Markov reward chains with transition matrices* $P_1(t)$ *and* $P_2(t)$*, respectively. Then the transition matrix of* $\mathsf{D}_1 \parallel \mathsf{D}_2$ *is given by* $P_1(t) \otimes P_2(t)$.

Proof. Let $\mathsf{D}_1 = (\sigma_1, \Pi_1, Q_1, \rho_1)$ and $\mathsf{D}_2 = (\sigma_2, \Pi_2, Q_2, \rho_2)$. As the matrices $Q_1 \otimes \Pi_2$ and $\Pi_1 \otimes Q_2$ commute, and $P_i(t)\Pi_i = \Pi_i P_i(t) = P_i(t)$, we derive:

$$
\begin{aligned}
(\Pi_1 \otimes \Pi_2)\mathrm{e}^{(Q_1 \otimes \Pi_2 + \Pi_1 \otimes Q_2)t} &= (\Pi_1 \otimes \Pi_2)(\mathrm{e}^{(Q_1 \otimes \Pi_2)t}\mathrm{e}^{(\Pi_1 \otimes Q_2)t}) \\
&= (\Pi_1 \otimes \Pi_2)\left(\sum_{n=0}^{\infty}(Q_1 \otimes \Pi_2)^n t^n/n!\right)\left(\sum_{n=0}^{\infty}(\Pi_1 \otimes Q_2)^n t^n/n!\right) \\
&= (\Pi_1 \otimes \Pi_2)\left(I \otimes I + \sum_{n=1}^{\infty}(Q_1 \otimes \Pi_2)^n t^n/n!\right)\left(I \otimes I + \sum_{n=1}^{\infty}(\Pi_1 \otimes Q_2)^n t^n/n!\right) \\
&= (\Pi_1 \otimes \Pi_2)\left(I \otimes I + \sum_{n=1}^{\infty}(Q_1^n \otimes \Pi_2^n)t^n/n!\right)\left(I \otimes I + \sum_{n=1}^{\infty}(\Pi_1^n \otimes Q_2^n)t^n/n!\right) \\
&= (\Pi_1 \otimes \Pi_2)\left(I \otimes I + \sum_{n=1}^{\infty}(Q_1^n \otimes \Pi_2)t^n/n!\right)\left(I \otimes I + \sum_{n=1}^{\infty}(\Pi_1 \otimes Q_2^n)t^n/n!\right) \\
&= (\Pi_1 \otimes \Pi_2)(I \otimes I + \left(\sum_{n=1}^{\infty}Q_1^n t^n/n!\right) \otimes \Pi_2)\left(I \otimes I + \Pi_1 \otimes \sum_{n=1}^{\infty}Q_2^n t^n/n!\right) \\
&= (\Pi_1 \otimes \Pi_2)(I \otimes I + (\mathrm{e}^{Q_1 t} - I) \otimes \Pi_2)(I \otimes I + \Pi_1 \otimes (\mathrm{e}^{Q_2 t} - I)) \\
&= (\Pi_1 \otimes \Pi_2)(I \otimes I + \mathrm{e}^{Q_1 t} \otimes \Pi_2 - I \otimes \Pi_2)(I \otimes I + \Pi_1 \otimes \mathrm{e}^{Q_2 t} - \Pi_1 \otimes I) \\
&= (\Pi_1 \otimes \Pi_2 + P_1(t) \otimes \Pi_2 - \Pi_1 \otimes \Pi_2)(I \otimes I + \Pi_1 \otimes \mathrm{e}^{Q_2 t} - \Pi_1 \otimes I) \\
&= (P_1(t) \otimes \Pi_2)(I \otimes I + \Pi_1 \otimes \mathrm{e}^{Q_2 t} - \Pi_1 \otimes I) \\
&= (P_1(t) \otimes \Pi_2 + P_1(t) \otimes P_2(t) - P_1(t) \otimes \Pi_2) \\
&= P_1(t) \otimes P_2(t). \quad \square
\end{aligned}
$$

Remark 37. We can motivate Definition 34 also from another perspective. By the standard probabilistic (i.e., non-matrix) representation of discontinuous Markov reward chain the same notion can be obtained by the following analysis. Let $\{X(t) \mid t \geqslant 0\}$ and $\{Y(t) \mid t \geqslant 0\}$ be two discontinuous Markov reward chains defined on state spaces S_X and S_Y, respectively. Their parallel composition can be defined as the stochastic process $\{(X \parallel Y)(t) \mid t \geqslant 0\}$ with the state space $S_X \times S_Y$, such that $(X \parallel Y)(t) = (x, y)$ if and only if $X(t) = x$ and $Y(t) = y$. One can show that this process is again a discontinuous Markov reward chain with transition matrix equal to the Kronecker product of the transition matrices of $\{X(t) \mid t \geqslant 0\}$ and $\{Y(t) \mid t \geqslant 0\}$. It is known that the matrices Π and Q characterising a transition matrix $P(t)$ are obtained as $\Pi = \lim_{t \to 0} P(t)$ and $Q = \lim_{h \to 0} (P(h) - \Pi)/h$ [13]. Applying this result on the transition matrix of $\{(X \parallel Y)(t) \mid t \geqslant 0\}$ and using the definition of $(X \parallel Y)(0)$ we obtain the first three components of the quadruple from Definition 34. The reward vector for the parallel composition encodes the assumption that the reward rate in (x, y) is the sum of the reward rates in x and y.

It is easy to see that the expected reward of the parallel composition is the sum of the expected rewards of the components. Using Proposition 33(1) and (2) we have $(\sigma_1 \otimes \sigma_2)(P_1(t) \otimes P_2(t))(\rho_1 \otimes \mathbf{1} + \mathbf{1} \otimes \rho_2) = \sigma_1 P_1(t)\rho_1 \otimes \sigma_1 P_1(t)\mathbf{1} + \sigma_2 P_2(t)\mathbf{1} \otimes \sigma_2 P_2(t)\rho_2 = R_1(t) \otimes 1 + 1 \otimes R_2(t) = R_1(t) + R_2(t)$.

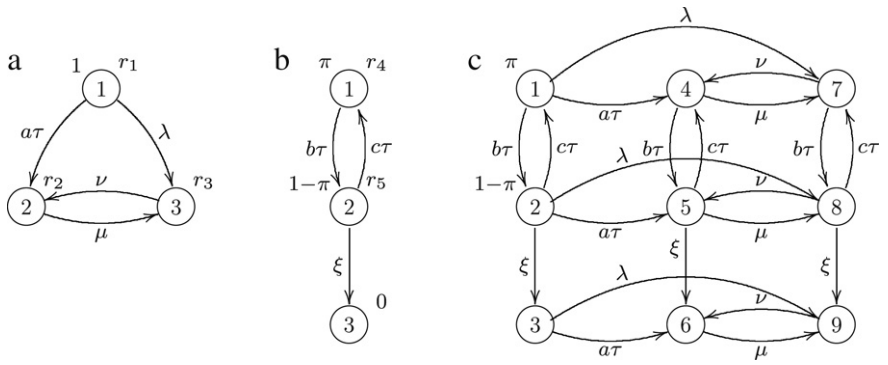


Fig. 8. Parallel composition of Markov reward chains with fast transitions.

The following theorem shows that both lumping and reduction are compositional with respect to the parallel composition of discontinuous Markov reward chains.

Theorem 38. *If* $\mathsf{D}_1 \overset{\mathcal{L}_1}{\to} \overline{\mathsf{D}}_1$ *and* $\mathsf{D}_2 \overset{\mathcal{L}_2}{\to} \overline{\mathsf{D}}_2$, *then* $\mathsf{D}_1 \parallel \mathsf{D}_2 \overset{\mathcal{L}_1 \otimes \mathcal{L}_2}{\to} \overline{\mathsf{D}}_1 \parallel \overline{\mathsf{D}}_2$. *Also, if* $\mathsf{D}_1 \to_r \mathsf{M}_1$ *and* $\mathsf{D}_2 \to_r \mathsf{M}_2$, *then* $\mathsf{D}_1 \parallel \mathsf{D}_2 \to_r \mathsf{M}_1 \parallel \mathsf{M}_2$.

Proof. Let $\mathsf{D}_1 = (\sigma_1, \Pi_1, \mathrm{Q}_1, \rho_1)$, $\overline{\mathsf{D}}_1 = (\overline{\sigma}_1, \overline{\Pi}_1, \overline{\mathrm{Q}}_1, \overline{\rho}_1)$, $\mathsf{D}_2 = (\sigma_2, \Pi_2, \mathrm{Q}_2, \rho_2)$, and $\overline{\mathsf{D}}_2 = (\overline{\sigma}_2, \overline{\Pi}_2, \overline{\mathrm{Q}}_2, \overline{\rho}_2)$. We first prove the compositionality of lumping. We show that $\mathcal{L}_1 \otimes \mathcal{L}_2$ is an ordinary lumping of

$$\mathsf{D}_1 \parallel \mathsf{D}_2 = (\sigma_1 \otimes \sigma_2, \Pi_1 \otimes \Pi_2, \mathrm{Q}_1 \otimes \Pi_2 + \Pi_1 \otimes \mathrm{Q}_2, \rho_1 \otimes \mathbf{1} + \mathbf{1} \otimes \rho_2).$$

Let U_1, U_2, and $U_1 \otimes U_2$ be distributors and V_1, V_2, and $V_1 \otimes V_2$ be the collectors for $\mathcal{L}_1$, $\mathcal{L}_2$, and $\mathcal{L}_1 \otimes \mathcal{L}_2$, respectively. By using the lumping conditions and Proposition 33(1) and (2) we have that

$$\begin{aligned}
(V_1 \otimes V_2)(U_1 \otimes U_2)(\Pi_1 \otimes \Pi_2)(V_1 \otimes V_2) &= (V_1 U_1 \Pi_1 V_1 \otimes V_2 U_2 \Pi_2 V_2) \\
&= (\Pi_1 V_1 \otimes \Pi_2 V_2) = (\Pi_1 \otimes \Pi_2)(V_1 \otimes V_2) \\
(V_1 \otimes V_2)(U_1 \otimes U_2)(\mathrm{Q}_1 \otimes \Pi_2 + \Pi_1 \otimes \mathrm{Q}_2)(V_1 \otimes V_2) &= V_1 U_1 \mathrm{Q}_1 V_1 \otimes V_2 U_2 \Pi_2 V_2 + V_1 U_1 \Pi_1 V_1 \otimes V_2 U_2 \mathrm{Q}_2 V_2 \\
&= \mathrm{Q}_1 V_1 \otimes \Pi_2 V_2 + \Pi_1 V_1 \otimes \mathrm{Q}_2 V_2 \\
&= (\mathrm{Q}_1 \otimes \Pi_2 + \Pi_1 \otimes \mathrm{Q}_2)(V_1 \otimes V_2) \\
(V_1 \otimes V_2)(U_1 \otimes U_2)(\rho_1 \otimes \mathbf{1} + \mathbf{1} \otimes \rho_2) &= (V_1 U_1 \rho_1 \otimes V_2 U_2 \mathbf{1} + V_1 U_1 \mathbf{1} \otimes V_2 U_2 \rho_2) \\
&= \rho_1 \otimes \mathbf{1} + \mathbf{1} \otimes \rho_2.
\end{aligned}$$

Next, we prove that the lumped parallel composition is the parallel composition of the lumped components. We have, by Proposition 33(1) and (2),

$$(U_1 \otimes U_2)(\Pi_1 \otimes \Pi_2)(V_1 \otimes V_2) = \overline{\Pi}_1 \otimes \overline{\Pi}_2 \quad \text{and}$$
$$(U_1 \otimes U_2)(\mathrm{Q}_1 \otimes \Pi_2 + \Pi_1 \otimes \mathrm{Q}_2)(V_1 \otimes V_2) = \overline{\mathrm{Q}}_1 \otimes \overline{\Pi}_2 + \overline{\Pi}_1 \otimes \overline{\mathrm{Q}}_2.$$

Next, we consider reduction. Let $\Pi_1 = R_1 L_1$ and $\Pi_2 = R_2 L_2$ be some canonical product decompositions. Put $L = L_1 \otimes L_2$ and $R = R_1 \otimes R_2$. Note that $L \geq 0$ and $R \geq 0$ because $L_1, L_2, R_1, R_2 \geq 0$. We also have $L \cdot \mathbf{1} = (L_1 \otimes L_2) \cdot (\mathbf{1} \otimes \mathbf{1}) = L_1 \cdot \mathbf{1} \otimes L_2 \cdot \mathbf{1} = \mathbf{1} \otimes \mathbf{1} = \mathbf{1}$. Since $\mathrm{rank}(A \otimes B) = \mathrm{rank}(A) \cdot \mathrm{rank}(B)$ by Proposition 33(6), we get that (L, R) is a canonical product decomposition of $\Pi = \Pi_1 \otimes \Pi_2$. Reducing $\mathsf{D}_1 \parallel \mathsf{D}_2$ using the canonical product decomposition (L, R) gives us $\mathsf{M}_1 \parallel \mathsf{M}_2$. □

6.2. Composing Markov reward chains with fast transitions

We now present the definition of the parallel composition of Markov reward chains with fast transitions. It comprises Kronecker sums of the generator matrices, i.e., interleaving of the rates for both slow and fast transitions.

Definition 39. Let $\mathsf{F}_1 = (\sigma_1, S_1, F_1, \rho_1)$ and $\mathsf{F}_2 = (\sigma_2, S_2, F_2, \rho_2)$ be two Markov reward chains with fast transitions. Then their parallel composition is defined as

$$\mathsf{F}_1 \parallel \mathsf{F}_2 = (\sigma_1 \otimes \sigma_2, S_1 \oplus S_2, F_1 \oplus F_2, \rho_1 \otimes \mathbf{1} + \mathbf{1} \otimes \rho_2).$$

It is not difficult to see that the parallel composition of Markov reward chains with fast transitions is well-defined.

Example 40. In Fig. 8 we present an example of parallel composition of two Markov reward chains with fast transitions: Fig. 8(c) is the parallel composition of Fig. 8(a) and Fig. 8(b), the same Markov reward chains with fast transitions from the example in Fig. 4. For readability the rewards of Fig. 8(c) are omitted. They are given by the vector

$$(r_1 + r_4, r_1 + r_5, r_1, r_2 + r_4, r_2 + r_5, r_2, r_3 + r_4, r_3 + r_5, r_3).$$

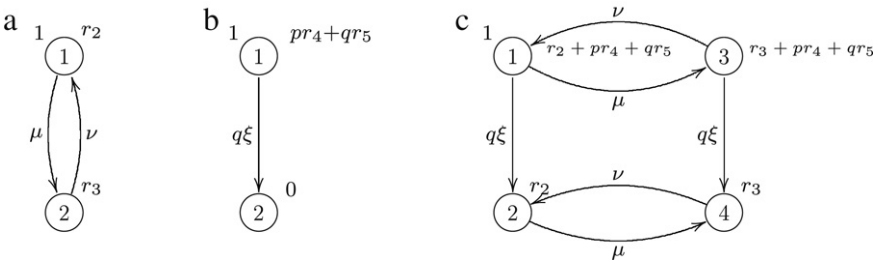


Fig. 9. Aggregated Markov reward chains with fast transitions.

Having defined parallel composition for both models, we show how they are related: the limit of the parallel composition of two Markov reward chains with fast transitions is the parallel composition of the limits of the components (that are discontinuous Markov reward chains). Hence, a continuity property of the parallel composition holds as stated in the next result.

Theorem 41. *Let* $\mathsf{F}_1 \to_\infty \mathsf{D}_1$ *and* $\mathsf{F}_2 \to_\infty \mathsf{D}_2$. *Then* $\mathsf{F}_1 \parallel \mathsf{F}_2 \to_\infty \mathsf{D}_1 \parallel \mathsf{D}_2$.

Proof. Let $\mathsf{F}_1 = (\sigma_1, S_1, F_1, \rho_1)$ and $\mathsf{F}_2 = (\sigma_2, S_2, F_2, \rho_2)$, and let their corresponding limits be $\mathsf{D}_1 = (\sigma_1, \Pi_1, Q_1, \Pi_1\rho_1)$ and $\mathsf{D}_2 = (\sigma_2, \Pi_2, Q_2, \Pi_2\rho_2)$. Using Proposition 33(4) and (5) we get that $\Pi_1 \otimes \Pi_2$ is the ergodic projection of $F_1 \oplus F_2$, i.e. $\lim_{t\to\infty} \mathrm{e}^{(F_1\oplus F_2)t} = \Pi_1 \otimes \Pi_2$. As before, using the distributivity of the Kronecker product and the fact that Π_1 is a stochastic matrix, we derive $Q_1 \otimes \Pi_2 + \Pi_2 \otimes Q_1 = (\Pi_1 \otimes \Pi_2)(S_1 \oplus S_2)(\Pi_1 \otimes \Pi_2)$ and $(\Pi_1 \otimes \Pi_2)(\rho_1 \otimes \mathbf{1} + \mathbf{1} \otimes \rho_2) = \Pi_1\rho_1 \otimes \mathbf{1} + \mathbf{1} \otimes \Pi_2\rho_2$. □

Next we show compositionality of τ-lumping and τ-reduction with respect to the parallel composition of Markov reward chains with fast transitions.

Theorem 42. *If* $\mathsf{F}_1 \overset{\mathcal{L}_1}{\rightsquigarrow} \overline{\mathsf{F}}_1$ *and* $\mathsf{F}_2 \overset{\mathcal{L}_2}{\rightsquigarrow} \overline{\mathsf{F}}_2$, *then* $\mathsf{F}_1 \parallel \mathsf{F}_2 \overset{\mathcal{L}_1\otimes\mathcal{L}_2}{\rightsquigarrow} \overline{\mathsf{F}}_1 \parallel \overline{\mathsf{F}}_2$. *Also, if* $\mathsf{F}_1 \rightsquigarrow_r \mathsf{M}_1$ *and* $\mathsf{F}_2 \rightsquigarrow_r \mathsf{M}_2$, *then* $\mathsf{F}_1 \parallel \mathsf{F}_2 \rightsquigarrow_r \mathsf{M}_1 \parallel \mathsf{M}_2$.

Proof. Let $\mathsf{F}_1 = (\sigma_1, S_1, F_1, \rho_1)$, $\mathsf{F}_2 = (\sigma_2, S_2, F_2, \rho_2)$, $\overline{\mathsf{F}}_1 = (\overline{\sigma}_1, \overline{S}_1, \overline{F}_1, \overline{\rho}_1)$, and $\overline{\mathsf{F}}_2 = (\overline{\sigma}_2, \overline{S}_2, \overline{F}_2, \overline{\rho}_2)$. By Theorem 38 and the continuity result Theorem 41, we get that $\mathcal{L}_1 \otimes \mathcal{L}_2$ is a τ-lumping for $\mathsf{F}_1 \parallel \mathsf{F}_2$. Let W_1 and W_2 be the τ-distributors used for the τ-lumped processes in the assumption, respectively. By Definition 14, Theorem 41, and Definition 34 for the parallel composition of discontinuous Markov reward chains, we have that $W_1 \otimes W_2$ is a τ-distributor for $\mathsf{F}_1 \parallel \mathsf{F}_2$. The τ-lumped process corresponding to $W_1 \otimes W_2$ is exactly $\overline{\mathsf{F}}_1 \parallel \overline{\mathsf{F}}_2$.

We next show the compositionality of τ-reduction. Let $\Pi_1 = R_1L_1$ and $\Pi_2 = R_2L_2$ be the canonical product decompositions of $\Pi_1 = \lim_{t\to\infty} \mathrm{e}^{F_1 t}$ and $\Pi_2 = \lim_{t\to\infty} \mathrm{e}^{F_2 t}$, respectively. Put $L = L_1 \otimes L_2$ and $R = R_1 \otimes R_2$. Then (L, R) is a canonical product decomposition of $\Pi = \Pi_1 \otimes \Pi_2$, as in the proof of Theorem 38. This canonical product decomposition applied to $\mathsf{F}_1 \parallel \mathsf{F}_2$ produces $\mathsf{M}_1 \parallel \mathsf{M}_2$ as the τ-reduced process. □

Example 43. In Fig. 9 we present the aggregated versions of the Markov reward chains with fast transitions from Fig. 8. As expected by Theorem 42, the Markov reward chain with fast transitions in Fig. 9(c) is the parallel composition of the Markov reward chains with fast transitions in Fig. 9(a) and Fig. 9(b) with $p = \frac{c}{b+c}$ and $q = \frac{b}{b+c}$. The aggregated versions can be obtained by either applying τ-reduction or τ-lumping as already discussed in Examples 18 and 26. See [10,11] for more details on the relationship between lumping-based and reduction-based aggregation methods. The τ-lumpings used are $\{\{1, 2\}, \{3\}\}$ for Fig. 8(a) and Fig. 8(b), and $\{\{1, 2, 4, 5\}, \{3, 6\}, \{7, 8\}, \{9\}\}$ for Fig. 8(c).

6.3. Composing Markov reward chains with silent transitions

We define the parallel composition of two Markov reward chains with silent transitions via the equivalence class of the parallel composition of the representative Markov reward chains with fast transitions.

Definition 44. Let $\mathsf{S}_1 = (\sigma_1, S_1, \mathcal{F}_1, \rho_1)$ and $\mathsf{S}_2 = (\sigma_2, S_2, \mathcal{F}_2, \rho_2)$ be two Markov reward chains with silent transitions. Then their parallel composition is defined as

$$\mathsf{S}_1 \parallel \mathsf{S}_2 = (\sigma_1 \otimes \sigma_2, S_1 \oplus S_2, \mathcal{F}_1 \oplus \mathcal{F}_2, \rho_1 \otimes \mathbf{1} + \mathbf{1} \otimes \rho_2),$$

where $\mathcal{F}_1 \oplus \mathcal{F}_2$ denotes the equivalence class of $F_1 \oplus F_2$ with respect to $\sim$, for some $F_1 \in \mathcal{F}_1$ and $F_2 \in \mathcal{F}_2$.

The parallel composition of Markov reward chains with silent transitions is well defined as the Kronecker sum respects the equivalence $\sim$. Next we state the compositionality result for $\tau_\sim$-lumping and $\tau_\sim$-reduction. It is a direct consequence of Theorem 42 for compositionality of τ-lumping and τ-reduction, and compositionality of ordinary lumping for standard Markov reward chain as a special case of Theorem 38.

Theorem 45. *Let* S_1 *and* S_2 *be two Markov reward chains with silent transitions. If* $\mathsf{S}_1 \overset{\mathcal{L}_1}{\rightsquigarrow} \overline{\mathsf{S}}_1$ *and* $\mathsf{S}_2 \overset{\mathcal{L}_2}{\rightsquigarrow} \overline{\mathsf{S}}_2$, *then* $\mathsf{S}_1 \parallel \mathsf{S}_2 \overset{\mathcal{L}_1\otimes\mathcal{L}_2}{\rightsquigarrow} \overline{\mathsf{S}}_1 \parallel \overline{\mathsf{S}}_2$. *Also, if* $\mathsf{S}_1 \overset{\mathcal{L}_1}{\rightsquigarrow}_r \mathsf{M}_1$ *and* $\mathsf{S}_2 \overset{\mathcal{L}_2}{\rightsquigarrow}_r \mathsf{M}_2$, *then* $\mathsf{S}_1 \parallel \mathsf{S}_2 \overset{\mathcal{L}_1\otimes\mathcal{L}_2}{\rightsquigarrow}_r \mathsf{M}_1 \parallel \mathsf{M}_2$. □

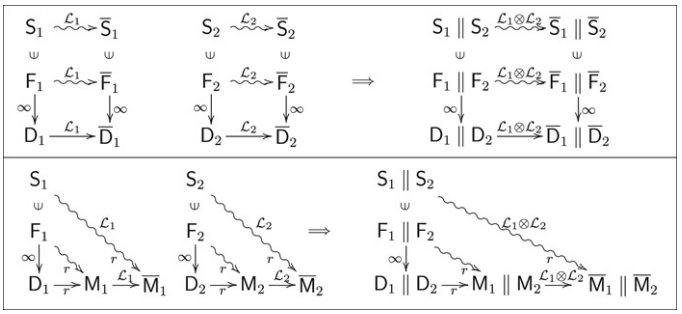


Fig. 10. Summary compositionality results.

7. Conclusion

We considered three types of performance models. Markov reward chains with fast transitions are our central model used for analysing systems with stochastic and instantaneous probabilistic transitions. Their limits are the discontinuous Markov reward chains. Their quotients are the Markov reward chains with silent transitions which can be used for the analysis of systems with stochastic transitions and non-deterministic (internal) τ steps.

For each type of model, we presented two aggregation methods: lumping and reduction for discontinuous Markov reward chains, τ-lumping and τ-reduction for Markov reward chains with fast transitions, and $\tau_\sim$-lumping and $\tau_\sim$-reduction for Markov reward chains with silent transitions. In short, the contributions of the paper are the following.

- A definition of parallel composition of discontinuous Markov reward chains, Markov reward chains with fast transitions, and Markov reward chains with silent transitions allowing for compositional modeling.
- Identification of preorder properties of the aggregation methods for all types of models.
- Compositionality theorems for each type of models and each corresponding aggregation preorder, and a continuity property of the parallel compositions.

The results on compositionality are summarised in Fig. 10 which is justified by the Theorems 29–31 Definition 32, ppsrefprop::kronecker-properties, Definition 34, Theorems 35 and 36, Remark 37, Theorem 38, Definition 39, Example 40, Theorems 41 and 42, Example 43, Definition 44, Theorem 45, as well as by Proposition 17 and
Proposition 25.

Further work focuses on the analysis of models that combine stochastic transitions and (non-internal) action labeled transitions, so that in addition to interleaving, synchronisation can be expressed too.

Acknowledgments

The first author's research has been funded by the Dutch BSIK/BRICKS project AFM 3.2. The second author was supported by the Austrian Science Fund (FWF) project P18913 and by the EU ArtistDesign Network of Excellence on Embedded Systems Design.

References

[1] R.A. Howard, Semi-Markov and Decision Processes, Wiley, 1971.
[2] H. Hermanns, Interactive Markov Chains: The Quest for Quantified Quality, in: Lecture Notes in Computer Science, vol. 2428, Springer, 2002.
[3] J. Hillston, A Compositional Approach to Performance Modelling, Cambridge University Press, 1996.
[4] M. Ajmone Marsan, G. Balbo, G. Conte, S. Donatelli, G. Franceschinis, Modelling with Generalized Stochastic Petri Nets, Wiley, 1995.
[5] G. Ciardo, J. Muppala, K.S. Trivedi, On the solution of GSPN reward models, Performance Evaluation 12 (1991) 237–253.
[6] S.-H. Wu, S.A. Smolka, E. Stark, Composition and behaviors of probabilistic I/O automata, Theoretical Computer Science 176 (1997) 1–38.
[7] L. Cheung, N. Lynch, R. Segala, F. Vaandrager, Switched PIOA: Parallel composition via distributed scheduling, Theoretical Compututer Science 365 (2006) 83–108.
[8] B. Plateau, K. Atif, Stochastic automata network of modeling parallel systems, IEEE Transactions on Software Engineering 17 (1991) 1093–1108.
[9] J. Markovski, N. Trčka, Lumping Markov chains with silent steps, in: Proceedings of QEST'06, IEEE Computer Society, Riverside, 2006, pp. 221–230.
[10] J. Markovski, N. Trčka, Aggregation methods for Markov reward chains with fast and silent transitions, in: F. Bause, P. Buchholz (Eds.), Proceedings of MMB 2008, VDE Verlag, 2008.
[11] N. Trčka, Silent steps in transition systems and Markov chains, Ph.D. Thesis, Eindhoven University of Technology, 2007.
[12] W. Doeblin, Sur l'équation matricielle $A(t+s) = A(t) \cdot A(s)$ et ses applications aux probabilités en chaine, Bulletin des Sciences Mathématiques 62 (1938) 21–32.
[13] M. Coderch, A.S. Willsky, S.S. Sastry, D.A. Castanon, Hierarchical aggregation of singularly perturbed finite state Markov processes, Stochastics 8 (1983) 259–289.
[14] H.H. Ammar, Y.F. Huang, R.W. Liu, Hierarchical models for systems reliability, maintainability, and availability, IEEE Transactions on Circuits and Systems 34 (1987) 629–638.
[15] P. Buchholz, Markovian process algebra: Composition and equivalence, in: Proceedings of PAPM 94, Universität Erlangen-Nürnberg, 1994, pp. 11–30.
[16] P. Buchholz, Structured analysis techniques for large Markov chains, in: Proceedings of SMCTools 2006, Pisa, in: ACM International Conference Proceedings Series, vol. 201, 2006, pp. 2–10.

[17] W.J. Stewart, Introduction to the Numerical Solution of Markov Chains, Princeton University Press, 1994.
[18] J.G. Kemeny, J.L. Snell, Finite Markov Chains, Springer, 1976.
[19] P. Buchholz, Exact and ordinary lumpability in finite Markov chains, Journal of Applied Probability 31 (1994) 59–75.
[20] A. Sokolova, E.P. de Vink, On relational properties of lumpability, in: Proceedings of 4th PROGRESS symposium on Embedded Systems, Utrecht, The Netherlands, 2003.
[21] F. Delebecque, J.P. Quadrat, Optimal control of Markov chains admitting strong and weak interactions, Automatica 17 (1981) 281–296.
[22] F. Delebecque, A reduction process for perturbed Markov chains, SIAM Journal of Applied Mathematics 2 (1983) 325–330.
[23] J. Markovski, A. Sokolova, N. Trcka, E. de Vink, Compositionality for Markov reward chains with fast transitions, in: K. Wolter (Ed.), Proceedings of EPEW 2007, in: Lecture Notes of Computer Science, vol. 4748, Springer, 2007, pp. 18–32.
[24] C. Baier, B. Haverkort, H. Hermanns, J.-P. Katoen, Model checking meets performance evaluation, SIGMETRICS Performance Evaluation Review 32 (2005) 10–15.
[25] M. Kwiatkowska, G. Norman, D. Parker, PRISM: Probabilistic symbolic model checker, in: Proceedings TOOLS 2002, Springer, 2002, pp. 200–204.
[26] H. Hermanns, J.-P. Katoen, J. Meyer-Kayser, M. Siegle, A tool for model checking Markov chains, Software Tools for Technology Transfer 4 (2003) 153–172.
[27] J.L. Doob, Stochastic Processes, Wiley, 1953.
[28] K.L. Chung, Markov Chains with Stationary Probabilities, Springer, 1967.
[29] E. Hille, R.S. Phillips, Functional Analysis and Semi-Groups, AMS, 1957.
[30] R. Milner, A Calculus of Communicating Systems, Springer, 1982.
[31] J. Baeten, W. Weijland, Process Algebra, in: Cambridge Tracts in Theoretical Computer Science, vol. 18, Cambridge University Press, 1990.
[32] V. Nicola, Lumping in Markov reward processes, in: IBM Research Report RC 14719, IBM, 1989.
[33] P. Buchholz, P. Kemper, Kronecker based matrix representations for large Markov chains, in: Validation of Stochastic Systems, in: Lecture Notes in Computer Science, vol. 2925, Springer, 2004, pp. 256–295.
[34] A. Graham, Kronecker Products and Matrix Calculus with Applications, Ellis Horwood, 1981.

J. Markovski graduated in Informatics at the University of Ss. Cyril and Methodius in Skopje, Macedonia in 2001 and in 2004 he obtained a M.Sc. degree from the same institution. He obtained a Ph.D. degree in computer science in the Formal Methods Group at the Eindhoven University of Technology in 2008 with the thesis title "Real and Stochastic Time in Process Algebras for Performance Evaluation". Currently, he is working as a postdoctoral researcher at the Systems Engineering Group at the same university. His fields of interest include supervisory control synthesis from process algebraic specifications, timed and stochastic modeling formalisms and systems, Markovian and non-Markovian models for performance analysis, and process algebras.

A. Sokolova is a postdoctoral researcher at the Computational Systems Group, Department of Computer Sciences, University of Salzburg, Austria. She has obtained a Ph.D. degree at the Technische Universiteit Eindhoven, The Netherlands, in 2005. Between Eindhoven and Salzburg she was a postdoc at the Radboud University in Nijmegen, The Netherlands. Her research interests are in the area of formal methods, in particular probabilistic systems, real-time systems, and the theory of coalgebra as a mathematical theory of dynamic systems.

N. Trčka studied computer science at the Faculty of Mathematics, University of Belgrade, and obtained the degree *Graduated Mathematician for Computer Science* (equivalent to M.Sc.) in 2003. In July 2003 he became a Ph.D. student in the Formal Methods Group of the Department of Mathematics and Computer Science, Eindhoven University of Technology, The Netherlands. He obtained his Ph.D. degree in June 2007, with the thesis *Silent Steps in Transition Systems and Markov Chains*. He currently works as a postdoc in the Architecture of Information Systems group within the same Department.

E.P. de Vink graduated in Mathematics at the University of Amsterdam in 1985. He defended his Ph.D. thesis on direct and continuation semantics of concurrency and logic programming in 1990 at Vrije Universiteit also in Amsterdam. Continuing working at Vrije Universiteit he wrote, together with Jaco de Bakker, the monograph 'Control Flow Semantics' documenting the joint efforts of the Amsterdam Concurrency Group on a decade of research in metric semantics. After a couple of years at KPN Research, the R&D laboratory of the main Dutch telecom operator, he returned to academia in 2000. At present, he is associate professor of applied logic in the Formal Methods group of the Eindhoven University of Technology. Erik de Vink wrote and co-authored over 60 research papers and other scientific contributions and was, up to now, supervisor at five Ph.D. defenses. His current research interest focusses on the coordination of evolving systems and formal modelling in computational biology.

executed on all hosts. With the current approach, originating in the decoupling of task executions from network transmissions, only the code of modules mapped to a host needs to execute on that host leading to fully modular code distribution.

VI. Related work

The foundation for HTL is the LET model, introduced in previous work on Giotto [7] and the E machine [9]. Modular refinement-level time safety checking was shown in the original work on HTL [1]. Modular code generation was described for HTL in [3], [5], and for Giotto in [18]. Modular code generation is a relevant issue also in synchronous reactive programming languages, for it allows reusability and integration of components, but calls for different solutions due to the zero-time rather than LET semantics [10]. The quest for verifiable, predictable, and compositional time-determinism of distributed programs, motivated the work on "network code" programs [19], distributed synchronous reactive programs on so-called "loosely"-time-triggered architectures (LTTA) [20], and further back, work on TTA [21], [22].

Exact schedulability analysis of real-time programs is a complex problem, which motivated the development of sustainable [23], compositional [23], [24], and "interface"-based [25], [26] techniques.

There is significant research interest in component-based real-time software in general, and its modular compilation. We highlight some examples of recent work in this area. BIP components [27] provide a framework where correctness of components is inferred compositionally by properties of sub-components. BIP also works as a "verification back-end" for other component-based systems such as AADL [28] and synchronous languages such as Lustre [29]. Ptolemy actors [11] allow for the composition of heterogeneous models of computation and have been considered in verification [30]. Relational interfaces [31] endow (real-time) interfaces [32] with synchronous input-output relations and are compositional with respect to refinement.

VII. Conclusion

We have presented a modular abstract syntax and semantics for HTL, modular checks of well-formedness, race freedom, and transmission safety, and modular code distribution. Our contributions here complement previous results on HTL time safety and modular code generation, and complete the study of modularity in HTL. Modularity in HTL can be utilized in easy program composition as well as fast program analysis and code generation, but also in runtime patching, which may eventually be used as software foundation for addressing uncertainty in control systems. While there is already an implementation of a modular HTL compiler, an implementation of runtime patching for HTL is still future work.

References

[1] A. Ghosal, T. Henzinger, D. Iercan, C. Kirsch, and A. Sangiovanni-Vincentelli, "A hierarchical coordination language for interacting real-time tasks," in Proc. EMSOFT, 2006.

[2] K. Chatterjee, A. Ghosal, D. Iercan, C. Kirsch, T. Henzinger, C. Pinello, and A. Sangiovanni-Vincentelli, "Logical reliability of interacting real-time tasks," in Proc. DATE, 2008.

[3] A. Ghosal, D. Iercan, C. Kirsch, T. Henzinger, and A. Sangiovanni-Vincentelli, "Separate compilation of hierarchical real-time programs into linear-bounded embedded machine code," in Proc. APGES, 2007.

[4] C. Kirsch, L. Lopes, and E. Marques, "Semantics-preserving and incremental runtime patching of real-time programs," in Proc. APRES, 2008.

[5] D. Iercan, "Contributions to the development of real-time programming techniques and technologies," Ph.D. dissertation, Politehnica University of Timisoara, 2008.

[6] D. Stewart, R. Volpe, and P. Khosla, "Design of dynamically reconfigurable real-time software using port-based objects," IEEE Transactions on Software Engineering, 1997.

[7] T. Henzinger, B. Horowitz, and C. Kirsch, "Giotto: A time-triggered language for embedded programming," Proc. of the IEEE, 2003.

[8] J. Auerbach, D. Bacon, D. Iercan, C. Kirsch, V. Rajan, H. Röck, and R. Trummer, "Low-latency time-portable real-time programming with Exotasks," ACM TECS, 2009.

[9] T. Henzinger and C. Kirsch, "The Embedded Machine: predictable, portable real-time code," in Proc. PLDI, 2002.

[10] R. Lublinerman, C. Szegedy, and S. Tripakis, "Modular code generation from synchronous block diagrams — modularity vs. code size," in Proc. POPL, 2009.

[11] J. Eker, J. Janneck, E. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong, "Taming heterogeneity—the Ptolemy approach," Proc. of the IEEE, 2003.

[12] HTL site, http://htl.cs.uni-salzburg.at.

[13] L. Almeida, P. Pedreiras, and J. Fonseca, "The FTT-CAN protocol: Why and how," IEEE Trans. on Industrial Electronics, 2002.

[14] M. Anand and I. Lee, "Robust and sustainable schedulability analysis of embedded software," in Proc. LCTES, 2008.

[15] S. Baruah and A. Burns, "Sustainable scheduling analysis," in Proc. RTSS, 2006.

[16] G. Buttazzo, Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Norwell, MA, USA: Kluwer Academic Publishers, 1997.

[17] J. Leung and M. Merrill, "A note on preemptive scheduling of periodic, real-time tasks," Information Processing Letters, 1980.

[18] T. Henzinger, C. Kirsch, and S. Matic, "Composable code generation for distributed Giotto," in Proc. LCTES, 2005.

[19] S. Fischmeister, O. Sokolsky, and I. Lee, "A verifiable language for programming real-time communication schedules," IEEE Trans. on Computers, 2007.

[20] S. Tripakis, C. Pinello, A. Benveniste, A. Sangiovanni-Vincent, P. Caspi, and M. Di Natale, "Implementing synchronous models on loosely time triggered architectures," IEEE Trans. on Computers, 2008.

[21] H. Kopetz and G. Bauer, "The Time-Triggered Architecture," Proc. of the IEEE, 2003.

[22] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, and P. Niebert, "From Simulink to SCADE/Lustre to TTA: a layered approach for distributed embedded applications," ACM SIGPLAN Notices, 2003.

[23] A. Easwaran, M. Anand, and I. Lee, "Compositional analysis framework using EDP resource models," in Proc. RTSS, 2007.

[24] L. Thiele, E. Wandeler, and N. Stoimenov, "Real-time interfaces for composing real-time systems," in Proc. EMSOFT, 2006.

[25] R. Alur and G. Weiss, "RTComposer: a framework for real-time components with scheduling interfaces," in Proc. EMSOFT, 2008.

[26] T. Henzinger and S. Matic, "An interface algebra for real-time components," in Proc. RTAS, 2006.

[27] S. Bensalem, M. Bozga, J. Sifakis, and T. Nguyen, "Compositional verification for component-based systems and application," in Proc. ATVA, 2008.

[28] M. Chkouri, A. Robert, M. Bozga, and J. Sifakis, "Translating AADL into BIP-application to the verification of real-time systems," in Proc. MoDELS Workshops, 2008.

[29] M. Bozga, V. Sfyrla, and J. Sifakis, "Modelling synchronous systems in BIP," in Proc. EMSOFT, 2009.

[30] Y. Zhou and E. A. Lee, "Causality interfaces for actor networks," ACM TECS, 2008.

[31] S. Tripakis, B. Lickly, T. Henzinger, and E. Lee, "On relational interfaces," in Proc. EMSOFT, 2009.

[32] L. de Alfaro and T. A. Henzinger, "Interface theories for component-based design," in Proc. EMSOFT, 2001.

Exemplaric Expressivity of Modal Logics

BART JACOBS, *Institute for Computing and Information Sciences, Radboud University Nijmegen, P.O. Box 9010, 6500 GL Nijmegen, The Netherlands.*
E-mail: B.Jacobs@cs.ru.nl

ANA SOKOLOVA, *Department of Computer Sciences, University of Salzburg, Jakob-Haringer-Str. 2, 5020 Salzburg, Austria.*
E-mail: Ana.Sokolova@cs.uni-salzburg.at

Abstract

This article investigates expressivity of modal logics for transition systems, multitransition systems, Markov chains and Markov processes, as coalgebras of the powerset, finitely supported multiset, finitely supported distribution and measure functor, respectively. Expressivity means that logically indistinguishable states, satisfying the same formulas, are behaviourally indistinguishable. The investigation is based on the framework of dual adjunctions between spaces and logics and focuses on a crucial injectivity property. The approach is generic both in the choice of systems and modalities, and in the choice of a 'base logic'. Most of these expressivity results are already known, but the applicability of the uniform setting of dual adjunctions to these particular examples is what constitutes the contribution of the article.

Keywords: Modal logic, Coalgebra, (Dual) Adjunction, Markov chains, Markov processes

1 Introduction

During the last decade, coalgebra [18, 31] has become accepted as an abstract framework for describing state-based dynamic systems. Fairly quickly it was recognized, first in [27], that modal logic is the natural logic for coalgebras and also that coalgebras provide obvious models for modal logics. Intuitively there is indeed a connection, because modal operators can be interpreted in terms of next or previous states, with respect to some transition system or, more abstractly, coalgebra. The last few years have shown a rapid development in this (combined) area [5, 7, 17, 20, 23–25, 28, 30, 32, 33]. One of the more interesting aspects is the use of dualities or *dual adjunctions*.

Here is a brief 'historical' account of how the emergence of dual adjunctions in logical settings can be understood. For reasoning about functors the idea of *predicate lifting* was used already early in [15, 16]. This involves the extension of a predicate (formula) $P \subseteq X$ to a *lifted* predicate $\overline{P} \subseteq TX$, for an endofunctor $T\colon \mathbf{Sets} \to \mathbf{Sets}$ whose coalgebras $X \to TX$ we wish to study. The notion of *invariant* arises via such predicate liftings. Liftings can be described as a function $\mathcal{P}X \to \mathcal{P}TX$, or actually as a natural transformation $\mathcal{P} \Rightarrow \mathcal{P}T$, for the contravariant powerset functor $\mathcal{P}\colon \mathbf{Sets}^{\mathrm{op}} \to \mathbf{Sets}$, cf. [29]. With the introduction of polyadic modal operators [20, 32] this natural transformation morphed into maps of the form $\coprod_{n\in\mathbb{N}}(\mathcal{P}X)^n \to \mathcal{P}TX$, or more abstractly via a functor L into a natural transformation $L\mathcal{P} \Rightarrow \mathcal{P}T$. All this takes place in a situation:

$$T \circlearrowright \mathbf{Sets}^{\mathrm{op}} \underset{\mathcal{P}}{\overset{\mathcal{P}}{\rightleftarrows}} \mathbf{Sets} \circlearrowright L \qquad \text{with} \quad L\mathcal{P} \overset{\sigma}{\Longrightarrow} \mathcal{P}T \tag{1}$$

Vol. 20 No. 5, © The Author, 2009. Published by Oxford University Press. All rights reserved.
For Permissions, please email: journals.permissions@oxfordjournals.org
Published online 3 February 2009 doi:10.1093/logcom/exn093

where we have a dual adjunction $\mathcal{P} \dashv \mathcal{P}$. The category **Sets** on the left describes the spaces on which we have coalgebra structures (of the functor T). **Sets** on the right describes the logical universe, on which there is a functor L for modal operators.

The above dual adjunction thus provides the raw setting for considering coalgebras and their (modal) logics. For specific kinds of coalgebras (given by particular functors T), there may be more structure around. In this article, we shall study examples with the categories **Sets** of sets and **Meas** of measure spaces on the left in (1), and the categories **BA** of Boolean algebras and **MSL** of meet semilattices on the right. The latter capture Boolean logic and logic with only finite conjunctions, respectively. Section 2 will describe the adjunctions involved. Similar adjunctions have been used in process semantics (see e.g. [1]) or more generally in [19].

Section 3 will enrich these dual adjunctions with endofunctors like T and L in the above diagram (1). It also contains two 'folklore' results about the natural transformation involved (the σ in (1)). The most important one is Theorem 4 that relates a certain injectivity condition to the fundamental property of *expressivity* of the logic—which means that logically indistinguishable states are also behaviourally indistinguishable. This theorem is known for some time already in the community and has appeared in print in various places [7, 20, 21], in one form or another. We present a convenient formulation (and proof) that is useful in our setting, but we do not claim it as our contribution.

What we do claim as contribution appears in Section 4. There we use Theorem 4 to prove expressivity for four concrete examples. In all these cases we describe appropriate modalities, and prove expressivity by adding:

- Boolean logic for image-finite transition systems, as coalgebras of the finite powerset functor on **Sets**;
- finite conjunctions logic for multitransition systems, as coalgebras of the finitely supported multiset functor on **Sets**;
- finite conjunctions logic for Markov chains, as coalgebras of the finitely supported discrete (sub)distribution functor on **Sets**;
- finite conjunctions for Markov processes, as coalgebras of the Giry functor on the category of measure spaces.

The first point goes back to [14]. Here, we cast it in the framework of dual adjunctions, with an explicit description of the 'modality' endofunctor L on the category **BA** of Boolean algebras and its relevant properties. An expressivity result of graded modal logic (based on Boolean logic) for multitransition systems already exists [32]. There is also already an expressivity result for Markov chains with the standard modalities and Boolean logic (including negation), cf. [7, 26]. Here, we give a proof that finite conjunctions suffice for expressivity for both multitransition systems and Markov chains, just as they do for non-discrete probabilistic systems [8, 11]. Then we reformulate the expressivity result of [8, 11] within our uniform setting of dual adjunctions. Additionally, we elaborate on the relation between the discrete and non-discrete Markov chains/processes and show precisely how expressivity for Markov processes yields expressivity for Markov chains, resulting in alternative indirect proofs for the third point.

Finally, we should emphasise that we include neither atomic propositions nor action labels in the logics and the systems, since we are interested in the essence of the expressivity results. This way we wish to prepare the ground for arbitrary (possibly modular) extensions in the setting of dual adjunctions.

2 Dual adjunctions

In this section, we shall be interested in an adjoint situation of the form:

$$\mathbb{C}^{\mathrm{op}} \underset{F}{\overset{P}{\rightleftarrows}} \mathbb{A} \qquad \text{with} \quad F \dashv P \tag{2}$$

We informally call this a *dual* adjunction because one of the categories involved occurs naturally in opposite form. In the next section, we shall extend such situations with endofunctors, on $\mathbb{C}$ for systems as coalgebras and on $\mathbb{A}$ for logics, but at this preparatory stage we only look at the adjunctions themselves.

Such situations (2) are familiar in duality-like settings, for instance with $\mathbb{C} = \mathbb{A} = \mathbf{Sets}$, and $P =$ $F =$ 'contravariant powerset', like in (1); with $\mathbb{C} = \mathbf{Sets}$, $\mathbb{A} = \mathbf{PreOrd}$, $P =$ 'contravariant powerset', b $F =$ 'upsets'; or with $\mathbb{C} =$ 'topological spaces' and $\mathbb{A} =$ 'frames'. Such situations are studied systematically in [19], and more recently also in the context of coalgebras and modal logic [5, 6, 20–22]. Typically the functor P describes predicates on spaces and the functor F theories of logical models.

In this situation, it is important to keep track of the direction of arrows. To be explicit, the (components of the) unit and counit of the adjunction $F \dashv P$ are maps $\eta_A : A \to PFA$ in $\mathbb{A}$ and $\varepsilon_X : FPX \to X$ in $\mathbb{C}^{\mathrm{op}}$, i.e. $\varepsilon_X : X \to FPX$ in $\mathbb{C}$. The familiar triangular identities are $P\varepsilon \circ \eta P = \mathrm{id}$ in $\mathbb{A}$ and $\varepsilon F \circ F\eta = \mathrm{id}$ in $\mathbb{C}^{\mathrm{op}}$, i.e. $F\eta \circ \varepsilon F = \mathrm{id}$ in $\mathbb{C}$.

2.1 Examples

The following three instances of the dual adjunction (2) will be used throughout the article.

Sets versus Boolean algebras: The first dual adjunction is between sets and Boolean algebras:

$$\mathbf{Sets}^{\mathrm{op}} \underset{\mathcal{F}_u}{\overset{\mathcal{P}}{\rightleftarrows}} \mathbf{BA} \tag{3}$$

Here, **BA** is the category of Boolean algebras. The functor $\mathcal{P}$ is (contravariant) powerset and $\mathcal{F}_u$ sends a Boolean algebra A to the set of its ultrafilters. These ultrafilters are filters (see below) $\alpha \subseteq A$ such that for each $a \in A$, either $a \in \alpha$ or $\neg a \in \alpha$, but not both. The unit $\eta_A : A \to \mathcal{P}\mathcal{F}_u(A)$ for this adjunction is given by $\eta(a) = \{\alpha \in \mathcal{F}_u(A) \mid a \in \alpha\}$. The adjunction (3) amounts to the standard correspondence:

$$\frac{X \xrightarrow{f} \mathcal{F}_u(A) \quad \text{in } \mathbf{Sets}}{A \xrightarrow{g} \mathcal{P}(X) \quad \text{in } \mathbf{BA}} \qquad \text{via} \qquad \frac{a \in f(x)}{x \in g(a)}$$

Sets versus meet semilattices: The second example uses the category **MSL** of meet semilattices, in a situation:

$$\mathbf{Sets}^{\mathrm{op}} \underset{\mathcal{F}}{\overset{\mathcal{P}}{\rightleftarrows}} \mathbf{MSL} \tag{4}$$

The functor $\mathcal{F}$ sends a meet semilattice A to the set $\mathcal{F}(A)$ of its filters, i.e. to the upsets $\alpha \subseteq A$ which are closed under finite meets: $\top \in \alpha$ and $x, y \in \alpha \Rightarrow x \wedge y \in \alpha$. Here the unit is as before, $\eta_A(a) = \{\alpha \in \mathcal{F}(A) \mid a \in \alpha\}$, and the correspondence is also as before.

Measure spaces versus meet semilattices: Our third example is less standard. It uses the category **Meas** of measure spaces, instead of **Sets**. An object of **Meas** is a pair $\mathcal{X}=(X,S_X)$ of a set X together with a σ-algebra $S_X\subseteq\mathcal{P}(X)$. The latter is a collection of 'measurable' subsets closed under $\emptyset$, complements (negation) and countable unions. We shall use that it is closed, in particular, under finite intersections. A morphism $\mathcal{X}\to\mathcal{Y}$ in **Meas**, from $\mathcal{X}=(X,S_X)$ to $\mathcal{Y}=(Y,S_Y)$, is any measurable function $f: X\to Y$, i.e. a function satisfying $f^{-1}(M)\in S_X$ for each measurable set $M\in S_Y$.

Interestingly, in this case we also have an adjunction with meet semilattices:

$$\mathbf{Meas}^{\mathrm{op}} \underset{\mathcal{F}}{\overset{\mathcal{S}}{\rightleftarrows}} \mathbf{MSL} \tag{5}$$

The functor $\mathcal{S}$ maps a measure space to its σ-algebra, i.e. for $\mathcal{X}=(X,S_X)$, $\mathcal{S}(\mathcal{X})=S_X$. The functor $\mathcal{F}$ is the filter functor from (4) that maps a meet semilattice to the set of its filters, with a σ-algebra generated by the subsets $\eta(a)\subseteq\mathcal{F}(A)$, for $a\in A$. Again we have a bijective correspondence

$$\frac{\mathcal{X}\xrightarrow{f}\mathcal{F}(A)\quad\text{in }\mathbf{Meas}}{A\xrightarrow[g]{}\mathcal{S}(\mathcal{X})\quad\text{in }\mathbf{MSL}}\qquad\text{via}\qquad\frac{a\in f(x)}{x\in g(a)}.$$

– Given a measurable function $f: X\to\mathcal{F}(A)$, we obtain $\widehat{f}: A\to\mathcal{S}(\mathcal{X})$ as

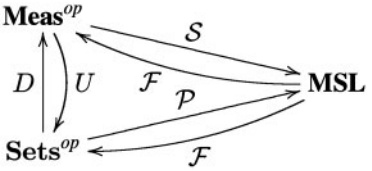
$$\widehat{f}(a)=f^{-1}(\eta(a))=\{x\in X\mid f(x)\in\eta(a)\}=\{x\in X\mid a\in f(x)\}.$$

  This $\widehat{f}$ is well-defined because f is a measurable function, so $f^{-1}(\eta(a))\in\mathcal{S}(\mathcal{X})$, and it preserves finite meets $\top,\wedge$ because η and f^{-1} do.
– Conversely, given $g: A\to\mathcal{S}(\mathcal{X})$ in **MSL** one defines $\widehat{g}: X\to\mathcal{F}(A)$ as $\widehat{g}(x)=\{a\in A\mid x\in g(a)\}$. This yields a filter because g preserves finite meets. The function $\widehat{g}$ is measurable since $\widehat{g}^{-1}(\eta(a))=\{x\in X\mid a\in\widehat{g}(x)\}=g(a)\in\mathcal{S}(\mathcal{X})$.

It is obvious that $\widehat{\widehat{f}}=f$ and $\widehat{\widehat{g}}=g$. The unit $\eta_A: A\to\mathcal{S}\mathcal{F}(A)$ of this adjunction is as before: $\eta_A(a)=\{\alpha\in\mathcal{F}(A)\mid a\in\alpha\}$.

REMARK 1

In the end we notice that the adjunction $\mathbf{Sets}^{\mathrm{op}}\leftrightarrows\mathbf{MSL}$ can be obtained from the adjunction $\mathbf{Meas}^{\mathrm{op}}\leftrightarrows\mathbf{MSL}$ in the following manner. The forgetful functor $U:\mathbf{Meas}\to\mathbf{Sets}$, $U(\mathcal{X})=X$ for $\mathcal{X}=(X,S_X)$, has a left adjoint D which equips a set X with the discrete σ-algebra $\mathcal{P}(X)$ in which all subsets are measurable. Then, when we switch to opposite categories, the forgetful functor $U:\mathbf{Meas}^{\mathrm{op}}\to\mathbf{Sets}^{\mathrm{op}}$ is left adjoint to D. Hence, the adjunction $\mathcal{F}\dashv\mathcal{P}$ between sets and meet semilattices can be obtained as $U\mathcal{F}\dashv\mathcal{S}D$ by composition of adjoints in:



3 Logical set-up

We now extend the adjunction (2) with endofunctors T and L as in:

$$T \circlearrowright \mathbb{C}^{\mathrm{op}} \underset{F}{\overset{P}{\rightleftarrows}} \mathbb{A} \circlearrowright L \qquad \text{with} \quad F \dashv P \tag{6}$$

We shall be interested in coalgebras of the functor T, describing dynamic systems, and in algebras of L, capturing logical models. These models typically contain certain logical connectives, as incorporated in categories of Boolean algebras or of meet semilattices; the functor L adds modal operators. Via a suitable relation between T and L, one can capture logics for dynamic systems in this set-up. But before doing so we recall the following standard result.

PROPOSITION 2
In the situation of the previous diagram we have a bijective correspondence between natural transformations:

$$\frac{LP \overset{\sigma}{\Longrightarrow} PT}{TF \underset{\tau}{\Longrightarrow} FL} \qquad i.e. \qquad \frac{\mathbb{C}^{op} \overset{LP}{\underset{PT}{\Downarrow \sigma}} \mathbb{A}}{\mathbb{A} \overset{FL}{\underset{TF}{\Downarrow \tau}} \mathbb{C}^{op}}$$

PROOF. The correspondence is obtained as follows.

– For $\sigma\colon LP \Rightarrow PT$ one puts:

$$\overline{\sigma} \overset{\text{def}}{=} \Big(TF \overset{\varepsilon TF}{\Longrightarrow} FPTF \overset{F\sigma F}{\Longrightarrow} FLPF \overset{FL\eta}{\Longrightarrow} FL \Big).$$

– Conversely, for $\tau\colon TF \Rightarrow FL$ one similarly takes:

$$\overline{\tau} \overset{\text{def}}{=} \Big(LP \overset{\eta LP}{\Longrightarrow} PFLP \overset{P\tau P}{\Longrightarrow} PTFP \overset{PT\varepsilon}{\Longrightarrow} PT \Big). \qquad \blacksquare$$

ASSUMPTION 3
In the situation (6) we shall assume the following.

– There is an 'interpretation' natural transformation $\sigma\colon LP \Rightarrow PT$.
– The functor $L\colon \mathbb{A} \to \mathbb{A}$ has an initial algebra of 'formulas'. We shall write it as $L(Form) \xrightarrow{\cong} Form$.
– There is a factorization system $(\mathcal{M}, \mathcal{E})$ on the category $\mathbb{C}$ with $\mathcal{M} \subseteq Monos$ and $\mathcal{E} \subseteq Epis$. We shall write the maps in $\mathcal{M}$ as $\rightarrowtail$ and those in $\mathcal{E}$ as $\twoheadrightarrow$, and call them abstract monos and abstract epis, respectively. Hence, every map in $\mathbb{C}$ factors as abstract mono-epi $\twoheadrightarrow\!\!\rightarrowtail$ and the 'diagonal fill-in' property holds: if the outer square

$$\begin{array}{ccc} X & \twoheadrightarrow & Y \\ \downarrow & \nearrow & \downarrow \\ A & \rightarrowtail & B \end{array}$$

commutes, then there exists a diagonal map making the two triangles commute.

– The functor T preserves the maps in $\mathcal{M}$, *i.e.* $m \in \mathcal{M} \Rightarrow T(m) \in \mathcal{M}$.

It is not hard to see that such a factorization system lifts to the category **CoAlg**(T) of coalgebras of the functor T. Given a coalgebra homomorphism $(X \xrightarrow{c} TX) \xrightarrow{f} (Y \xrightarrow{d} TY)$, we can factorize $f = m \circ e$ in $\mathbb{C}$ and obtain a coalgebra d' by diagonal fill-in, since T preserves abstract monos.

$$\begin{array}{ccccc} TX & \xrightarrow{T(e)} & TY' & \xrightarrow{T(m)} & TY \\ {\scriptstyle c}\uparrow & & \uparrow{\scriptstyle d'} & & \uparrow{\scriptstyle d} \\ X & \xrightarrow{e} & Y' & \xrightarrow{m} & Y \\ & & {\scriptstyle f}\searrow & & \nearrow \end{array} \tag{7}$$

One can show that the diagonal fill-in property also holds in **CoAlg**(T), using that T preserves abstract monos, and abstract monos are monos.

For an arbitrary coalgebra $X \xrightarrow{c} TX$ by initiality of *Form* one obtains an interpretation of formulas as predicates on the state space X as in:

$$\begin{array}{ccc} L(Form) & \xrightarrow{L[\![-]\!]} & LPX \\ {\scriptstyle \cong}\downarrow & & \downarrow{\scriptstyle \sigma_X} \\ & & PTX \\ & & \downarrow{\scriptstyle Pc} \\ Form & \xrightarrow[[\![-]\!]]{} & PX \end{array} \tag{8}$$

The adjunction $F \dashv P$ yields a *theory* map $th\colon X \to F(Form)$ corresponding to the interpretation $[\![-]\!]\colon Form \to PX$. Intuitively, for a state $x \in X$, we have a theory $th(x) \in F(Form)$ of formulas that hold in x. Two states $x, y \in X$ will be called *logically indistinguishable*, written as $x \equiv y$, if their theories are the same: $th(x) = th(y)$. In general, logical equivalence is the subobject of $X \times X$ which is the following equalizer in $\mathbb{C}$,

$$\equiv \rightarrowtail X \times X \underset{th \circ \pi_2}{\overset{th \circ \pi_1}{\rightrightarrows}} F(Form)$$

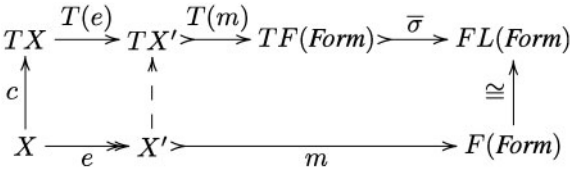
We are interested in comparing logical indistinguishability with behavioural equivalence. Two states $x, y \in X$ of a coalgebra are *behaviourally equivalent*, notation $x \approx y$, if there exists a coalgebra homomorphism f with $f(x) = f(y)$. With the above assumptions, using (7), we may assume that this f is an abstract epi.

Behavioural equivalence may also be formulated for two states $x \in X, y \in Y$ of two different coalgebras (of the same functor). One then requires that there exist coalgebra homomorphisms f and g with $f(x) = g(y)$. This formulation is equivalent to the previous one in categories with pushouts, see [29]. Behavioural equivalence coincides with bisimilarity in case the functor involved preserves weak pullbacks. In the context of expressivity of modal logics behavioural equivalence works better, as commonly accepted in the community, and noted explicitly for probabilistic systems in [8].

It is known and not difficult to show that behavioural equivalence implies logical indistinguishability. For the converse we now present our version of a 'folklore' result (see also [20]).

THEOREM 4
In the context of Assumption 3, if the transpose $\overline{\sigma}\colon TF \Rightarrow FL$ of $\sigma\colon LP \Rightarrow PT$, according to Proposition 2, is componentwise abstract mono, then logically indistinguishable elements are behaviourally equivalent.

PROOF. One factors the theory map $th\colon X \to F(Form)$ as $X \overset{e}{\twoheadrightarrow} X' \overset{m}{\rightarrowtail} F(Form)$. Then $x \equiv y$ if and only if $e(x) = e(y)$. The main point is to obtain a (quotient) coalgebra on X' via the diagonal fill-in property of the factorization in:



Logically indistinguishable elements are then equated by a coalgebra homomorphism (namely e), and are thus behaviourly equivalent. ■

The main technical part of applying Theorem 4 is showing that the natural transformation $\overline{\sigma}$ is mono. That will be the topic of the next section.

3.1 Examples

In the remainder of this section, we extend the three adjunctions in the examples from Section 2 with suitable coalgebra functors—the T in (6)—that we are interested in. Moreover, we discuss that Assumption 3 holds in each case. We consider two base categories, **Sets** and **Meas**. The category **Sets** has a standard factorization system given by monos (injections) and epis (surjections), with a diagonal fill-in property. We discuss the factorisation system on **Meas** below in the section on Markov processes.

Transition systems: We shall write $\mathcal{P}_f\colon \mathbf{Sets} \to \mathbf{Sets}$ for the finite powerset functor:

$$\mathcal{P}_f(X) = \{S \subseteq X \mid S \text{ is finite}\}.$$

A coalgebra $X \to \mathcal{P}_f(X)$ is an image-finite unlabelled transition system. The functor $\mathcal{P}_f$ preserves injections.

Multitransition systems: Next we consider the finitely supported multiset functor $\mathcal{M}_f\colon \mathbf{Sets} \to \mathbf{Sets}$. It is described as follows.

$$\mathcal{M}_f(X) = \{\varphi\colon X \to \mathbb{N} \mid \mathrm{supp}(\varphi) \text{ is finite}\}.$$

The support set of a multiset φ is defined as $\mathrm{supp}(\varphi) = \{x \mid \varphi(x) \neq 0\}$. A function $f\colon X \to Y$ is mapped to $\mathcal{M}_f(f)\colon \mathcal{M}_f(X) \to \mathcal{M}_f(Y)$ by

$$\mathcal{M}_f(f)(\varphi) = \lambda y \in Y.\, \textstyle\sum_{x \in f^{-1}(y)} \varphi(x).$$

A coalgebra $X \to \mathcal{M}_f(X)$ is a multitransition system in which multiple non-labelled transitions are possible between any two states. The functor $\mathcal{M}_f$ preserves injections.

Markov chains: The third endofunctor on **Sets** is the finitely supported discrete subdistribution functor $\mathcal{D}_f$: **Sets** $\to$ **Sets**. It is described as follows.

$$\mathcal{D}_f(X) = \{\varphi\colon X \to [0,1] \mid \mathrm{supp}(\varphi) \text{ is finite and } \textstyle\sum_{x\in X}\varphi(x) \leq 1\}.$$

The support set of a subdistribution φ is defined, as before, as $\mathrm{supp}(\varphi) = \{x \mid \varphi(x) \neq 0\}$. A function $f\colon X \to Y$ yields a mapping $\mathcal{D}_f(f)\colon \mathcal{D}_f(X) \to \mathcal{D}_f(Y)$ by

$$\mathcal{D}_f(f)(\varphi) = \lambda y \in Y.\, \textstyle\sum_{x\in f^{-1}(y)}\varphi(x).$$

A coalgebra $X \to \mathcal{D}_f(X)$ is a Markov chain [3, 9]. In this context, subdistributions (with sum ≤ 1) are more common than distributions (with sum $= 1$), but the difference does not really matter here. The functor $\mathcal{D}_f$ preserves injections.

What subsets, multisets and distributions have in common: Although they might seem different on first sight, the functors $\mathcal{P}_f$, $\mathcal{D}_f$ and $\mathcal{M}_f$ are all instances of the same generic functor. Let $(M, +, 0, \leq)$ be a partially ordered commutative monoid with the property

$$x \leq x + y \quad \text{for all } x, y \in M. \tag{9}$$

Let O be any downward-closed subset of M, $O \subseteq M$ with $x \in O$ whenever $x \leq y, y \in O$. Let $\mathcal{V}_O$ be the functor on **Sets** defined on objects as

$$\mathcal{V}_O(X) = \{\varphi\colon X \to O \mid \mathrm{supp}(\varphi) \text{ is finite and } \textstyle\sum_{x\in X}\varphi(x) \in O\}.$$

We will call the elements of $\mathcal{V}_O(X)$ valuations with values in O. The support set of a valuation is defined as before, $\mathrm{supp}(\varphi) = \{x \in X \mid \varphi(x) \neq 0\}$. Since M is a commutative monoid with the property (9) and O is downward-closed, any valuation $\varphi\colon X \to O$ extends to a function $\mathcal{P}(X) \to O$, which we also denote by φ, by

$$\varphi(S) = \textstyle\sum_{x\in S}\varphi(x).$$

Now, for a function $f\colon X \to Y$, we define $\mathcal{V}_O(f)\colon \mathcal{V}_O(X) \to \mathcal{V}_O(Y)$ as

$$\mathcal{V}_O(f)(\varphi)(y) = \big(\varphi \circ f^{-1}\big)(\{y\}) \quad \text{for} \quad \varphi \in \mathcal{V}_O(X), y \in Y.$$

We have:

- subsets are valuations, $\mathcal{P}_f = \mathcal{V}_O$, for $M = (\{0,1\}, \vee, 0)$ with $O = M$;
- multisets are also valuations, $\mathcal{M}_f = \mathcal{V}_O$ for $M = (\mathbb{N}, +, 0)$ also with $O = M$; and
- subdistributions are valuations as well, $\mathcal{D}_f = \mathcal{V}_O$ for $M = (\mathbb{R}^{\geq 0}, +, 0)$ and $O = [0,1]$. Notice that the requirement that the sum should be in $O = [0,1]$ automatically implies that the sum is at most one, as was required explicitly in the earlier description of Markov chains.

The difference between the functor $\mathcal{P}_f$, on the one hand, and both functors $\mathcal{M}_f$ and $\mathcal{D}_f$, on the other hand, is that the latter are instances of $\mathcal{V}_O$ for $O \subseteq M$ of cancellative monoids M, whereas $M = (\{0,1\}, \vee, 0)$ is not cancellative. This plays an important role for expressivity of the conjunction fragment of suitable modal logics, as we demonstrate below.

Markov processes: On the category **Meas** we consider the Giry functor (or monad) from [13]. It maps a measure space $\mathcal{X} = (X, S_X)$ to the space $\mathcal{G}(\mathcal{X}) = (\mathcal{G}_\mathcal{X}, \mathcal{SG}(\mathcal{X}))$ of subprobability measures

$\varphi\colon S_X \to [0,1]$, satisfying $\varphi(\emptyset)=0$ and $\varphi(\bigcup_i M_i)=\sum_i \varphi(M_i)$ for countable unions of pairwise disjoint subsets $M_i \in S_X$. For each $M \in S_X$ there is an evaluation function $ev_M\colon \mathcal{G}_{\mathcal{X}} \to [0,1]$ given by $\varphi \mapsto \varphi(M)$. The set $\mathcal{G}_{\mathcal{X}}$ is equipped with the smallest σ-algebra $\mathcal{SG}(\mathcal{X})$ making all these maps ev_M measurable. It is generated by the collection:

$$\big\{L_r(M) \mid r \in \mathbb{Q}\cap[0,1], M \in S_X\big\} \quad \text{where} \quad \begin{aligned} L_r(M) &= \{\varphi \in \mathcal{G}_{\mathcal{X}} \mid \varphi(M) \geq r\} \\ &= ev_M^{-1}([r,1]). \end{aligned}$$

These L_r's will be used later as modalities, see (19).

Let $\mathcal{X}=(X,S_X)$ and $\mathcal{Y}=(Y,S_Y)$ be measure spaces. On a measurable function $f\colon X \to Y$ one defines $\mathcal{G}(f)\colon \mathcal{G}(\mathcal{X}) \to \mathcal{G}(\mathcal{Y})$, *i.e.*, $\mathcal{G}(f)\colon \mathcal{G}_{\mathcal{X}} \to \mathcal{G}_{\mathcal{Y}}$ by:

$$\mathcal{G}(f)\Big(S_X \xrightarrow{\varphi} [0,1]\Big) = \Big(S_Y \xrightarrow{f^{-1}} S_X \xrightarrow{\varphi} [0,1]\Big).$$

This $\mathcal{G}(f)$ is a measurable function since for $M \in S_Y$ one has $\mathcal{G}(f)^{-1}(L_r(M)) = L_r(f^{-1}(M))$, where $f^{-1}(M) \in S_X$.

A coalgebra $\mathcal{X} \to \mathcal{G}(\mathcal{X})$ is a Markov process, see [11].

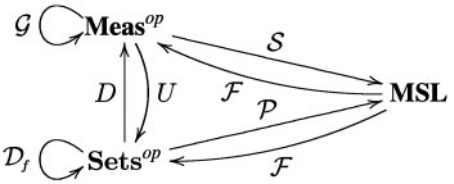
As factorization system on **Meas** we take as abstract monos the collection $\mathcal{M}$ given by morphisms $f\colon \mathcal{X} \to \mathcal{Y}$ for which: f, as function $X \to Y$, is injective, and f^{-1}, as function $S_Y \to S_X$, is surjective. As abstract epis $\mathcal{E}$ we take those morphisms $f\colon \mathcal{X} \to \mathcal{Y}$ for which f, as function $X \to Y$, is surjective. Every morphism $f\colon \mathcal{X} \to \mathcal{Y}$ in **Meas** factors as $X \overset{e}{\twoheadrightarrow} f(X) \overset{m}{\rightarrowtail} Y$ where the image $f(X)$ is given the σ-algebra $S_{f(X)} = \{m^{-1}(M) \mid M \in S_Y\}$. Clearly, $m^{-1}\colon S_Y \to S_{f(X)}$ is surjective. It is not hard to see that this factorization system satisfies the diagonal fill-in property.

Finally we check that $\mathcal{G}$ preserves the maps in $\mathcal{M}$. Given a measurable function $f\colon X \to Y$ in $\mathcal{M}$, so that f is injective and $f^{-1}\colon S_Y \to S_X$ is surjective, we claim that also $\mathcal{G}(f) \in \mathcal{M}$.

- The map $\mathcal{G}(f)$ is injective: assume $\varphi, \psi \in \mathcal{G}_{\mathcal{X}}$ with $\mathcal{G}(f)(\varphi) = \mathcal{G}(f)(\psi)$, i.e. $\varphi \circ f^{-1} = \psi \circ f^{-1}$. Since f^{-1} is surjective, it can be cancelled on the right and we get $\varphi = \psi$.
- The map $\mathcal{G}(f)^{-1}\colon \mathcal{SG}(\mathcal{Y}) \to \mathcal{SG}(\mathcal{X})$ is surjective: let $L_r(M) \in \mathcal{SG}(\mathcal{X})$ be a generator, where $M \in S_X$ and $r \in \mathbb{Q}\cap[0,1]$. Since f^{-1} is surjective we can find a measurable subset $N \in S_Y$ with $M = f^{-1}(N)$. But then $\mathcal{G}(f)^{-1}(L_r(N)) = L_r(f^{-1}(N)) = L_r(M)$. This is enough to conclude surjectivity of $\mathcal{G}(f)^{-1}$ since all set operations are preserved by inverse images.

REMARK 5

In Remark 1, we have used the (discrete) adjunction between sets and measure spaces in order to relate the adjunctions with meet semilattices. Here, we shall also relate the subdistribution functor $\mathcal{D}_f$ and the Giry functor $\mathcal{G}$, in the situation:



There is an obvious natural transformation $\rho\colon \mathcal{D}_f U \Rightarrow U\mathcal{G}$ with component on $\mathcal{X} = (X, S_X) \in \boldsymbol{Meas}$, $\rho_{\mathcal{X}}\colon \mathcal{D}_f(X) \to \mathcal{G}_{\mathcal{X}}$ given by:

$$\left[\begin{array}{l} \varphi \in \mathcal{D}_f(X) \textit{ i.e. } \varphi\colon X \to [0,1] \text{ with} \\ \mathrm{supp}(\varphi) \text{ finite and } \sum_{x\in X} \varphi(x) \leq 1 \end{array}\right] \longmapsto \left[\begin{array}{l} \text{the measure } S_X \to [0,1] \text{ given by} \\ M \mapsto \varphi(M) = \sum_{x\in M} \varphi(x) \end{array}\right]$$

This ρ captures the standard way in which a discrete measure (on points) forms a proper measure (on events/subsets). It plays an imporant role, later on in Section 4.4.

4 Expressivity results

The aim of this section is to show that the functors $\mathcal{P}_f$, $\mathcal{M}_f$, $\mathcal{D}_f$ and $\mathcal{G}$ given in the examples of Section 3 have expressive logics via Theorem 4. Thus we shall define, for each of them, an associated modality functor with *interpretation* natural transformation σ whose transpose $\overline{\sigma}$ is componentwise (abstract) mono. We shall first describe the finite powerset $\mathcal{P}_f$ case because it is already well-studied, see e.g. [7, 14, 22]. This will set the scene for the other examples. Their expressivity is the main topic of this article.

4.1 Boolean logic for image-finite transition systems

Expressivity of modal logic for image-finite transition systems has originally been proved in [14]. Such transition systems can be captured as coalgebras of the finite powerset functor $\mathcal{P}_f$. Coalgebraic generalisations of this expressivity result have been studied in e.g. [7, 32].

Here we shall reproduce this expressivity result for $\mathcal{P}_f$ in the context of dual adjunctions, following [7]. We do so not only in order to prepare for the more complicated probabilistic examples $\mathcal{D}_f$ and $\mathcal{G}$ later on, but also to indicate where the negations of Boolean algebras are used. The main point of the latter examples $\mathcal{M}_f$, $\mathcal{D}_f$ and $\mathcal{G}$ is that they do not involve negation (nor disjunctions).

For a transition system $c\colon X \to \mathcal{P}_f(X)$ as coalgebra, the familiar modal operator $\Box(c)$ is described as:

$$\begin{aligned} \Box(c)(S) &= \{x \in X \mid \forall y. x \to y \Rightarrow y \in S\} \\ &= \{x \in X \mid c(x) \subseteq S\} \\ &= c^{-1}(\Box S), \end{aligned}$$

where $\Box\colon \mathcal{P}(X) \to \mathcal{P}(\mathcal{P}_f(X))$ is defined independently of the coalgebra c as:

$$\Box(S) = \{u \in \mathcal{P}_f(X) \mid u \subseteq S\}. \tag{10}$$

It is not hard to see that $\Box$ preserves finite meets (intersections) and is thus a morphism in the category **MSL**. This leads to the following more abstract description.

PROPOSITION 6
There is an endofunctor $L\colon \mathbf{BA} \to \mathbf{BA}$ in a situation:

$$\mathcal{P}_f \circlearrowright \mathbf{Sets}^{op} \underset{\mathcal{F}_u}{\overset{\mathcal{P}}{\rightleftarrows}} \mathbf{BA} \circlearrowright L$$

which is an instance of (6), such that the definition of $\Box$ in (10) corresponds to an interpretation natural transformation $\boxtimes\colon L\mathcal{P} \Rightarrow \mathcal{P}\mathcal{P}_f$.

This shows that the familiar $\Box$ operator fits into the current framework of dual adjunctions, so that we can use Theorem 4. The details of the functor L are not so relevant, but are given for reasons of completeness.

PROOF. The construction of the functor L (and $\boxtimes$) follows [22]. There is an obvious forgetful functor $V\colon \mathbf{BA} \to \mathbf{MSL}$, which has a left adjoint G. We write $L = GV\colon \mathbf{BA} \to \mathbf{BA}$ for the resulting functor (actually comonad). By θ and ξ we denote the unit and the counit of this adjunction, respectively. Notice that morphisms $LA \to B$ in **BA** correspond, via the adjunction, to functions $VA \to VB$ that preserve finite meets. The map $\Box\colon \mathcal{P}(X) \to \mathcal{P}\mathcal{P}_f(X)$ from (10) is formally such a map $V\mathcal{P}(X) \to V\mathcal{P}\mathcal{P}_f(X)$ in the category **MSL**. Hence it corresponds to a map $\boxtimes\colon L\mathcal{P}(X) \to \mathcal{P}\mathcal{P}_f(X)$ as claimed. ■

REMARK 7
On first sight it might seem that it is enough to take for L the identity functor. However, this is not the case since each component of the interpretation natural transformation is required to be an arrow in **BA**, whereas the modality maps $\Box$ preserve finite intersections only.

The Boolean algebras $LA = GVA$ can be understood as models of Boolean logic with a finite meet preserving modal operator ■: We put $\blacksquare = \theta_{VA} \circ V(\xi_A)\colon VLA \to VLA$. The unit $\theta_{VA}\colon VA \to VLA$ of the adjunction $G \dashv V$ embeds elements (formulas) of a Boolean algebra A into its extension LA with ■—which is illustrated after the next result.

LEMMA 8
The above defined finite meet preserving endofunction (or modal operator) satisfies:

(1) idempotency: $\blacksquare \circ \blacksquare = \blacksquare$;
(2) naturality: $VLf \circ \blacksquare = \blacksquare \circ VLf$, for $f\colon A \to B$ in **BA**;
(3) $V\widehat{f} \circ \blacksquare = f \circ V\xi_A$ for $f\colon VA \to VB$ finite meet preserving, with $\widehat{f}\colon LA \to B$ as corresponding transpose.

This last point of Lemma 8 yields a relation between the three boxes, as expressed by the following commuting diagram.

$$\begin{array}{ccc} VL\mathcal{P}X & \xrightarrow{\ \blacksquare\ } & VL\mathcal{P}X \\ {\scriptstyle V\xi}\downarrow & & \downarrow{\scriptstyle V\boxtimes} \\ V\mathcal{P}X & \xrightarrow{\ \Box\ } & V\mathcal{P}\mathcal{P}_fX \end{array} \tag{11}$$

This diagram tells us that the ■ operator is interpreted appropriately. This is best illustrated for the initial algebra $\alpha\colon L(Form) \xrightarrow{\cong} Form$ with its interpretation map $[\![-]\!]\colon Form \to PX$ from (8), for a coalgebra $c\colon X \to TX$. Consider an arbitrary formula $\varphi \in Form$ and map it into $L(Form)$, formally by considering $\varphi \in V(Form)$ and applying $\theta(\varphi) \in VL(Form)$. Then we can apply ■ and obtain $\blacksquare\theta(\varphi) \in VL(Form)$, which can be sent back to $Form = V(Form)$ via the initial algebra α, resulting in $V(\alpha)(\blacksquare\theta(\varphi)) \in V(Form)$. For convenience, this formula may simply be written as $\blacksquare\varphi$. We have been very explicit about all the maps and (forgetful) functors involved, so that we can check that this

box-formula is interpreted appropriately:

$$\begin{aligned}
[\![\blacksquare\varphi]\!] &= \Big(V[\![-]\!]\circ V\alpha\circ\blacksquare\circ\theta\Big)(\varphi)\\
&= \Big(VPc\circ V\boxtimes\circ VL[\![-]\!]\circ\blacksquare\circ\theta\Big)(\varphi) && \text{by (8)}\\
&= \Big(VPc\circ V\boxtimes\circ\blacksquare\circ VL[\![-]\!]\circ\theta\Big)(\varphi) && \text{by Lemma 8.(2)}\\
&= \Big(VPc\circ\Box\circ V(\xi)\circ\theta\circ V[\![-]\!]\Big)(\varphi) && \text{by (11) and naturality}\\
&= \Big(VPc\circ\Box\circ V[\![-]\!]\Big)(\varphi) && \text{by the triangular identities}\\
&= c^{-1}(\Box[\![\varphi]\!])\\
&= \{x\in X\mid\forall y.x\to y\Rightarrow y\in[\![\varphi]\!]\}.
\end{aligned}$$

The next result now gives a semantical reformulation, following [7], of the expressivity result of [14]. It uses Theorem 4. The proof that we present is substantially more complicated than the standard proof, but we include it in order to illustrate the general method on a well-known example. In the other applications, the proofs actually become simpler than the original ones.

THEOREM 9
The transpose $\overline{\boxtimes}\colon \mathcal{P}_f\mathcal{F}_u\Rightarrow\mathcal{F}_uL$ of $\boxtimes$ in Proposition 6, according to Proposition 2, is componentwise mono. Hence Boolean modal logic with $\Box$ is expressive for image-finite transition systems.

PROOF. By unraveling the definition of $\overline{\boxtimes}$ given in the proof of Proposition 2 we see, for finite $S\subseteq\mathcal{F}_uA$,

$$\overline{\boxtimes}(S)=\Big\{\varphi\in LA\mid S\in\boxtimes\Big(L(\eta)(\varphi)\Big)\Big\},$$

where $\eta=\lambda a.\{\alpha\mid a\in\alpha\}$ is the unit $A\to\mathcal{P}\mathcal{F}_uA$ of the dual adjunction $\mathcal{F}_u\dashv\mathcal{P}$.

In order to prove injectivity of $\overline{\boxtimes}$, assume $S,M\in\mathcal{P}_f\mathcal{F}_u(A)$, $S\neq M$, say $\alpha\in S$, $\alpha\notin M=\{\beta_1,\ldots,\beta_n\}$ for ultrafilters $\alpha,\beta_i\in\mathcal{F}_u(A)$. Then $\alpha\neq\beta_i$, so there are elements $b_i\in\beta_i$ with $b_i\notin\alpha$. These b_i exist because we work in a Boolean algebra, with negation $\neg$, and each ultrafilter γ satisfies either $a\in\gamma$ or $\neg a\in\gamma$, for each $a\in A$.

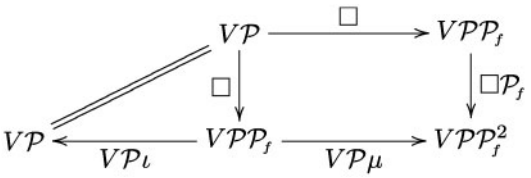
We can put these b_i into LA as $\theta(b_i)$, take their join and write $a=\blacksquare(\bigvee_i\theta(b_i))\in LA$. Then, for any set $W\in\mathcal{P}_f\mathcal{F}_u(A)$,

$$\begin{aligned}
a\in\overline{\boxtimes}(W) &\iff W\in\boxtimes\Big(L(\eta)(\blacksquare\textstyle\bigvee_i\theta(b_i))\Big)\\
&\iff W\in\boxtimes\Big(\blacksquare L(\eta)(\textstyle\bigvee_i\theta(b_i))\Big) && \text{by (2) in Lemma 8}\\
&\iff W\in\boxtimes\Big(\blacksquare\textstyle\bigvee_i L(\eta)(\theta(b_i))\Big)\\
&\iff W\in\boxtimes\Big(\blacksquare\textstyle\bigvee_i\theta(\eta(b_i))\Big) && \text{by naturality}\\
&\iff W\in\Box\Big(V(\xi)\textstyle\bigvee_i\theta(\eta(b_i))\Big) && \text{via (11)}\\
&\iff W\in\Box\Big(\textstyle\bigcup_i V(\xi)\theta(\eta(b_i))\Big) && \text{since } \xi \text{ is a map in } \mathbf{BA}\\
&\iff W\in\Box\Big(\textstyle\bigcup_i\eta(b_i)\Big) && \text{by the triangular equations}\\
&\iff W\subseteq\textstyle\bigcup_i\eta(b_i)\\
&\iff \forall\alpha\in W.\exists i.b_i\in\alpha.
\end{aligned}$$

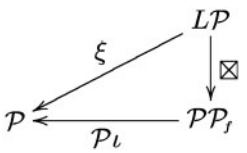
Then $a\in\overline{\boxtimes}(M)$, but $a\notin\overline{\boxtimes}(S)$, proving that $\overline{\boxtimes}$ is mono. ■

REMARK 10
As is well-known, the finite powerset $\mathcal{P}_f$ is a monad on **Sets** and hence a comonad on $\mathbf{Sets}^{\mathrm{op}}$. The functor $L\colon \mathbf{BA} \to \mathbf{BA}$ in Proposition 6 is a comonad by construction. We do not use these comonad structures in this paper but we do like to point out that they are related in the following sense: the functor $\mathcal{P}\colon \mathbf{Sets}^{\mathrm{op}} \to \mathbf{BA}$ with natural transformation $\boxtimes\colon L\mathcal{P} \Rightarrow \mathcal{P}\mathcal{P}_f$ is a morphism of comonads. Hence there is a close connection between the two sides of the dual adjunction in Proposition 6.

The underlying reason is that the natural transformation $\Box\colon \mathcal{P} \Rightarrow \mathcal{P}\mathcal{P}_f$ from (10) commutes with the monad operations $\iota = \{-\}, \mu = \bigcup$ of the monad $\mathcal{P}_f$, in the sense that the following diagram commutes (in **MSL**).



Hence, $\mathcal{P}$ with $\boxtimes$ indeed constitutes a comonad morphism. As a consequence we get, for example, the result:



since:

$$\mathcal{P}\iota \circ \boxtimes = \mathcal{P}\iota \circ \xi \circ G(\Box) = \xi \circ GV\mathcal{P}\iota \circ G(\Box) = \xi.$$

Similarly for commutation of $\boxtimes$ with (co)multiplications.

4.2 *Finite-conjunctions logic for multitransition systems and Markov chains*

The situation for the finitely supported multiset functor and the finitely supported distribution functor $\mathcal{D}_f$ is similar, but a bit more complicated. We obtain an expressivity result for finitely supported cancellative 'valuation' systems and logic with the corresponding modalities and only finite conjunctions. This result then directly instantiates both to multitransition systems and to Markov chains.

It has been shown in [7] that Boolean logic together with probabilistic modalities is expressive for Markov chains. That result also fits in the framework of dual adjunctions [7]. Here we provide an expressivity result for a weaker logic, namely with finite conjunctions only. It has also been shown that graded modal logic is expressive for multitransition systems [32]. Now we show that finite conjunctions suffice in this case as well.

We focus on the valuations functor $\mathcal{V}_O$ for a subset $O \subseteq M$ of an ordered *cancellative* monoid $(M, +, 0)$, with the property $x \leq x+y$ for all $x, y \in M$, as defined in Section 3.1. A subset $\widehat{O} \subseteq O$ is *dense* in O if between any two elements $x, y \in O$ with $x \leq y$, there exists an element $z \in \widehat{O}$ such that $x \leq z \leq y$. Note that O is dense in itself.

For a dense subset $\widehat{O}$ of O, we consider the 'valuation modalities' $\Box_o\colon \mathcal{P}(X) \to \mathcal{P}\mathcal{V}_O(X)$, for $o \in \widehat{O}$, given as:

$$\Box_o(S) = \{\varphi \in \mathcal{V}_O(X) \mid \textstyle\sum_{s \in S} \varphi(x) \geq o\}, \tag{12}$$

From (9) it follows that $\Box_o$ is a monotone function, and thus a map in the category **PoSets** of posets and monotone functions. From these $\Box_o$ we get a modality functor like in Proposition 6, bringing us in the framework of dual adjunctions.

PROPOSITION 11
There is an endofunctor $K_{\widehat{O}}\colon \mathbf{MSL} \to \mathbf{MSL}$ in a situation:

$$\mathcal{V}_O \circlearrowright \mathbf{Sets}^{op} \underset{\mathcal{F}}{\overset{\mathcal{P}}{\rightleftarrows}} \mathbf{MSL} \circlearrowright K_{\widehat{O}}$$

which is an instance of (6), such that all $\Box_o$ in (12) correspond to an interpretation natural transformation $\boxtimes\colon K_{\widehat{O}}\mathcal{P} \Rightarrow \mathcal{P}\mathcal{V}_O$.

PROOF. In order to construct $K_{\widehat{O}}$, we now consider the forgetful functor $V\colon \mathbf{MSL} \to \mathbf{PoSets}$ with its left adjoint H. As before, we denote the unit and the counit of this adjunction by θ and ξ, respectively. We then define the functor $K_{\widehat{O}}\colon \mathbf{MSL} \to \mathbf{MSL}$ as:

$$K_{\widehat{O}}(A) = \coprod_{o \in \widehat{O}} HVA$$

Here we use that the category **MSL** has arbitrary coproducts—which follows for instance from Linton's Theorem [2], using that **MSL** is algebraic over **Sets** (and thus cocomplete). A map $K_{\widehat{O}}(A) \to B$ in **MSL** now corresponds to an $\widehat{O}$-indexed family of (monotone) functions $VA \to VB$ in **PoSets**. The family of maps $\Box_o\colon \mathcal{P}(X) \to \mathcal{P}\mathcal{V}_O(X)$ from (12) is formally such an $\widehat{O}$-indexed family of functions $V\mathcal{P}(X) \to V\mathcal{P}\mathcal{V}_O(X)$ in **PoSets**. Hence, this family corresponds to a (natural) map $\boxtimes\colon K_{\widehat{O}}\mathcal{P}(X) \to \mathcal{P}\mathcal{V}_O(X)$. ■

As before, the meet semilattices $K_{\widehat{O}}(A)$ can be seen as models of logic with only finite conjunctions, with a family of order preserving modal operators $\blacksquare_o\colon VK_{\widehat{O}}(A) \to VK_{\widehat{O}}(A)$, for $o \in \widehat{O}$, defined as composite:

$$V\Big(\coprod_o HVA\Big) \xrightarrow{V(\nabla)} VHVA \xrightarrow{V(\xi_A)} VA \xrightarrow{\theta_{VA}} VHVA \xrightarrow{V(\kappa_o)} V\Big(\coprod_o HVA\Big)$$

where κ_o is a coprojection and $\nabla = [\mathrm{id}]_o$ is the cotuple of identities. These monotone modal operators satisfy idempotency and naturality, and commute appropriately with $\Box$ and $\boxtimes$. We do not elaborate on these $\blacksquare_o$'s because we do not need them (explicitly) in the expressivity proof below.

Recall that the unit of the filter-powerset adjunction $\eta\colon A \to \mathcal{P}\mathcal{F}(A)$, given by $\eta(a) = \{\alpha \in \mathcal{F}A \mid a \in \alpha\}$, preserves finite meets. We define for an arbitrary subset $\alpha \subseteq A$ the set of filters $\uparrow\alpha = \{\beta \in \mathcal{F}A \mid \alpha \subseteq \beta\}$ that contain α. This map $\uparrow\colon \mathcal{P}(A) \to \mathcal{P}\mathcal{F}(A)$ can be seen as free extension of η from A to the complete lattice $(\mathcal{P}(A), \supseteq)$, since:

$$\uparrow\alpha = \{\beta \in \mathcal{F}A \mid \forall a \in \alpha.\, a \in \beta\} = \bigcap_{a \in \alpha} \eta(a).$$

As a result $\alpha \subseteq \alpha'$ implies $\uparrow\alpha \supseteq \uparrow\alpha'$. We further note that:

$$\begin{aligned} \uparrow\{a_1, \ldots, a_n\} &= \{\beta \in \mathcal{F}A \mid a_1, \ldots, a_n \in \beta\} \\ &= \{\beta \in \mathcal{F}A \mid a_1 \wedge \cdots \wedge a_n \in \beta\} \\ &= \eta(a_1 \wedge \cdots \wedge a_n). \end{aligned} \tag{13}$$

The next auxiliary property will be used for showing expressivity.

LEMMA 12
Let X be a set, $S \subseteq \mathcal{P}(X)$, and let $S_f \subseteq S$ be finite. Let $\uparrow_S \alpha = \{\beta \in S \mid \forall a \in \alpha. a \in \beta\}$, for $\alpha \in \mathcal{P}(X)$. Then for each $\alpha \in \mathcal{P}(X)$ there is a finite $\alpha_f \subseteq \alpha$ with $S_f \cap \uparrow_S \alpha = S_f \cap \uparrow_S \alpha_f$.

PROOF. Write $S_f - \uparrow_S \alpha = \{\beta_1, \ldots, \beta_n\}$, where by construction $\alpha \not\subseteq \beta_i$, say via elements $a_i \in \alpha, a_i \notin \beta_i$. Take $\alpha_f = \{a_1, \ldots, a_n\} \subseteq \alpha$. Then $\uparrow_S \alpha_f \supseteq \uparrow_S \alpha$ and so $S_f \cap \uparrow_S \alpha_f \supseteq S_f \cap \uparrow_S \alpha$. For the converse, assume $\beta \in S_f \cap \uparrow_S \alpha_f$. If $\beta \notin \uparrow_S \alpha$, there must be an i with $\beta = \beta_i$ and thus $a_i \notin \beta$. This contradicts $\beta \in \uparrow_S \alpha_f$. ■

We will apply the above lemma to semilattices A with $S = \mathcal{F}(A)$, in which case $\uparrow_S = \uparrow$, as defined above. We next state and prove the expressivity result for logic with finite conjunctions and finitely supported valuation systems. It uses Theorem 4.

THEOREM 13
The transpose $\overline{\boxtimes}\colon \mathcal{V}_O\mathcal{F} \Rightarrow \mathcal{F}K_{\widehat{O}}$ of $\boxtimes$ in Proposition 11, according to Proposition 2, is componentwise mono. Hence modal logic with finite conjunctions and valuation modalities is expressive for finitely supported valuation systems.

PROOF. The transpose $\overline{\boxtimes}\colon \mathcal{V}_O\mathcal{F}(A) \to \mathcal{F}K_{\widehat{O}}(A)$ is given on a valuation $\Phi\colon \mathcal{F}(A) \to O$ on filters of A as:

$$\overline{\boxtimes}(\Phi) = \left\{\varphi \in K_{\widehat{O}}(A) \mid \Phi \in \boxtimes_{\mathcal{F}(A)}\left(K_{\widehat{O}}(\eta)(\varphi)\right)\right\}.$$

We first note that for an element $a \in VA$ we have $\theta(a) \in VHVA$ and thus $a_o = V(\kappa_o)(\theta(a)) \in VK_{\widehat{O}}(A)$, for $o \in \widehat{O}$. For such elements a_o we reason as follows, writing forgetful functors $V\colon \mathbf{MSL} \to \mathbf{PoSets}$ explicitly in order to justify all manipulations.

$$\begin{aligned}
& a_o = V(\kappa_o)(\theta(a)) \in V(\overline{\boxtimes})(\Phi) \\
\iff & \Phi \in \left(V(\boxtimes \circ K_{\widehat{O}}(\eta)) \circ V(\kappa_o) \circ \theta\right)(a) \\
\iff & \Phi \in \left(V([\xi \circ H(\Box_p)]_p \circ [\kappa_p \circ HV(\eta)]_p \circ \kappa_o) \circ \theta\right)(a) \\
& \qquad \text{by definition of } \boxtimes \text{ and } K_{\widehat{O}} \\
\iff & \Phi \in \left(V([\xi \circ H(\Box_p)]_p \circ \kappa_o \circ VHV(\eta) \circ \theta\right)(a) \\
\iff & \Phi \in \left(V(\xi) \circ VH(\Box_o) \circ \theta \circ V(\eta)\right)(a) \\
\iff & \Phi \in \left(V(\xi) \circ \theta \circ \Box_o \circ V(\eta)\right)(a) \\
\iff & \Phi \in \left(\Box_o \circ V(\eta)\right)(a) \\
\iff & \textstyle\sum_{\alpha \in \eta(a)} \Phi(\alpha) \geq o \\
\iff & \Phi(\eta(a)) \geq o.
\end{aligned}$$

Towards injectivity of $\overline{\boxtimes}$ assume $\overline{\boxtimes}(\Phi) = \overline{\boxtimes}(\Psi)$. By the reasoning above we then get $\Phi(\eta(a)) \geq o \iff \Psi(\eta(a)) \geq o$ for all $o \in \widehat{O}$ and all $a \in A$. This yields:

$$\Phi(\eta(a)) = \Psi(\eta(a)). \tag{14}$$

Namely, since $\widehat{O}$ is dense in O, we have that for any $x, y \in O$, $\forall o \in \widehat{O}. x \geq o \iff y \geq o$ implies $\forall o \in O. x \geq o \iff y \geq o$, which further implies (taking $o = x$ and $o = y$) that $x = y$.

Let $S_f = \mathrm{supp}(\Phi) \cup \mathrm{supp}(\Psi)$ be the joined support of Φ and Ψ. Since Φ and Ψ have finite support, S_f is also finite. We now reach our second conclusion: for all filters $\alpha \in \mathcal{F}A$,

$$\begin{aligned}
\Phi(\uparrow\alpha) &= \Phi(S_f \cap \uparrow\alpha) && \text{since values outside } S_f \text{ do not contribute to the sum}\\
&= \Phi(S_f \cap \uparrow\alpha_f) && \text{with } \alpha_f \subseteq \alpha \text{ finite, as in Lemma 12}\\
&= \Phi(\uparrow\alpha_f)\\
&= \Phi(\eta(\textstyle\bigwedge \alpha_f)) && \text{by (13), since } \alpha_f \text{ is finite}\\
&= \Psi(\eta(\textstyle\bigwedge \alpha_f)) && \text{by (14)}\\
&= \ldots && \text{as before}\\
&= \Psi(\uparrow\alpha).
\end{aligned} \tag{15}$$

If $\Phi \neq \Psi$ we can now construct a contradiction: let α be a maximal filter with the property $\Phi(\alpha) \neq \Psi(\alpha)$. Such a maximal filter exists since Φ and Ψ have finite support. Thus $\Phi(\beta) = \Psi(\beta)$ for $\beta \supsetneq \alpha$. But then, since the monoid is cancellative by assumption, we get:

$$\begin{aligned}
\Phi(\uparrow\alpha) &= \Phi(\alpha) + \textstyle\sum_{\beta \supsetneq \alpha} \Phi(\beta)\\
&= \Phi(\alpha) + \textstyle\sum_{\beta \supsetneq \alpha} \Psi(\beta)\\
&\neq \Psi(\alpha) + \textstyle\sum_{\beta \supsetneq \alpha} \Psi(\beta)\\
&= \Psi(\uparrow\alpha),
\end{aligned} \tag{16}$$

contradicting (15). ■

As a consequence we get expressivity for multitransition systems and Markov chains, which we elaborate next. Recall that multitransition systems are $\mathcal{V}_O$-coalgebras for $M = O = \mathbb{N}$ which is a cancellative monoid (with $0, +$). The modalities $\Diamond_k$ of graded modal logic, for $k \in \mathbb{N}$, are given by

$$\Diamond_k(S) = \{\varphi \in \mathcal{M}_f(X) \mid \textstyle\sum_{s \in S} \varphi(x) \geq k\}, \tag{17}$$

and are obviously the valuation modalities for $\widehat{O} = O = \mathbb{N}$. Therefore we get the following result.

COROLLARY 14
The conjunction fragment of graded modal logic is expressive for multitransition systems.

Markov chains are $\mathcal{V}_O$-coalgebras for the cancellative monoid $M = (\mathbb{R}^{\geq 0}, +, 0)$ with $O = [0, 1]$. We consider the dense subset $\widehat{O} = \mathbb{Q} \cap [0, 1]$ of O. The standard probabilistic modalities $L_r : \mathcal{P}(X) \to \mathcal{P}\mathcal{D}_f(X)$ are defined as:

$$L_r(S) = \{\varphi \in \mathcal{D}_f(X) \mid \textstyle\sum_{s \in S} \varphi(x) \geq r\}, \tag{18}$$

for $r \in \mathbb{Q} \cap [0, 1]$, so they coincide with the valuation modalities. Hence we get expressivity for Markov chains.

COROLLARY 15
Finite conjunction modal logic with the standard probabilistic modalities is expressive for Markov chains.

REMARK 16
In the expressivity proof of Theorem 13, a crucial point is the inequality (16), which fails for ordinary transition systems since the Boolean disjunction monoid $(\{0, 1\}, \vee, 0)$ is not cancellative.

4.3 Finite-conjunctions logic for Markov processes

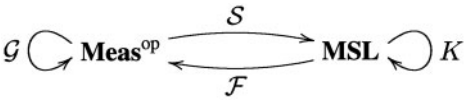
We now present an expressivity result for general, non-discrete, probabilistic systems and logic with the standard modalities and only finite conjunctions. This expressivity result was first shown in [10, 11] for Markov processes over analytic spaces, and recently for general Markov processes over any measure space [8]. It is common in the categorial treatment of non-discrete probabilistic systems (cf. [10–12]) to make the detour through analytic or Polish spaces. The main reason is that bisimilarity (in terms of spans) can not be described in general measure spaces, due to non-existence of pullbacks. However, as we already noted before, behavioural equivalence can, which is also one of the main points of [8] where an explicit characterization of behavioural equivalence under the name *event bisimulation* is given. Hence, we consider general measure spaces.

We start from the general probabilistic modalities $L_r\colon \mathcal{S}(\mathcal{X}) \to \mathcal{SG}(\mathcal{X})$, where $\mathcal{X} = (X, S_X)$, for $r \in \mathbb{Q} \cap [0,1]$.

$$L_r(M) = \{\varphi \in \mathcal{G}_{\mathcal{X}} \mid \varphi(M) \geq r\}, \tag{19}$$

Note that these modalities are well-defined *i.e.* $L_r(M) \in \mathcal{SG}(\mathcal{X})$ by definition.

These modalities are obviously monotone. Hence we can use the functor $K = K_{\widehat{O}} = \coprod_{o \in \widehat{O}} HVA$ for $\widehat{O} = \mathbb{Q} \cap [0,1]$ from Proposition 11. With this functor $K\colon \mathbf{MSL} \to \mathbf{MSL}$ we transform these $L_r\colon V\mathcal{S}(\mathcal{X}) \to V\mathcal{SG}(\mathcal{X})$ in **PoSets** into a natural transformation $\boxtimes\colon K\mathcal{S} \Rightarrow \mathcal{SG}$ in the situation:



This means that $\boxtimes$ satisfies $V(\boxtimes) \circ V(\kappa_r) \circ \theta = L_r$, where $\theta\colon \mathrm{Id} \Rightarrow VH$ is the unit of the adjunction between **MSL** and **PoSets**.

We can now present the expressivity result for logic with finite conjunctions and finitely supported probabilistic systems (from [8, 11]), using Theorem 4. It uses standard measure theoretic results [4], just like the proof in [11] does.

THEOREM 17
The transpose $\overline{\boxtimes}\colon \mathcal{GF} \Rightarrow \mathcal{F}K$ of $\boxtimes$ described above, according to Proposition 2, is componentwise abstract mono. Hence modal logic with finite conjunctions and probabilistic modalities is expressive for Markov processes.

PROOF. The transpose $\overline{\boxtimes}\colon \mathcal{GF}(A) \to \mathcal{F}K(A)$ is given, on a measure $\Phi\colon \mathcal{F}(A) \to [0,1]$ on filters of A, as:

$$\overline{\boxtimes}(\Phi) = \left\{\varphi \in KA \mid \Phi \in \boxtimes_{\mathcal{F}(A)}\left(K(\eta)(\varphi)\right)\right\}.$$

As before, $\overline{\boxtimes}(\Phi) = \overline{\boxtimes}(\Psi)$ implies $\Phi(\eta(a)) = \Psi(\eta(a))$ for all $a \in A$. Now we use [4, Theorem 10.4]: let ν_1, ν_2 be finite measures on a measure space (X, S_X) with generated σ-algebra S_X. If (1) the set of generators is closed under binary intersections, (2) the whole space X is a countable union of generators and (3) ν_1 and ν_2 coincide on the generators, then $\nu_1 = \nu_2$. In our case Φ, Ψ are finite measures on $\mathcal{F}(A)$ with its σ-algebra $\mathcal{SF}(A)$ generated by the sets $\eta(a)$, for $a \in A$. The whole space $\mathcal{F}(A)$ is $\eta(\top)$ since $\top$ is a top element in A which is contained in every filter. Moreover, Φ and Ψ coincide on the generators. Hence, $\Phi = \Psi$ which shows that $\overline{\boxtimes}$ is a mono.

We still need to show that it satisfies the second condition for an abstract mono in **Meas**, namely that $\overline{\boxtimes}^{-1}\colon \mathcal{SF}K(A) \to \mathcal{SGF}(A)$ is a surjective map. For a generator $L_r(N) \in \mathcal{SGF}(A)$, where $N \in \mathcal{SF}(A)$,

we need to find a measurable set $M \in \mathcal{SF}KA$ with $\overline{\boxtimes}^{-1}(M) = L_r(N)$. We shall do so first for generators $N = \eta(a)$, for each $r \in \mathbb{Q} \cap [0,1]$.

Assume $N = \eta(a) = \{\alpha \in \mathcal{F}(A) \mid a \in \alpha\}$, for some $a \in A$. We then take $a_r = (\kappa_r \circ \theta)(a) \in KA$, or more formally, $a_r = V(\kappa_r)\big(\theta_{VA}(a)\big) \in VKA$ as in:

$$a \in VA \xrightarrow{\theta_{VA}} VHVA \xrightarrow{V(\kappa_r)} V\Big(\coprod_{r \in \mathbb{Q} \cap [0,1]} HVA\Big) = VKA.$$

The measurable set $\eta(a_r) \in \mathcal{SF}KA$ does the job in this case:

$$\begin{aligned}
\overline{\boxtimes}^{-1}(\eta(a_r)) &= \{\Phi \mid \overline{\boxtimes}(\Phi) \in \eta(a_r)\} \\
&= \{\Phi \mid a_r \in \overline{\boxtimes}(\Phi)\} \\
&= \{\Phi \mid \Phi \in \boxtimes\big(K(\eta)(a_r)\big)\} \\
&= \boxtimes\big(K(\eta)((\kappa_r \circ \theta)(a))\big) \\
&= \boxtimes\big((\kappa_r \circ \theta)(\eta(a))\big) && \text{by naturality} \\
&= L_r(\eta(a)) && \text{by definition of } \boxtimes.
\end{aligned}$$

We will next show that the σ-algebra Σ_0 generated by the sets $L_r(\eta(a))$ equals the σ-algebra generated by $L_r(M)$ for arbitrary $M \in \mathcal{SF}A$, which we will denote simply by $\Sigma = \mathcal{SGF}A$. Clearly $\Sigma_0 \subseteq \Sigma$. For the converse, we recall the original definition of Σ: the smallest σ-algebra making the evaluation maps ev_M measurable. We aim at showing that Σ_0 also makes all evaluation maps measurable, which completes the proof.

We start by defining the notion of λ-system. A collection of subsets $\Lambda \subseteq \mathcal{P}(X)$ is called a λ-system if (1) $X \in \Lambda$, (2) $M \in \Lambda \Longrightarrow \neg M \in \Lambda$, and (3) $\bigcup_n M_n \in \Lambda$, for all pairwise disjoint $M_n \in \Lambda$ where $n \in \mathbb{N}$.

We will further use the following result from [4, Theorem 3.2]. If $\mathcal{N} \subseteq \mathcal{P}(X)$ is closed with respect to binary intersections, then $\langle \mathcal{N} \rangle_\lambda = \langle \mathcal{N} \rangle_\sigma$, where $\langle \mathcal{N} \rangle_\lambda$ and $\langle \mathcal{N} \rangle_\sigma$ are the smallest λ-system and the smallest σ-algebra containing $\mathcal{N}$, respectively.

In our case, let $\mathcal{N} = \{\eta(a) \mid a \in A\}$, which is closed under finite intersections, and let

$$\mathcal{L} = \{M \in \Sigma = \mathcal{SF}A \mid ev_M : (\mathcal{G}_{\mathcal{F}A}, \Sigma_0) \to [0,1] \text{ is measurable }\}.$$

Note that all evaluation maps ev_M are measurable on the measure space $(\mathcal{G}_{\mathcal{F}A}, \Sigma)$. In $\mathcal{L}$ we gather those M such that ev_M are measurable on the 'smaller' measure space $(\mathcal{G}_{\mathcal{F}A}, \Sigma_0)$. We have that $\eta(a) \in \mathcal{L}$ for all $a \in A$, and the following hold:

(1) $\mathcal{F}A = \eta(\top) \in \mathcal{L}$;
(2) For $M \in \mathcal{L}$, we have that $\neg M = \mathcal{F}A \setminus M$ where $M \subseteq \mathcal{F}A$ and therefore $ev_{\neg M} = ev_{\mathcal{F}A} - ev_M$. Then $ev_{\neg M}$ is a measurable map since measurable maps form a vector space, cf. [4, Theorem 13.3]. Hence $\neg M \in \mathcal{L}$.
(3) Let $M_n \in \mathcal{L}$ for $n \in \mathbb{N}$ be pairwise disjoint sets and let $M = \bigcup_n M_n$. Then, by the sigma-additivity of the measures, we have $ev_M(\varphi) = \sum_n ev_{M_n}(\varphi)$. We further consider the functions $ev^k_M = \sum_{n \leq k} ev_{M_n}$, which are measurable since measurable functions form a vector space. This sequence of functions is pointwise convergent, and therefore we can use [4, Theorem 13.4]: for any sequence of measurable functions with a pointwise limit, the limit function is measurable, to conclude that ev_M is a measurable function, and therefore $M \in \mathcal{L}$.

Hence, the set $\mathcal{L}$ is a λ-system and it contains the set $\mathcal{N} = \{\eta(a) \mid a \in A\}$. By the above mentioned theorem we get that $\langle \mathcal{N} \rangle_\lambda = \langle \mathcal{N} \rangle_\sigma = \mathcal{SF}A$. Since $\mathcal{N} \subseteq \mathcal{L}$ and $\mathcal{L}$ is a λ-system, we get that $\langle \mathcal{N} \rangle_\lambda \subseteq \mathcal{L}$, which implies that $\mathcal{L} = \mathcal{SF}A$ and completes the proof. ■

4.4 *Relating Markov chains and Markov processes*

In Remarks 1 and 5, we have already seen how the categories and functors for Markov chains and for Markov processes are related. Here we elaborate further on these relations and complete the picture by showing how to obtain expressivity for chains from expressivity for processes. This depends on auxiliary results which are of interest on their own.

Recall that Markov chains and Markov processes are related via the natural transformation $\rho \colon \mathcal{D}_f U \Rightarrow U\mathcal{G}$ given by $\rho(\varphi) = \lambda M. \sum_{x \in M} \varphi(x)$, where U is the forgetful functor $\mathbf{Meas} \to \mathbf{Sets}$, with the discrete measure functor $D \colon \mathbf{Sets} \to \mathbf{Meas}$ as left adjoint. Using ρ we can transform any Markov chain, i.e. a $\mathcal{D}_f$-coalgebra in **Sets**, $c \colon X \to \mathcal{D}_f(X)$ into a $U\mathcal{G}D$-coalgebra in **Sets**, by

$$\Big(X \xrightarrow{c} \mathcal{D}_f(X) = \mathcal{D}_f UD(X)\Big) \longmapsto \Big(X \xrightarrow{c} \mathcal{D}_f UD(X) \xrightarrow{\rho_{DX}} U\mathcal{G}D(X)\Big).$$

The latter coalgebras are in one-one correspondence, via the adjunction $D \dashv U$, with $\mathcal{G}$-coalgebras $D(X) \to \mathcal{G}D(X)$ in **Meas** with carriers discrete measure spaces.

$$\frac{X \longrightarrow U\mathcal{G}D(X) \quad \text{in } \mathbf{Sets}}{D(X) \longrightarrow \mathcal{G}D(X) \quad \text{in } \mathbf{Meas}} \tag{20}$$

We use the term discrete Markov process both for a $\mathcal{G}$-coalgebra with carrier discrete measure space in **Meas**, and for the corresponding $U\mathcal{G}D$-coalgebra in **Sets**. The whole picture is shown in the following diagram:

$$\begin{array}{ccccc} & & \overset{\mathcal{T}}{\frown} & & \\ \mathbf{CoAlg}(\mathcal{D}_f) & \xrightarrow[\rho D \circ -]{} & \mathbf{CoAlg}(U\mathcal{G}D) & \xrightarrow[(20)]{} & \mathbf{CoAlg}(\mathcal{G}) \\ & \searrow & \downarrow & & \downarrow \\ & & \mathbf{Sets} & \xrightarrow{\quad D \quad} & \mathbf{Meas} \end{array} \tag{21}$$

where the square on the right is a pullback of functors.

An important but non-trivial result, see Theorem 18 below, is that behavioural equivalence on a Markov chain c and on the corresponding discrete Markov processes $\mathcal{T}(c)$ coincide. Also the logical theories coincide. This leads to a direct proof of expressivity for Markov chains from expressivity for Markov processes. We can also obtain expressivity for Markov chains from expressivity for Markov processes in an indirect way, using Theorem 4, without explicitly comparing behavioural equivalence. Both proofs are presented below.

Behavioural equivalence coincides: The relationship between Markov chains and Markov processes is made explicit with the following theorem. It states that Markov chains can be embedded in the class of Markov processes, namely as discrete Markov processes.

THEOREM 18
The translation functor $\big(c\colon X\to\mathcal{D}_f(X)\big)\mapsto\big(\mathcal{T}(c)\colon D(X)\to\mathcal{G}D(X)\big)$ from (21) preserves and reflects behavioural equivalence: for two states $x,x'\in X$ we have

$$x\approx x' \text{ in } c \iff x\approx x' \text{ in } \mathcal{T}(c).$$

The proof outlines the main steps and refers to auxiliary results in Appendix A.

PROOF. The direction ($\Rightarrow$) is obvious, by functoriality. The reverse direction is done in two steps. Lemma 24 tells that the functor $\rho D\circ -$ in (21) reflects behavioural equivalence, so it remains to show that the functor labeled with (20) does.

We use that the category **Meas** is cocomplete. Its colimits are constructed as in **Sets**, with those subsets measurable that make the coprojections measurable functions, see also [8]. These colimits are inherited by $\mathbf{CoAlg}(\mathcal{G})$. For an arbitrary coalgebra $c\colon\mathcal{X}\to\mathcal{G}(\mathcal{X})$ we can consider all abstract epis $h\colon\mathcal{X}\twoheadrightarrow\mathcal{Y}$ with coalgebra $d\colon\mathcal{Y}\to\mathcal{G}(\mathcal{Y})$ forming a homomorphism. Since such abstract epis are surjections, they correspond[1] to equivalence relations on the underlying set of $\mathcal{X}$. Hence these pairs (h,d) form a proper set, and so we can form their joint pushout in:

$$\begin{array}{ccc}\mathcal{G}(\mathcal{X}) & \xrightarrow{\mathcal{G}(\pi)} & \mathcal{G}(\mathcal{X}_{\approx})\\ {\scriptstyle c}\uparrow & & \uparrow{\scriptstyle c_{\approx}}\\ \mathcal{X} & \xrightarrow{\ \pi\ } & \mathcal{X}_{\approx}\end{array}$$

Then $\approx\subseteq\ker(\pi)=\{(x,x')\mid\pi(x)=\pi(x')\}$, by construction.

If we apply this starting from a discrete Markov process $c\colon D(X)\to\mathcal{G}D(X)$ we also obtain such a homomorphism $\pi\colon D(X)\twoheadrightarrow\mathcal{Y}$ with $\approx\subseteq\ker(\pi)$. Lemma 25.3 says that $\mathcal{Y}$ then 'separates points', and Lemma 27 that π is a homomorphism to a discrete $\mathcal{G}$-coalgebra on the carrier of $\mathcal{Y}$. This completes the proof. ■

Relating the logics and the expressivity results: We will now relate the logics for Markov chains and Markov processes, and show that the expressivity result for chains follows from the expressivity result for processes. We start by comparing the logic interpretations for Markov chains and Markov processes. In order to disambiguate the two modal operators L_r for chains (18) and processes (19) we shall now write them with additional superscripts, namely:

$$\begin{aligned}\mathcal{P}(X)&\xrightarrow{L_r^{\mathcal{D}_f}}\mathcal{P}\mathcal{D}_f(X)\quad\text{is } S\longmapsto\{\varphi\in\mathcal{D}_f(X)\mid\textstyle\sum_{x\in S}\varphi(x)\geq r\}\\ \mathcal{S}(\mathcal{X})&\xrightarrow{L_r^{\mathcal{G}}}\mathcal{S}\mathcal{G}(\mathcal{X})\quad\text{is}\quad M\longmapsto\{\varphi\in\mathcal{G}(\mathcal{X})\mid\varphi(M)\geq r\}.\end{aligned}$$

They give rise to natural transformations:

$$K\mathcal{P}\xrightarrow{\boxtimes^{\mathcal{D}_f}}\mathcal{P}\mathcal{D}_f\quad\text{and}\quad K\mathcal{S}\xrightarrow{\boxtimes^{\mathcal{G}}}\mathcal{S}\mathcal{G}$$

with transposes:

$$\mathcal{D}_f\mathcal{F}\xrightarrow{\overline{\boxtimes}^{\mathcal{D}_f}}\mathcal{F}K\quad\text{and}\quad\mathcal{G}\mathcal{F}\xrightarrow{\overline{\boxtimes}^{\mathcal{G}}}\mathcal{F}K$$

[1] To be precise, equivalence classes of surjections $X\twoheadrightarrow\bullet$ correspond to equivalence relations on X.

LEMMA 19
For a set X, a distribution $\varphi \in \mathcal{D}_f(X)$ and a subset $S \in \mathcal{P}(X) = \mathcal{S}(DX)$, where $D(X)$ is the set X with the discrete σ-algebra $\mathcal{P}(X)$, one has:

$$\varphi \in L_r^{\mathcal{D}_f}(S) \iff \rho(\varphi) \in L_r^{\mathcal{G}}(S).$$

In a diagram:

$$\begin{array}{ccccc} V\mathcal{P}(X) & \xrightarrow{L_r^{\mathcal{D}_f}} & V\mathcal{P}\mathcal{D}_f(X) & = & V\mathcal{P}\mathcal{D}_f U D(X) \\ \| & & & & \uparrow{\scriptstyle V\mathcal{P}(\rho_{DX})} \\ V\mathcal{S}D(X) & \xrightarrow{L_r^{\mathcal{G}}} & V\mathcal{S}\mathcal{G}D(X) & \hookrightarrow & V\mathcal{P}U\mathcal{G}D(X) \end{array}$$

where V is the forgetful functor $\mathbf{MSL} \to \mathbf{PoSets}$. As a result:

$$\boxtimes^{\mathcal{D}_f} = \mathcal{P}(\rho) \circ \boxtimes^{\mathcal{G}} \qquad \text{and} \qquad \overline{\boxtimes}^{\mathcal{D}_f} = U(\overline{\boxtimes}^{\mathcal{G}}) \circ \rho\mathcal{F},$$

where the latter equation involves the diagram:

$$\begin{array}{ccccc} \mathcal{D}_f\mathcal{F} & & \xrightarrow{\overline{\boxtimes}^{\mathcal{D}_f}} & & \mathcal{F}K \\ \| & & & & \| \\ \mathcal{D}_f U\mathcal{F} & \xrightarrow{\rho\mathcal{F}} & U\mathcal{G}\mathcal{F} & \xrightarrow{U(\overline{\boxtimes}^{\mathcal{G}})} & U\mathcal{F}K \end{array}$$

PROOF. By unravelling the definitions we get:

$$\begin{aligned} \varphi \in L_r^{\mathcal{D}_f}(S) &\iff \rho(\varphi)(S) = \textstyle\sum_{x \in S} \varphi(x) \geq r \\ &\iff \rho(\varphi) \in L_r^{\mathcal{G}}(S). \end{aligned}$$

Then

$$\begin{aligned} \boxtimes^{\mathcal{D}_f} &= [\xi \circ H(L_r^{\mathcal{D}_f})]_r \\ &= [\xi \circ H(V\mathcal{P}(\rho) \circ L_r^{\mathcal{G}})]_r \\ &= [\mathcal{P}(\rho) \circ \xi \circ H(L_r^{\mathcal{G}})]_r \\ &= \mathcal{P}(\rho) \circ [\xi \circ H(L_r^{\mathcal{G}})]_r \\ &= \mathcal{P}(\rho) \circ \boxtimes^{\mathcal{G}}. \end{aligned}$$

Now for $\Phi \in \mathcal{D}_f\mathcal{F}(A)$,

$$\begin{aligned} \big(U(\overline{\boxtimes}^{\mathcal{G}}) \circ \rho_{\mathcal{F}(A)}\big)(\Phi) &= \{\psi \in KA \mid \rho(\Phi) \in \boxtimes^{\mathcal{G}}(K(\eta)(\psi))\} \\ &= \{\psi \in KA \mid \Phi \in \rho^{-1}\boxtimes^{\mathcal{G}}(K(\eta)(\psi))\} \\ &= \{\psi \in KA \mid \Phi \in \big(\mathcal{P}(\rho) \circ \boxtimes^{\mathcal{G}}(K(\eta)(\psi))\big)\} \\ &= \{\psi \in KA \mid \Phi \in \boxtimes^{\mathcal{D}_f}(K(\eta)(\psi))\} \\ &= \overline{\boxtimes}^{\mathcal{D}_f}(\Phi). \end{aligned}$$

■

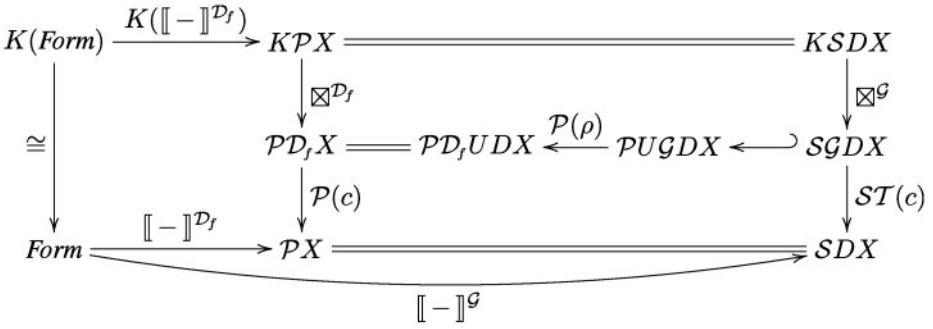
The next result shows that the translation functor does not change the theory maps.

LEMMA 20
For a coalgebra $c\colon X \to \mathcal{D}_f(X)$ let $[\![-]\!]^{\mathcal{D}_f}$ be the corresponding interpretation map from (8) arising from $\boxtimes^{\mathcal{D}_f}$, and let $[\![-]\!]^{\mathcal{G}}$ correspond to $\mathcal{T}(c)\colon D(X) \to \mathcal{G}(D(X))$ with $\boxtimes^{\mathcal{G}}$. Then

$$[\![-]\!]^{\mathcal{D}_f} = [\![-]\!]^{\mathcal{G}} \quad \text{and } th^{\mathcal{D}_f} = th^{\mathcal{G}}$$

where $th^{\mathcal{D}_f}$ and $th^{\mathcal{G}}$ are the transposes of $[\![-]\!]^{\mathcal{D}_f}$ and $[\![-]\!]^{\mathcal{G}}$, obtained via the dual adjunctions $\mathcal{F} \dashv \mathcal{P}$ from (4) and $\mathcal{F} \dashv \mathcal{S}$ from (5).

PROOF. We obtain $[\![-]\!]^{\mathcal{D}_f} = [\![-]\!]^{\mathcal{G}}$ by initiality in:



The upper right square commutes by the previous lemma, and the lower right one by construction of $\mathcal{T}(c)$.

Hence also $th^{\mathcal{D}_f} = th^{\mathcal{G}}$, formally as maps $th^{\mathcal{D}_f}\colon X \to \mathcal{F}(Form)$ in **Sets** and $th^{\mathcal{G}}\colon DX \to \mathcal{F}(Form)$ in **Meas**. ■

COROLLARY 21
Expressivity of modal logic for Markov chains follows from expressivity for Markov processes.

PROOF. By the earlier results, using expressivity for Markov processes in $(*)$:

$$\begin{aligned} x \approx x' \text{ for } X \xrightarrow{c} \mathcal{D}_f(X) &\overset{\text{Thm. 18}}{\Longleftrightarrow} x \approx x' \text{ for } D(X) \xrightarrow{\mathcal{T}(c)} \mathcal{G}D(X) \\ &\overset{(*)}{\Longleftrightarrow} th^{\mathcal{G}}(x) = th^{\mathcal{G}}(x') \text{ for } D(X) \xrightarrow{\mathcal{T}(c)} \mathcal{G}D(X) \\ &\overset{\text{Lem. 20}}{\Longleftrightarrow} th^{\mathcal{D}_f}(x) = th^{\mathcal{D}_f}(x') \text{ for } X \xrightarrow{c} \mathcal{D}_f(X). \end{aligned}$$

■

We can also obtain expressivity for Markov chains from expressivity for Markov processes in an indirect way, using Theorem 4, without explicitly comparing behavioural and logical equivalence. It follows from the next property, the proof of which is in Appendix B.

LEMMA 22
The natural transformation $\rho\mathcal{F}\colon \mathcal{D}_f U\mathcal{F} \Rightarrow U\mathcal{G}\mathcal{F}$ is componentwise mono.

COROLLARY 23
Expressivity of modal logic for Markov chains follows from expressivity for Markov processes, in the sense that $\overline{\boxtimes}^{\mathcal{D}_f}$ is componentwise mono because $\overline{\boxtimes}^{\mathcal{G}}$ is.

PROOF. By Theorem 17 we know that $\overline{\boxtimes}^{\mathcal{G}}$ is componentwise mono. Hence so is $U(\overline{\boxtimes}^{\mathcal{G}})$, because the forgetful functor U: **Meas** $\to$ **Sets** is a right adjoint (and thus preserves monos). By Lemma 19 we know that $\overline{\boxtimes}^{\mathcal{D}_f} = U(\overline{\boxtimes}^{\mathcal{G}}) \circ \rho\mathcal{F}$. Lemma 22 shows that $\rho\mathcal{F}$ is componentwise mono. This completes the alternative proof. ■

5 Conclusions

We have analysed the semantics and logic of four examples of possibilistic and probabilistic state-based systems in a uniform categorical framework and proved expressivity in each of these cases.

Acknowledgements

We thank Harald Woracek and the anonymous referee for many helpful suggestions that lead to significant improvements of the article. We also thank Alexander Kurz and Ernst Erich Doberkat for providing some background information.

Funding

Austrian Science Fund (FWF) (project P18913 to A.S.).

References

[1] S. Abramsky. Domain theory in logical form. *Annals of Pure and Applied Logic*, **51**, 1–77, 1991.
[2] M. Barr and Ch. Wells. *Toposes, Triples and Theories*. Springer, Berlin, 1985. [Revised and corrected version available from URL: www.cwru.edu/artsci/math/wells/pub/ttt.html.]
[3] F. Bartels, A. Sokolova, and E. de Vink. A hierarchy of probabilistic system types. *Theoretical Computer Science*, **327**, 3–22, 2004.
[4] P. Billingsley. *Probability and Measure*. Wiley-Interscience, New York, 1995.
[5] M. Bonsangue and A. Kurz. Duality for logics of transition systems. In *Foundations of Software Science and Computation Structures*, V. Sassone, ed. Vol. 3441 of *Lecture Notes in Computer Science*, pp. 455–469. Springer, Berlin, 2006.
[6] M. Bonsangue and A. Kurz. Presenting functors by operations and equations. In *Foundations of Software Science and Computation Structures*, L. Aceto and A. Ingólfsdóttir, eds. Vol. 3921 of *Lecture Notes in Computer Science*, pp. 172–186. Springer, Berlin, 2006.
[7] C. Cïrstea and D. Pattinson. Modular proof systems for coalgebraic logics. *Theoretical Computer Science*, **388**, 83–108, 2007.
[8] V. Danos, J. Desharnais, F. Laviolette, and P. Panangaden. Bisimulation and cocongruence for probabilistic systems. *Information and Computation*, **204**, 503–523, 2006.
[9] E. P. de Vink and J. J. M. M. Rutten. Bisimulation for probabilistic transition systems: a coalgebraic approach. *Theoretical Computer Science*, **221**, 271–293, 1999.
[10] J. Desharnais, A. Edalat, and P. Panangaden. A logical characterization of bisimulation for labeled markov processes. In *Logic in Computer Science*, pp. 478–487. IEEE Computer Society, Indianopolis, IN, USA, 1998.
[11] J. Desharnais, A. Edalat, and P. Panangaden. Bisimulation for labeled Markov processes. *Information and Computation*, **179**, 163–193, 2002.

[12] E.-E. Doberkat. Eilenberg-Moore algebras for stochastic relations. *Information and Computation*, **204**, 1756–1781, 2006.
[13] M. Giry. A categorical approach to probability theory. In *Categorical Aspects of Topology and Analysis*, B. Banaschewski, ed. Vol. 915 of *Lecture Notes in Mathematics*, pp. 68–85. Springer, Berlin, 1982.
[14] M. Hennessy and R. Milner. On observing non-determinism and concurrency. In *Mathematical Foundations of Computer Science*, J. W. de Bakker and J. van Leeuwen, eds. Vol. 85 of *Lecture Notes in Computer Science*, pp. 299–309. Springer, Berlin, 1980.
[15] C. Hermida. *Fibrations, Logical Predicates and Indeterminates*. PhD thesis, University of Edinburgh. *Technical Report LFCS-93-277*, 1993. [Also available as *Technical Report PB-462*, Aarhus University of DAIMI.
[16] C. Hermida and B. Jacobs. Structural induction and coinduction in a fibrational setting. *Information and Computation*, **145**, 107–152, 1998.
[17] B. Jacobs. Many-sorted coalgebraic modal logic: a model-theoretic study. *Informatique Théorique et Applications*, **35**, 31–59, 2001.
[18] B. Jacobs and J. Rutten. A tutorial on (co)algebras and (co)induction. *EATCS Bulletin*, **62**, 222–259, 1997.
[19] P. T. Johnstone. *Stone Spaces*. In *Cambridge Studies in Advanced Mathematics*, Vol. 3. Cambridge University Press, Cambridge, 1982.
[20] B. Klin. Coalgebraic modal logic beyond sets. In *Mathematical Foundations of Programming Semantics*, M. Fiore, ed. Vol. 173 of *Electronic Notes in Theoretical Computer Science*. Elsevier, Amsterdam, 2007.
[21] C. Kupke, A. Kurz, and D. Pattinson. Algebraic semantics for coalgebraic logics. In *Coalgebraic Methods in Computer Science*, Vol. 106 of *Electronic Notes in Theoretical Computer Science*, pp. 219–241. Elsevier, Amsterdam, 2004.
[22] C. Kupke, A. Kurz, and Y. Venema. Stone coalgebras. In *Coalgebraic Methods in Computer Science*, H. P. Gumm, ed. Vol. 82(1) of *Electronic Notes in Theoretical Computer Science*. Elsevier, Amsterdam, 2003.
[23] C. Kupke, A. Kurz, and Y. Venema. Stone coalgebras. *Theoretical Computer Science*, **327**, 109–134, 2004.
[24] A. Kurz. Specifying coalgebras with modal logic. *Theoretical Computer Science*, **260**, 119–138, 2001.
[25] A. Kurz and D. Pattinson. Coalgebraic modal logic of finite rank. *Mathematical Structures in Computer Science*, **15**, 453–473, 2005.
[26] K. G. Larsen and A. Skou. Bisimulation through probabilistic testing. *Information and Computation*, **94**, 1–28, 1991.
[27] L. S. Moss. Coalgebraic logic. *Annals of Pure and Applied Logic*, **96**, 277–317, 1999; Erratum in *Annals of Pure and Applied Logic*, **99**, 241–259, 1999.
[28] D. Pattinson. An introduction to the theory of coalgebras. *Course notes at the North American Summer School in Logic, Language and Information (NASSLLI)*. 2003.
[29] D. Pattinson. Coalgebraic modal logic: Soundness, completeness and decidability of local consequence. *Theoretical Computer Science*, **309** 177–193, 2003.
[30] D. Pavlović, M. Mislove, and J. Worrell. Testing semantics: connecting processes and process logics. In *Algebraic Methods and Software Technology*, M. Johnson and V. Vene, eds. Vol. 4019 of *Lecture Notes in Computer Science*, pp. 308–322. Springer, Berlin, 2006.
[31] J. Rutten. Universal coalgebra: a theory of systems. *Theoretical Computer Science*, **249**, 3–80, 2000.

[32] L. Schröder. Expressivity of coalgebraic modal logic: the limits and beyond. In *Foundations of Software Science and Computation Structures*, V. Sassone, ed. Vol. 3441 of *Lecture Notes in Computer Science*, pp. 440–454. Springer, Berlin, 2005.
[33] L. Schröder. A finite model construction for coalgebraic modal logic. *Journal of Logic and Algebraic Programming*, **73**, 97–110, 2007.

Received 28 November 2007

Appendix

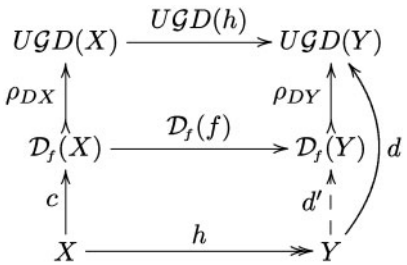
A Auxiliary Results for the Proof of Theorem 18

LEMMA 24
The mapping $\big(c\colon X\to\mathcal{D}_f(X)\big)\mapsto\big(\rho_{DX}\circ c\colon X\to U\mathcal{G}D(X)\big)$ from (21) preserves and reflects behavioural equivalence: for two states $x,x'\in X$ we have

$$x\approx x' \text{ in } c \iff x\approx x' \text{ in } X\to U\mathcal{G}DX.$$

PROOF. Behavioural equivalence preservation is direct by functoriality: if $h(x)=h(x')$ for a homomorphism $h\colon X\to Y$ from $c\colon X\to\mathcal{D}_f(X)$ to $d\colon Y\to\mathcal{D}_f(Y)$, then h is also a homomorphism from $\rho_{DX}\circ c$ to $\rho_{DY}\circ d$, by naturality of ρ.

For the reflection, assume $h(x)=h(x')$ for a coalgebra homomorphism h from $\rho_{DX}\circ c\colon X\to U\mathcal{G}D(X)$ to $d\colon Y\to Y\mathcal{G}D(Y)$. As noted before, see (7), we may assume that such an h is an abstract epi. Furthermore, the natural transformation $\rho D\colon \mathcal{D}_f=\mathcal{D}_fUD\Rightarrow U\mathcal{G}D$ is componentwise mono, resulting in the existence of the dashed arrow d' in:



This d' is obtained by diagonal fill-in. It proves that $x\approx x'$ in c. ■

For an arbitrary function f we write $\ker(f)=\{(x,x')\,|\,f(x)=f(x')\}$ for the kernel equivalence relation. We recall that a measure space $\mathcal{X}=(X,S_X)$ separates points if for different $x,x'\in X$, there exists $M\in S_X$ with $x\in M, x'\notin M$.

LEMMA 25
Let $c\colon\mathcal{X}\to\mathcal{G}(\mathcal{X})$ in **Meas** be a $\mathcal{G}$-coalgebra, with $\mathcal{X}=(X,S_X)$. Then

(1) $\mathcal{R}(S_X)\subseteq\ker(c)$, where $\mathcal{R}(S_X)=\{(x,x')\,|\,\forall M\in S_X.(x\in M\Leftrightarrow x'\in M)\}$;
(2) $\ker(c)\subseteq\approx_c$;
(3) If $\approx_c\subseteq\ker(h)$ for a homomorphism $\big(\mathcal{X}\xrightarrow{c}\mathcal{G}(\mathcal{X})\big)\xrightarrow{h}\big(\mathcal{Y}\xrightarrow{d}\mathcal{G}(\mathcal{Y})\big)$, then $\mathcal{Y}$ separates points.

PROOF. (1) Assume $(x,x') \in \mathcal{R}(S_X)$. For each $M \in S_X$ and $r \in \mathbb{Q} \cap [0,1]$, one has $L_r(M) = \{\varphi \in \mathcal{G}(\mathcal{X}) \mid \varphi(M) \geq r\} \in \mathcal{SG}(\mathcal{X})$ and so $c^{-1}(L_r(M)) \in S_X$, so that, by definition of $\mathcal{R}(S_X)$,

$$x \in c^{-1}(L_r(M)) \Leftrightarrow x' \in c^{-1}(L_r(M)) \quad \textit{i.e.} \quad c(x)(M) \geq r \Leftrightarrow c(x')(M) \geq r.$$

The latter yields $c(x) = c(x')$, and thus $(x,x') \in \ker(c)$.

(2) The coalgebra map $c\colon \mathcal{X} \to \mathcal{G}(\mathcal{X})$, is a homomorphism from c to $\mathcal{G}(c)$ in:

$$\begin{array}{ccc} \mathcal{G}(\mathcal{X}) & \xrightarrow{\mathcal{G}(c)} & \mathcal{GG}(\mathcal{X}) \\ {\scriptstyle c}\uparrow & & \uparrow{\scriptstyle \mathcal{G}(c)} \\ \mathcal{X} & \xrightarrow[c]{} & \mathcal{G}(\mathcal{X}) \end{array}$$

This shows that $\ker(c) \subseteq \approx_c$.

(3) Assume that $h\colon X \twoheadrightarrow Y$ is a homomorphism from c to d. By (1) it is enough to show that d is injective. Hence assume $d(y) = d(y')$, and write $y = h(x), y' = h(x')$ for certain $x, x' \in X$. Then:

$$\begin{array}{lll} (x,x') \in (h \times h)^{-1}(\ker(d)) & & \text{by construction} \\ \subseteq (h \times h)^{-1}(\approx_d) & & \text{by (2)} \\ \subseteq \approx_c & & \text{because } h \text{ is a homomorphism} \\ \subseteq \ker(h) & & \text{by assumption.} \end{array}$$

Hence $y = h(x) = h(x') = y'$. ■

The final step (Lemma 27) uses some basic facts about measure spaces that separate points, which we list first.

LEMMA 26
Assume $\mathcal{Y} = (Y, S_Y) \in \mathbf{Meas}$ separates points. Then

(1) For each countable $U \subseteq Y$ and $y \notin U$ there is an $N \in S_Y$ with $U \subseteq N$ and $y \notin N$.
(2) For disjoint countable $U, V \subseteq Y$ there is an $N \in S_Y$ with $U \subseteq N$ and $V \subseteq \neg N$.

PROOF. (1) For each $z \in U$ we have $z \neq y$ so that there is an $N_z \in S_Y$ with $z \in N_z$ and $y \notin N_z$. Hence we can take $N = \bigcup_z N_z \in S_Y$.

(2) For each $y \in V$ there is, by (1), an $N_y \in S_Y$ with $U \subseteq N_y$ and $y \notin N_y$. Now we can take $N = \bigcap_y N_y \in S_Y$. ■

LEMMA 27
Assume a discrete $\mathcal{G}$-coalgebra $c\colon D(X) \to \mathcal{G}D(X)$ and a surjective homomorphism h from c to some $\mathcal{G}$-coalgebra $d\colon \mathcal{Y} \to \mathcal{G}(\mathcal{Y})$. If $\mathcal{Y} = (Y, S_Y)$ separates points, then there exists a discrete $\mathcal{G}$-coalgebra $e\colon D(Y) \to \mathcal{G}D(Y)$ such that h is a homomorphism from c to e.

PROOF. We define $e\colon D(Y) \to \mathcal{G}D(Y)$ as:

$$e(y) = c(x) \circ h^{-1}$$

where $h^{-1}:\mathcal{P}(Y)\to\mathcal{P}(X)$, $y\in Y$, and $x\in X$ is such that $h(x)=y$. In order to show that e is well-defined, we must show that:

$$c(x_1)\circ h^{-1} = c(x_2)\circ h^{-1}$$

whenever $h(x_1)=h(x_2)$. Note that if e is well-defined, then it is certainly measurable and h is a homomorphism from c to e.

What we know is that if $h(x_1)=h(x_2)$, then also $(d\circ h)(x_1)=(d\circ h)(x_2)$ and using that h is a homomorphism from c to d, we get:

$$(c(x_1)\circ h^{-1})(N) = (c(x_2)\circ h^{-1})(N)$$

for all $N\in S_Y$.

So, assume $h(x_1)=h(x_2)$ and $U\subseteq Y$ arbitrary. We will construct a subset $N\in S_Y$ such that:

$$(c(x_i)\circ h^{-1})(U) = (c(x_i)\circ h^{-1})(N)$$

for $i\in\{1,2\}$ and this will complete the proof.

The subset $S=\{x\in X\mid c(x_1)(\{x\})\neq 0\vee c(x_2)(\{x\})\neq 0\}$ is countable. Write:

$$U_1 = h(S\cap h^{-1}(U)),\quad U_2 = h(S\cap h^{-1}(\neg U)).$$

Clearly, $U_1,U_2\subseteq Y$ are countable since S is. Moreover, they are disjoint: if $y\in U_1\cap U_2$, then $y=h(x_1)=h(x_2)$ for some $x_1\in h^{-1}(U)$ and $x_2\in h^{-1}(\neg U)$, which implies that $y=h(x_1)\in U$ and $y=h(x_2)\in\neg U$, a contradiction. Hence, by Lemma 26, there exists $N\in S_Y$ such that $U_1\subseteq N$ and $U_2\subseteq\neg N$. We are going to show that

$$S\cap h^{-1}(U) = S\cap h^{-1}(N).$$

Indeed, if $x\in S\cap h^{-1}(U)$, then $h(x)\in U_1\subseteq N$, so $x\in h^{-1}(N)$ and also $x\in S\cap h^{-1}(N)$. For the reverse inclusion, if $x\in S\cap h^{-1}(N)$, then we either have $x\in h^{-1}(U)$ or $x\in h^{-1}(\neg U)$. But the latter is not possible since it implies that $h(x)\in U_2$ so $h(x)\notin N$, a contradiction. Hence $x\in h^{-1}(U)$, and so $x\in S\cap h^{-1}(U)$.

We have shown that $S\cap h^{-1}(U)=S\cap h^{-1}(N)$. Now, for $i\in\{1,2\}$, we have

$$\begin{aligned} c(x_i)(h^{-1}(U)) &= \sum_{x\in S\cap h^{-1}(U)} c(x_i)(\{x\}) \\ &= \sum_{x\in S\cap h^{-1}(N)} c(x_i)(\{x\}) \\ &= c(x_i)(h^{-1}(N)) \end{aligned}$$

which completes the proof. ■

B Proof of Lemma 22

For a meet semilattice A we need to show that the mapping $\rho_{\mathcal{F}(A)}\colon \mathcal{D}_f\mathcal{F}(A)\to\mathcal{G}\mathcal{F}(A)$, given by $\rho(\Phi)=\lambda M\in\mathcal{S}\mathcal{F}(A).\,\Phi(M)$ is injective—where, as before, $\Phi(M)=\sum_{\alpha\in M}\Phi(\alpha)$. Assume therefore

$\Phi, \Psi \in \mathcal{D}_f\mathcal{F}(A)$ satisfy $\rho(\Phi) = \rho(\Psi)$. In order to show $\Phi = \Psi$ we assume an arbitrary $\alpha \in \mathcal{F}(A)$ and wish to show $\Phi(\alpha) = \Psi(\alpha)$.

Let $S = \mathrm{supp}(\Phi) \cup \mathrm{supp}(\Psi)$ be the join of the two (finite) supports. We may assume $\alpha \in S$, because otherwise $\Phi(\alpha) = 0 = \Psi(\alpha)$ and we are done. We form two subsets $B, C \subseteq A$ in the following way. For each $\beta \in S - \{\alpha\}$ we have $\beta \neq \alpha$, so that either $\exists b \in \alpha. b \notin \beta$ or $\exists c \in \beta. c \notin \alpha$. In the first case, we choose such a $b \in \alpha - \beta$ and put it in B, and in the second case we take a $c \in \beta - \alpha$ and put it in C. Since S is finite both B and C are finite (and obtained in finitely many steps). We now define $M \subseteq \mathcal{F}(A)$ as:

$$\begin{aligned} M &= \{\gamma \in \mathcal{F}(A) \mid B \subseteq \gamma \text{ and } C \cap \gamma = \emptyset\} \\ &= \left(\bigcap_{b \in B} \eta(b)\right) \cap \left(\bigcap_{c \in C} \neg\eta(c)\right). \end{aligned}$$

This second line describes M as finite intersection of measurable subsets. Hence $M \in \mathcal{S}\mathcal{F}(A)$ so that $\Phi(M) = \Psi(M)$ since $\rho(\Phi) = \rho(\Psi)$.

Next we claim:

$$M \cap S = \{\alpha\}.$$

The inclusion ($\supseteq$) is obvious by construction of B, C. For ($\subseteq$) assume $\gamma \in M \cap S$, but $\gamma \neq \alpha$. Then we have constructed either:

- a $b \in B$ with $b \in \alpha - \gamma$; this is impossible since $B \subseteq \gamma$,
- a $c \in C$ with $c \in \gamma - \alpha$. But since $C \cap \gamma = \emptyset$ this is also impossible.

Hence $\gamma = \alpha$.

We now have $\Phi(\alpha) = \Phi(\{\alpha\}) = \Phi(M \cap S) = \Phi(M) = \Psi(M) = \Psi(\alpha)$, as required. Thus $\rho_{\mathcal{F}(A)}$ is injective. ■

Theoretical Computer Science 412 (2011) 5095–5110

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Probabilistic systems coalgebraically: A survey

Ana Sokolova *

Department of Computer Sciences, University of Salzburg, Austria

ARTICLE INFO

Keywords:
Probabilistic systems
Coalgebra
Markov chains
Markov processes

ABSTRACT

We survey the work on both discrete and continuous-space probabilistic systems as coalgebras, starting with how probabilistic systems are modeled as coalgebras and followed by a discussion of their bisimilarity and behavioral equivalence, mentioning results that follow from the coalgebraic treatment of probabilistic systems. It is interesting to note that, for different reasons, for both discrete and continuous probabilistic systems it may be more convenient to work with behavioral equivalence than with bisimilarity.

1. Introduction

Probabilistic systems are models of systems that involve quantitative information about uncertainty. They have been extensively studied in the past two decades in the area of probabilistic verification and concurrency theory. The models originate in the rich theory of Markov chains and Markov processes (see e.g. [49]) and in the early work on probabilistic automata [63,61].

Discrete probabilistic systems, see e.g. [49,77,30,55,62,67,33,22,70] for an overview, are transition systems on discrete state spaces and come in different flavors: fully probabilistic (Markov chains), labeled (with reactive or generative labels), or combining non-determinism and probability. Probabilities in discrete probabilistic systems appear as labels on transitions between states. For example, in a Markov chain a transition from one state to another is taken with a given probability.

Continuous probabilistic systems, see e.g. [7,23,26,11,21,45] as well as the recent books [59,27,28] that contain most of the research on continuous probabilistic systems, are transition systems modeling probabilistic behavior on continuous state spaces. The basic model is that of a Markov process. Central to continuous probabilistic systems is the notion of a probability measure on a measurable space. Therefore, the state space of a continuous probabilistic system is equipped with a σ-algebra and forms a measurable space. It is no longer the case that the probability of moving from one state to another determines the behavior of the system. Actually, the probability of reaching any single state from a given state may be zero while the probability of reaching a subset of states is nonzero. A Markov process is specified by the probability of moving from any source state to any measurable subset in the σ-algebra, which is intuitively interpreted as the probability of moving from the source state to some state in the subset.

Both discrete and continuous probabilistic systems can be modeled as coalgebras and coalgebra theory has proved a useful and fruitful means to deal with probabilistic systems. In this paper, we give an overview of how to model probabilistic systems as coalgebras and survey coalgebraic results on discrete and continuous probabilistic systems. Having modeled probabilistic systems as coalgebras, there are two types of results where coalgebra meets probabilistic systems: (1) particular problems for probabilistic systems have been solved using coalgebraic techniques, and (2) probabilistic systems appear as popular examples on which generic coalgebraic results are instantiated. The results of the second kind are not to be considered of less importance: sometimes they lead to completely new results not known in the community of probabilistic

* Tel.: +43 69913005971; fax: +43 6628044611.
E-mail addresses: anas@cs.uni-salzburg.at, sokolova.ana@gmail.com.
URL: http://cs.uni-salzburg/~anas.

0304-3975
doi:10.1016/j.tcs.2011.05.008

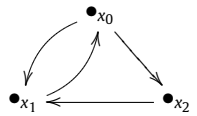
systems, e.g. [2,5,17,52,65,60]. Moreover, the variety of probabilistic systems provides a nice set of (motivating) examples for generic coalgebra results or observations, e.g. [20,60]. Also, looking at probabilistic systems from different perspectives provides evidence in favor of behavioral equivalence rather than bisimilarity.

In the paper, we take the following route. We start with an introduction to basic coalgebra notions, and some particular results concerning bisimilarity and behavioral equivalence that are needed for what follows (Section 2). Then we discuss discrete probabilistic systems (Section 3), and the inductive class of functors that turns each of them into a coalgebra on **Sets**, the category of sets and functions. We proceed with an expressiveness comparison of discrete probabilistic systems which benefits from the coalgebraic modeling, and discuss existing results of both types mentioned above on discrete probabilistic systems. We note that from this general (coalgebraic) perspective probabilistic transitions can be viewed as transitions labeled by elements of some commutative monoid, and therefore are comparable to non-deterministic transitions. We also show an alternative way of modeling (reactive) discrete probabilistic systems as coalgebras on the category of pseudometric spaces and nonexpansive functions taken by van Breugel and Worrell that allows for a definition of behavioral distances between states [15,16]. Next we move to continuous probabilistic systems (Section 4) where we show how they are modeled as coalgebras on **Meas**, the category of measurable spaces and measurable maps. Actually, in the literature, most of the time continuous probabilistic systems live on some special categories of measurable spaces (analytic, Polish, metric/pseudometric/ultrametric spaces). We discuss the reasons for this and present some observations (related to the dilemma of bisimilarity versus behavioral equivalence) that allow us to stay in **Meas**. We end our trip with a short discussion on how discrete systems are embedded into continuous systems, i.e., as expected, Markov chains are Markov processes. Please fasten your seat belt and enjoy the flight over the landscape of probabilistic systems and coalgebras.

2. Coalgebras, bisimilarity, and behavioral equivalence

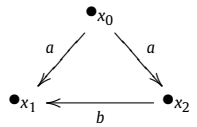
Let $\mathbb{C}$ be a category and F an endofunctor on $\mathbb{C}$. An F-coalgebra is a pair $\langle X, c\colon X \to FX\rangle$ where X in $\mathbb{C}$ is the carrier, and c is the coalgebra structure. For brevity, we often identify a coalgebra with its coalgebra structure. Given two F-coalgebras $c\colon X \to FX$ and $d\colon Y \to FY$, a coalgebra homomorphism from c to d is a map $h\colon X \to Y$ such that $d \circ h = Fh \circ c$. F-coalgebras together with their coalgebra homomorphisms form a category, denoted by Coalg_F.

In this paper we only consider coalgebras on concrete categories, i.e., categories with a faithful forgetful functor to **Sets**, the category of sets and functions. Then the carrier X of a coalgebra provides the set of states (after the application of the forgetful functor) and the coalgebra map c gives the transitions to the next state(s). The functor F determines the type of transitions. For example, coalgebras of the powerset functor $\mathcal{P}$ on **Sets** are non-deterministic transition systems in which from any state there is a set of (non-labeled) transitions to possible next states. An example of a non-deterministic transition system as coalgebra is presented below.



$$X = \{x_0, x_1, x_2\}, \quad X \xrightarrow{c} \mathcal{P}X$$
$$c(x_0) = \{x_1, x_2\}$$
$$c(x_1) = \{x_0\}$$
$$c(x_2) = \{x_1\}$$

Coalgebras of the functor $\mathcal{P}(A \times \_)$ on **Sets**, for a set of labels A, are labeled transition systems (LTS). An example is:

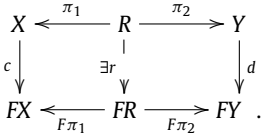


$$X = \{x_0, x_1, x_2\}, \quad X \xrightarrow{c} \mathcal{P}(A \times X)$$
$$c(x_0) = \{\langle a, x_1\rangle, \langle a, x_2\rangle\}$$
$$c(x_1) = \emptyset$$
$$c(x_2) = \{\langle b, x_1\rangle\}$$

Homomorphisms of (labeled) transition systems are transition preserving and reflecting maps. We refer the reader to [64,44] for a gentle introduction to coalgebra and many interesting examples.

A final F-coalgebra is a final object in the category Coalg_F: from any F-coalgebra c there is a unique homomorphism beh_c to the final one. If a final coalgebra exists, it induces a final coalgebra semantics which identifies two states if and only if they are mapped to the same element of the final coalgebra via the unique homomorphism. For weak pullback preserving functors on **Sets** the final coalgebra semantics coincides with coalgebraic bisimilarity defined in the following way via the notion of a bisimulation.

Let $c\colon X \to FX$ and $d\colon Y \to FY$ be two coalgebras on **Sets**. A relation $R \subseteq X \times Y$ is a bisimulation between c and d if there exists a mediating coalgebra structure $r\colon R \to FR$ making the two projections coalgebra homomorphisms, i.e., making the following diagram commute



Two states x and y are bisimilar, notation $x \sim y$, if they are related by some bisimulation relation. Weak pullback preservation of the type functor F suffices for bisimilarity to be an equivalence. Therefore, it also suffices for bisimilarity on a given coalgebra to be the union of all equivalence bisimulation relations (bisimulations that are equivalences). In order to relate coalgebraic bisimilarity to concrete notions of bisimilarity in the literature, the notion of relation lifting is helpful.

Let $R \subseteq X \times Y$ be a relation, and F a functor on **Sets**. The relation R can be lifted to a relation $\mathrm{Rel}(F)(R) \subseteq FX \times FY$ defined by

$$\langle x, y\rangle \in \mathrm{Rel}(F)(R) \;\Leftrightarrow\; \exists z \in FR.\, F\pi_1(z) = x,\; F\pi_2(z) = y.$$

It is easy to see that relation liftings provide transfer conditions for bisimulation, as stated in the next property.

Lemma 2.1. *A relation $R \subseteq X \times Y$ is a bisimulation between the F-coalgebras $c: X \to FX$ and $d: Y \to FY$ if and only if*

$$\langle x, y\rangle \in R \implies \langle c(x), d(y)\rangle \in \mathrm{Rel}(F)(R).$$

Moreover, one can show [69] the following characterization of equivalence liftings for weak pullback preserving functors.

Lemma 2.2. *If F preserves weak pullbacks and R is an equivalence on X, then $\mathrm{Rel}(F)(R)$ is the pullback of the cospan $FX \xrightarrow{Fe} F(X/R) \xleftarrow{Fe} FX$ where $e : X \to X/R$ is the canonical map mapping each element to its equivalence class.*

As a consequence we get the following characterization of equivalence bisimulations in terms of transfer conditions.

Corollary 2.3. *A relation $R \subseteq X \times X$ is an equivalence bisimulation on the F-coalgebra $c: X \to FX$, where F preserves weak pullbacks, if and only if*

$$\langle x, y\rangle \in R \implies (Fe \circ c)(x) = (Fe \circ c)(y)$$

where e is the canonical map mapping each element to its equivalence class.

We will see later how Lemmas 2.1 and 2.2, Corollary 2.3, and some properties of relation liftings of inductively defined functors provide a modular way to show that coalgebraic and concrete bisimilarity coincide for all discrete probabilistic systems.

A way to define bisimulations on general categories is using a span of morphisms. A span $\langle R, r_1: R \to X, r_2: R \to Y\rangle$ is a bisimulation between two F-coalgebras $c: X \to FX$ and $d: Y \to FY$ on a category $\mathbb{C}$ if R, r_1, and r_2 satisfy some additional conditions that ensure non-triviality (e.g. r_1, r_2 being epi) and there exists a coalgebra map $r: R \to FR$ making both r_1 and r_2 coalgebra homomorphisms. One can then define that the coalgebras c and d are bisimilar if there is a bisimulation between them. For coalgebras on a concrete category, two states $x \in X$ and $y \in Y$ are bisimilar if there exists $z \in R$ such that $x = r_1(z)$ and $y = r_2(z)$, excluding the need of additional conditions on r_1 and r_2. On **Sets** the two definitions, general span versus relation with projections, are equivalent.

Another behavior semantics, closely related to bisimilarity, called behavioral equivalence always coincides with the final coalgebra semantics. It has proven very useful in reasoning about probabilistic systems. It is based on the notion of a cocongruence [54,81] which is a cospan rather than a span.

A cocongruence between two F-coalgebras $c: X \to FX$ and $d: Y \to FY$ is a cospan $\langle U, u_1: X \to U, u_2: Y \to U\rangle$, with u_1 and u_2 jointly epi, such that there exists an F-coalgebra map $u: U \to FU$ making u_1 and u_2 coalgebra homomorphisms, i.e., making the following diagram commute

$$\begin{array}{ccccc} X & \xrightarrow{u_1} & U & \xleftarrow{u_2} & Y \\ {\scriptstyle c}\downarrow & & \downarrow{\scriptstyle \exists u} & & \downarrow{\scriptstyle d} \\ FX & \xrightarrow[Fu_1]{} & FU & \xleftarrow[Fu_2]{} & FY \end{array}.$$

We say that the coalgebras c and d are behaviorally equivalent if they are connected by a cocongruence. For coalgebras on a concrete category, we say that $x \in X$ and $y \in Y$ are behaviorally equivalent, and write $x \approx y$, if they are identified by some cocongruence between them, i.e., if there exists a cocongruence $\langle U, u_1, u_2\rangle$ with $u_1(x) = u_2(y)$.

In particular in the study of probabilistic systems coalgebraically, but also in coalgebraic modal logics in general, behavioral equivalence has advantages over bisimilarity, as we will see below. However, the good side of bisimilarity is that it is computable by efficient algorithms.[1] Another reason for working with bisimilarity is traditional, many concrete types of systems come equipped with a concrete notion of bisimilarity. Bisimilarity always implies behavioral equivalence in categories with pushouts; cf. e.g. [5,69]. If additionally the type functor F preserves weak pullbacks, then the two notions coincide (cf. e.g. [5,69]). This property is useful in comparing expressivity of different types of coalgebras.

[1] Bisimilarity can be computed by iterative algorithms, see e.g. [39], thus making it possible to automatize coinduction proofs. The plain definition of behavioral equivalence, without knowing that it coincides with bisimilarity, does not provide such methods.

3. Discrete probabilistic systems

Discrete probabilistic systems are state-based systems in which a change of state is governed by a discrete probability distribution over possible next states. In addition, one may have labels, non-determinism, and/or termination. The basic model involving discrete probabilities is a Markov chain, given by a set of states and from each state a probability distribution over the set of states, determining the probability of transiting to any other state. We start by recalling the definition of a discrete (sub)probability distribution.

Definition 3.1. Let X be a set. A function $\mu: X \to \mathbb{R}^{\geq 0}$ is a discrete probability distribution, or distribution for short, on X if $\sum_{x\in X} \mu(x) = 1$. It is a discrete subprobability distribution if $\sum_{x\in X} \mu(x) \leq 1$. The set $\{x \in X \mid \mu(x) > 0\}$ is the support of μ and is denoted by $\mathrm{supp}(\mu)$.

Note that the general X-indexed sum is defined as

$$\mu[X] = \sum_{x\in X} \mu(x) = \sup_{\substack{X'\subseteq X \\ X' \text{ finite}}} \left\{ \sum_{x\in X'} \mu(x) \right\} \in \mathbb{R} \cup \{\infty\}$$

and it is well defined since $\mu(x) \geq 0$. One can show that for any discrete probability distribution (see e.g. [69]) the support set is at most countable, which justifies the use of the term "discrete".

A distribution that assigns probability 1 to a single element $x \in X$ is called a Dirac distribution, denoted by δ_x. Hence, $\delta_x(y) = 1$ if $y = x$ and $\delta_x(y) = 0$ otherwise.

In the remainder of this section we will discuss how to model various discrete probabilistic systems as coalgebras, starting from Markov chains as the basic discrete probabilistic system type and their bisimilarity, to more complex inductively defined models and the relationship of their coalgebraic and concrete notions of bisimilarity. Moreover, we will briefly discuss an expressiveness comparison of the different discrete probabilistic systems which is made possible using the generality of coalgebra, as well as other specific and general results involving discrete probabilistic systems as coalgebras. Finally, we point out that although the research on discrete probabilistic systems is very elaborate and advanced, probabilities may not be that special: the finitary functors used for modeling discrete probabilistic systems have a generalization that allows studying and modeling more general monoid-valued valuations and not only discrete probability distributions.

3.1. Coalgebraic modeling

Almost all[2] types of known discrete probabilistic systems used as models for different verification and analysis techniques can be modeled as coalgebras on **Sets**. The main step toward coalgebraic modeling of discrete probabilistic systems is in the choice of a functor to represent discrete probability distributions over a set of states.

Definition 3.2. The probability distribution functor

$$\mathcal{D}: \mathbf{Sets} \to \mathbf{Sets}$$

maps a set X to

$$\mathcal{D}X = \{\mu: X \to \mathbb{R}^{\geq 0} \mid \mu[X] = 1\}$$

and a function $f: X \to Y$ to $\mathcal{D}f: \mathcal{D}X \to \mathcal{D}Y$ given by

$$(\mathcal{D}f)(\mu) = \lambda y.\mu[f^{-1}(\{y\})].$$

Variants of the probability distribution functor are also used, and may be convenient depending on the application. Such are the subprobability distribution (subdistribution) functor $\mathcal{D}_{\leq 1}$ and the finitely supported (sub)distribution functors $\mathcal{D}_f$ and $\mathcal{D}_{\leq 1,f}$, whose definitions vary from the definition of $\mathcal{D}$ only on objects. We have

$$\mathcal{D}_{\leq 1}X = \{\mu: X \to \mathbb{R}^{\geq 0} \mid \mu[X] \leq 1\}$$
$$\mathcal{D}_f X = \{\mu: X \to \mathbb{R}^{\geq 0} \mid \mu[X] = 1,\ \mathrm{supp}(\mu) \text{ is finite}\}$$
$$\mathcal{D}_{\leq 1,f}X = \{\mu: X \to \mathbb{R}^{\geq 0} \mid \mu[X] \leq 1,\ \mathrm{supp}(\mu) \text{ is finite}\}.$$

All these functors are well behaved in the following sense.

Proposition 3.3. *The (sub)probability distribution functor and its finitary variant preserve weak pullbacks. Moreover, each of them has a final coalgebra.*

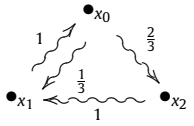
[2] All with exception of the strictly alternating systems [33], which can be modeled as multi-sorted coalgebras [60].

The proof of the weak pullback preservation of $\mathcal{D}_f$ goes back to the first coalgebraic treatment of discrete probabilistic systems by de Vink and Rutten [80], exploiting the graph-theoretical max-flow min-cut theorem as in [46], and was also shown by Moss [57], using an elementary matrix fill-in property. Similar arguments apply to $\mathcal{D}_{\leq 1,f}$. The same result is an instance of a more general result on (finitely supported) monoid valuations (also called copower) functors, by Gumm and Schroeder [32,31]. Finite support is not necessary for the weak pullback preservation of the probability distribution functor, also $\mathcal{D}$ preserves weak pullbacks [69] which is established by showing that the needed matrix fill-in property [57] can be used and holds for countably infinite matrices as well. Similarly $\mathcal{D}_{\leq 1}$ preserves weak pullbacks. The existence of a final coalgebra for $\mathcal{D}_f$ is trivial (the final coalgebra itself is trivial as well). In [80] it was proven that final coalgebra exists also for $\mathcal{D}_f + 1$ (allowing termination), using that the functor is bounded. Similar arguments apply to all variants. Also the finitely supported monoid valuation functor is bounded [68], generalizing the result on $\mathcal{D}_f$ and $\mathcal{D}_{\leq 1,f}$. We will discuss monoid valuation functors in Section 3.4 below.

Finally, we note that the (sub)distribution functors and their finitary versions are not only functors but also monads.

3.1.1. Markov chains

Discrete-time Markov chains (DTMCs) [49,40], or Markov chains for short, form the basic type of discrete probabilistic systems. They are coalgebras of the discrete probability distribution functor $\mathcal{D}$. That is, a Markov chain with a state set X is a coalgebra $c: X \to \mathcal{D}(X)$. An example is shown below where $x \overset{p}{\rightsquigarrow} y$ for states $x, y \in X$, denotes that the probability of moving from x to y is p, i.e., $c(x)(y) = p$.



$$X = \{x_0, x_1, x_2\}, \quad X \overset{c}{\to} \mathcal{D}X$$
$$c(x_0) = \{x_0 \mapsto 0, x_1 \mapsto \tfrac{1}{3}, x_2 \mapsto \tfrac{2}{3}\}$$
$$c(x_1) = \delta_{x_0}$$
$$c(x_2) = \delta_{x_1}$$

3.1.2. Probabilistic system types

We can now model most of the discrete probabilistic systems from the literature, in a modular way, using an inductively defined class of functors on **Sets**. The functors are built using the following syntax

$$F ::= \_ \mid A \mid \_^A \mid \mathcal{P} \mid \mathcal{D} \mid F \circ F \mid F \times F \mid F + F$$

where the basic functors used are the identity functor $\_$; the constant functor A mapping each set to the constant set A and each map to the identity on A; the constant exponent functor $\_^A$ mapping a set X to the set of all functions from A to X and a map $f: X \to Y$ to the map $f^A: X^A \to Y^A$ given by $f^A(g) = f \circ g$; the powerset functor $\mathcal{P}$ mapping a set to the collection of its subsets and a function $f: X \to Y$ to $\mathcal{P}(f): \mathcal{P}(X) \to \mathcal{P}(Y)$ with $\mathcal{P}(f)(X') = f(X') = \{f(x) \mid x \in X'\}$, the direct image; and the probability distribution functor $\mathcal{D}$.

An important property for some of the results presented later is weak pullback preservation. We note here that any functor in this inductively defined class preserves weak pullbacks [64,69].

This class of functors suffices to model various discrete probabilistic systems used as mathematical models of real systems for formal verification. Most of the existing probabilistic systems arose independently in the literature to improve modeling of one or another property of a system. One motivating issue was the need to model both non-deterministic and probabilistic choice, another is compositional modeling. In Fig. 1 (previously introduced in [5,69]) we present functors that allow for coalgebraic modeling of known discrete probabilistic systems (and some standard transition systems for comparison), together with the abbreviations that we use to denote the corresponding category of coalgebras, and references to papers introducing them. For some of the systems, names used here follow [70,5,69] and deviate from the original names. Such are (simple) Segala systems, also known as (simple) probabilistic automata and closely related to Markov decision processes (MDPs) [6], Vardi systems which were originally introduced as concurrent Markov chains, and Pnueli–Zuck systems that were originally named probabilistic finite-state programs.

The alternating systems considered here do not involve strict alternation as in the original definition of Hansson [33]. Strictly alternating systems can be modeled as multi-sorted coalgebras [60]. For more details on each of the discrete probabilistic systems, the reader is referred to [70,5,69]. Here we only briefly mention that both reactive and generative systems arise from LTS when replacing the non-deterministic choice modeled by $\mathcal{P}$ (in the isomorphic input view and output view on LTS) with probabilistic choice modeled by $\mathcal{D}$ and termination possibility. Some models make a distinction on the type of states, (non-)deterministic versus probabilistic, as in the case of alternating, stratified, or Vardi systems; some models involve both non-deterministic and probabilistic choices, like Segala systems or bundle systems. The last type of systems is added here in order to have a top element in the expressiveness hierarchy that we will discuss in Section 3.3.

3.2. Bisimulation correspondence

All of the concrete discrete probabilistic systems come with a concrete notion of bisimilarity defined via bisimulation equivalence in terms of transfer conditions, based on the original definition of Larsen and Skou for reactive systems [55]. The main argument that justifies generic coalgebraic results for these systems is the coincidence of the concrete notions of

Coalg_F	F	name for $X \to FX$/reference
MC	$\mathcal{D}$	Markov chains
DLTS	$(\_ + 1)^A$	deterministic automata
LTS	$\mathcal{P}(A \times \_) \cong \mathcal{P}^A$	non-deterministic automata, LTSs
React	$(\mathcal{D} + 1)^A$	reactive systems [55,30]
Gen	$\mathcal{D}(A \times \_) + 1$	generative systems [30]
Str	$\mathcal{D} + (A \times \_) + 1$	stratified systems [30]
Alt	$\mathcal{D} + \mathcal{P}(A \times \_)$	alternating systems [33]
Var	$\mathcal{D}(A \times \_) + \mathcal{P}(A \times \_)$	Vardi systems [77]
SSeg	$\mathcal{P}(A \times \mathcal{D})$	simple Segala systems [67,66]
Seg	$\mathcal{P}\mathcal{D}(A \times \_)$	Segala systems [67,66]
Bun	$\mathcal{D}\mathcal{P}(A \times \_)$	bundle systems [22]
PZ	$\mathcal{P}\mathcal{D}\mathcal{P}(A \times \_)$	Pnueli–Zuck systems [62]
MG	$\mathcal{P}\mathcal{D}\mathcal{P}(A \times \_ + \_)$	most general systems

Fig. 1. Discrete probabilistic system types.

bisimilarity with coalgebraic bisimilarity in every single case, which can be shown in a modular way using relation liftings and their properties. For the concrete bisimilarity definitions the reader is referred to [70,69] and the complete proof of bisimilarity correspondence can be found in [69]. Here we illustrate the bisimilarity correspondence proof method for the case of Markov chains and simple Segala systems.

Definition 3.4. An equivalence relation R on the set of states X of a Markov chain $c: X \to \mathcal{D}X$ is a (concrete) bisimulation if and only if $\langle x, y\rangle \in R$ implies

if $x \rightsquigarrow \mu$, then there is a distribution μ' with $y \rightsquigarrow \mu'$ and $\mu \equiv_R \mu'$

where $x \rightsquigarrow \mu$ denotes that $c(x) = \mu \in \mathcal{D}X$, and $\mu \equiv_R \mu'$ if and only if for any R-equivalence class C, $\mu[C] = \mu'[C]$.

The condition on the equivalence classes is closely related to the notion of distribution lifting [47], which is exactly the relation lifting for the probability distribution functor $\mathcal{D}$.

Definition 3.5. Let $R \subseteq X \times Y$ be a relation. Let $\mu \in \mathcal{D}(X)$ and $\mu' \in \mathcal{D}(Y)$ be distributions. Define $\mu \,\overline{R}\, \mu'$ if and only if there exists a joint distribution $\nu \in \mathcal{D}(X \times Y)$ such that

(i) ν has μ and μ' as marginals, i.e.,
  1. $\nu[x, Y] = \mu(x)$ for any $x \in X$
  2. $\nu[X, y] = \mu'(y)$ for any $y \in Y$

(ii) ν satisfies $\nu(x, y) \neq 0 \Longrightarrow \langle x, y\rangle \in R$.

It is easy to see [69] that $\mathrm{Rel}(\mathcal{D})(R) = \overline{R}$. In the special case when R is an equivalence relation on X, from Lemma 2.2, one gets that $\mathrm{Rel}(\mathcal{D})(R) = \equiv_R$, also shown directly in [48,72,1].

Hence, by Lemma 2.1, we get that an equivalence R on the set of states X of a Markov chain $c: X \to \mathcal{D}X$ is a bisimulation according to Definition 3.4 if and only if it is a coalgebraic equivalence bisimulation, showing that the concrete and the coalgebraic bisimilarity notions for Markov chains coincide. This fact was first shown by de Vink and Rutten [80]. The same technique was used by Bartels et al. [4] to sketch the correspondence of concrete bisimulation and coalgebraic bisimulation for general Segala-type systems. In [5] another, more modular proof is presented of the correspondence of concrete probabilistic bisimulation with the coalgebraic bisimulation in the case of simple Segala systems based on Corollary 2.3. At the same time, it was a proof of the correspondence for reactive systems. That technique can also be used in all the other cases. However, having Lemma 2.1 and the properties of relation liftings, it is a matter of simple, structured, and modular derivation to show the correspondence of coalgebraic and concrete bisimilarity for all of the probabilistic systems that come with a notion of bisimulation [69]. We briefly present the general method here and instantiate it to the example of simple Segala systems.

Note that bisimilarity for Markov chains is trivial, i.e., any two states in a Markov chain $c: X \to \mathcal{D}X$ are bisimilar since $X \times X$ is a bisimulation relation. This quickly changes in the presence of termination, action labels, and/or non-determinism.

The next lemma [41,69] shows that for our class of inductively defined functors, $\mathrm{Rel}(F)$ can be defined by structural induction. Item (v) repeats what we already discussed above.

Lemma 3.6. *Let $R \subseteq X \times Y$ be a relation. Then:*

(i) $\mathrm{Rel}(\_)(R) = R$,
(ii) $\mathrm{Rel}(A)(R) = \Delta_A$, *the diagonal relation on A,*
(iii) $\mathrm{Rel}(\mathcal{P})(R) = \left\{ \langle X, Y\rangle \mid (\forall x \in X)(\exists y \in Y)\langle x, y\rangle \in R \wedge (\forall y \in Y)(\exists x \in X)\langle x, y\rangle \in R \right\}$,
(iv) $\mathrm{Rel}(\_^A)(R) = \{\langle f, g\rangle \mid (\forall a \in A)\langle f(a), g(a)\rangle \in R\}$,

(v) $\mathrm{Rel}(\mathcal{D})(R) = \overline{R}$ *(see Definition 3.5),*
(vi) $\mathrm{Rel}(F \circ G)(R) = \mathrm{Rel}(F)(\mathrm{Rel}(G)(R))$,
(vii) $\mathrm{Rel}(F \times G)(R) = \{ \langle\langle x_1, x_2\rangle, \langle y_1, y_2\rangle\rangle \mid \langle x_1, y_1\rangle \in \mathrm{Rel}(F)(R) \wedge \langle x_2, y_2\rangle \in \mathrm{Rel}(G)(R) \}$,
(viii) $\mathrm{Rel}(F + G)(R) = \{\langle \kappa_1(x_1), \kappa_1(y_1)\rangle \mid \langle x_1, y_1\rangle \in \mathrm{Rel}(F)(R)\} \cup \{\langle \kappa_2(x_2), \kappa_2(y_2)\rangle \mid \langle x_2, y_2\rangle \in \mathrm{Rel}(G)(R)\}$

where κ_1, κ_2 denote the injections into the coproduct, $\kappa_i : X_i \to X_1 + X_2$.

Now we can apply Lemma 3.6 to show the correspondence of concrete and coalgebraic bisimilarity for simple Segala systems. Similar modular arguments apply to all other discrete probabilistic systems, see [69] for the complete proof. The concrete definition of bisimulation and bisimilarity for simple Segala systems [67] is the same as the original definition of bisimulation for reactive systems introduced by Larson and Skou [55], which we recall next.

Definition 3.7. An equivalence relation R on X is a (concrete) bisimulation on the simple Segala system $c : X \to \mathcal{P}(A \times \mathcal{D}X)$ if and only if $\langle x, y\rangle \in R$ implies

$$\begin{array}{l}\text{if } x \xrightarrow{a} \mu, \text{ then there exists a distribution } \mu' \text{ with } y \xrightarrow{a} \mu' \text{ and } \mu \equiv_R \mu',\\ \qquad \text{i.e., for any } R\text{-equivalence class } C,\ \mu[C] = \mu'[C]\end{array}$$

where $x \xrightarrow{a} \mu$ denotes that $\langle a, \mu\rangle \in c(x)$.

Using Lemma 2.1 we are going to provide a transfer condition for coalgebraic bisimulation between two simple Segala systems and see that in case of equivalences one gets the same transfer condition as in Definition 3.7. We have that a relation $R \subseteq X \times Y$ is a bisimulation between two simple Segala systems $c : X \to \mathcal{P}(A \times \mathcal{D}X)$ and $d : Y \to \mathcal{P}(A \times \mathcal{D}Y)$ if and only if

$$\langle x, y\rangle \in R \implies \langle c(x), d(y)\rangle \in \mathrm{Rel}(\mathcal{P}(A \times \mathcal{D}))(R).$$

Using the modular properties of relation liftings, by Lemma 3.6(vi),

$$\langle c(x), d(y)\rangle \in \mathrm{Rel}(\mathcal{P}(A \times \mathcal{D}))(R)$$

if and only if

$$\begin{array}{l}\big(\forall\langle a, \mu\rangle \in c(x), \exists\langle a', \mu'\rangle \in d(y) : \langle\langle a, \mu\rangle, \langle a', \mu'\rangle\rangle \in \mathrm{Rel}(A \times \mathcal{D})(R)\big) \wedge\\ \big(\forall\langle a', \mu'\rangle \in d(y), \exists\langle a, \mu\rangle \in c(x) : \langle\langle a, \mu\rangle, \langle a', \mu'\rangle\rangle \in \mathrm{Rel}(A \times \mathcal{D})(R)\big)\end{array}$$

which by Lemma 3.6(vii) is equivalent to

$$\begin{array}{l}\big(\forall\langle a, \mu\rangle \in c(x), \exists\langle a', \mu'\rangle \in d(y) : \langle a, a'\rangle \in \mathrm{Rel}(A)(R) \wedge \langle \mu, \mu'\rangle \in \mathrm{Rel}(\mathcal{D})(R)\big) \wedge\\ \big(\forall\langle a', \mu'\rangle \in d(y), \exists\langle a, \mu\rangle \in c(x) : \langle a, a'\rangle \in \mathrm{Rel}(A)(R) \wedge \langle \mu, \mu'\rangle \in \mathrm{Rel}(\mathcal{D})(R)\big).\end{array}$$

Applying Lemma 3.6(ii), (v), we get the following equivalent condition

$$\big(\forall\langle a, \mu\rangle \in c(x), \exists\langle a', \mu'\rangle \in d(y) : a = a' \wedge \mu\overline{R}\mu'\big) \wedge \big(\forall\langle a', \mu'\rangle \in d(y), \exists\langle a, \mu\rangle \in c(x) : a = a' \wedge \mu\overline{R}\mu'\big).$$

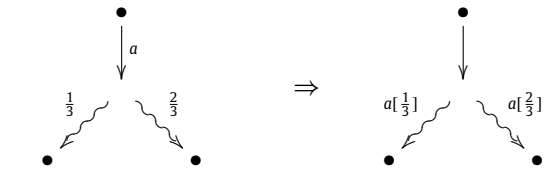
Finally, we rewrite the last condition using transition notation and obtain the transfer condition for bisimulation between simple Segala systems:

$$\begin{array}{l}\text{if } x \xrightarrow{a} \mu, \text{ then there exists } \mu' \text{ with } y \xrightarrow{a} \mu' \text{ and } \mu\overline{R}\mu', \text{ and}\\ \text{if } y \xrightarrow{a} \mu', \text{ then there exists } \mu \text{ with } x \xrightarrow{a} \mu \text{ and } \mu\overline{R}\mu'.\end{array} \tag{1}$$

If we restrict to coalgebraic bisimulations R on a simple Segala system which are equivalence relations, then as mentioned above $\overline{R} = \equiv_R$. In addition, when R is an equivalence the symmetric part of the transfer condition (1) becomes unnecessary. Hence, the transfer condition (1) is equivalent to the transfer condition of Larsen and Skou [55] (Definition 3.7) for an equivalence relation R on a simple Segala system, showing that coalgebraic and concrete bisimilarity coincide.

3.3. Expressiveness hierarchy

Having modeled all discrete probabilistic systems as coalgebras, we can now compare their expressiveness using a single coalgebraic result [5,69]. The question is: when can we consider one type of systems at least as expressive as another? We will soon define expressiveness precisely, its intuitive meaning being that a more expressive type can model all systems of a less expressive type. According to this intuition, it is clear that stratified systems are at least as expressive as Markov chains and Vardi systems are at least as expressive as LTS since in both cases the latter class is somehow contained in the former one. It is also clear that general Segala systems are at least as expressive as simple Segala systems since each simple Segala system can be considered a general Segala system after pushing each label into the target distribution, as in the example below.



a simple Segala transition $X \to \mathcal{P}(A \times \mathcal{D}X)$ as a Segala transition $X \to \mathcal{P}\mathcal{D}(A \times X)$

The expressiveness criterion chosen in [5,70,69], considers a class of systems embedded in another class of systems, in which case the latter is considered at least as expressive as the former, if there exists a translation $\mathcal{T}$ that maps any system of the former class into a system of the latter class keeping the same states such that bisimilarity is both preserved and reflected, i.e., two states are bisimilar in the original system if and only if they are bisimilar in the translated one. Since bisimilarity in all cases coincides with coalgebraic bisimilarity and the systems are modeled as coalgebras, we present a coalgebraic way of creating such translations.

We use translations of F-coalgebras into G-coalgebras in order to compare the expressiveness of coalgebras for different functors F and G. Such a translation can easily be obtained from a natural transformation between the two functors under consideration. A natural transformation $\tau: F \Rightarrow G$ is a set-indexed family of maps $\tau_X: FX \to GX$ that satisfies the naturality condition: for any map $f: X \to Y$, $\tau_Y \circ Ff = Gf \circ \tau_X$. A natural transformation $\tau: F \Rightarrow G$ induces a functor $\mathcal{T}_\tau: \mathsf{Coalg}_F \to \mathsf{Coalg}_G$ defined as [64]

$$\mathcal{T}_\tau \left(X \xrightarrow{c} FX \right) = \left(X \xrightarrow{c} FX \xrightarrow{\tau_X} GX \right) \quad \text{and} \quad \mathcal{T}_\tau h = h.$$

The induced functor is a translation map that preserves homomorphisms and thus preserves bisimilarity. For reflection of bisimilarity, we impose injectivity condition on the natural transformation [5,69].

Proposition 3.8. *Let F and G be two functors on* **Sets**. *If $\tau: F \Rightarrow G$ is a natural transformation with injective components τ_X and the functor F preserves weak pullbacks, then the induced functor $\mathcal{T}_\tau: \mathsf{Coalg}_F \to \mathsf{Coalg}_G$ preserves and reflects bisimilarity.*

Interestingly, the proof of this result uses cocongruences, i.e., behavioral equivalence. One shows that if $\tau: F \Rightarrow G$ is a natural transformation with injective components, then $\mathcal{T}_\tau$ preserves and reflects behavioral equivalence (without imposing any conditions on the functors). In the proof of preservation of behavioral equivalence [5], one uses the diagonal fill-in property to show that the mediating coalgebra structure factors as τ after an F-coalgebra structure, for which the change of directions (cospan vs. span) is handy. Since behavioral equivalence and bisimilarity coincide for weak pullback preserving functors, one gets reflection of bisimilarity in case F preserves weak pullbacks. As noted above, all our functors preserve weak pullbacks, so in order to embed a class of F-coalgebras into a class of G-coalgebras all we need is a natural transformation with injective components from F to G. We write $\mathsf{Coalg}_F \to \mathsf{Coalg}_G$ if there is an embedding of F-coalgebras into G-coalgebras that preserves and reflects bisimilarity. Providing suitable natural transformations with injective components we can build the hierarchy of discrete probabilistic systems [5,69] presented in Fig. 2. Each concrete translation is strict in the sense that the translation map is not surjective. However, in general it is very difficult to argue that any arrow in the hierarchy is strict due to the nature of the embedding definition.

All used natural transformations are well-known ones, for example the natural transformation for the translation of simple Segala to general Segala systems is obtained from the strength of the distribution functor $\mathrm{st}^{\mathcal{D}}$. It is

$$\mathcal{P}\mathrm{st}^{\mathcal{D}}: \mathcal{P}(A \times \mathcal{D}) \Rightarrow \mathcal{P}\mathcal{D}(A \times \_)$$

where the strength of the distribution functor at X maps $a \in A$ and $\mu \in \mathcal{D}X$ to $st^{\mathcal{D}}_X(a, \mu) = \mu_a \in \mathcal{D}(A \times X)$ such that $\mu_a(b, x) = \mu(x)$ for $a = b \in A$ and $x \in X$, and $\mu_a(b, x) = 0$ for $a \neq b$.

3.4. Are probabilities just a special case?

Although they might seem different at first sight, the finitary powerset functor $\mathcal{P}_f$ (mapping a set to its finite subsets) and the finitely supported (sub) probability distribution functor $\mathcal{D}_f$ can be seen as instances of the same thing. There exists a more general functor, a (subfunctor of a) functor of finitary monoid valuations, that subsumes both. We start with the definition of the finitary monoid valuations functor [31,32] which has attracted quite some attention lately.

Let $\mathbf{M} = \langle M, +, 0 \rangle$ be a commutative monoid. An $\mathbf{M}$-valuation on a set X is a function $v: X \to M$ and its support is $\mathrm{supp}(v) = \{x \in X \mid v(x) \neq 0\}$.

The finitary functor of monoid valuations $\mathbf{M}_f^- : \mathbf{Sets} \to \mathbf{Sets}$ for the monoid $\mathbf{M}$ maps a set X to the set of all finitely supported valuations on X,

$$\mathbf{M}_f^X = \{v: X \to M \mid \mathrm{supp}(v) \text{ is finite } \}$$

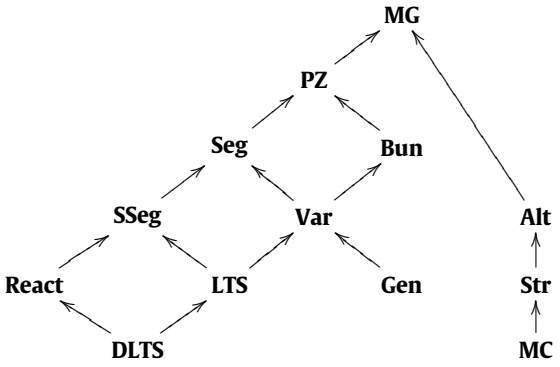


Fig. 2. Hierarchy of discrete probabilistic systems.

and a function $g: X \to Y$ to $\mathbf{M}_f^g: \mathbf{M}_f^X \to \mathbf{M}_f^Y$ given by

$$(\mathbf{M}_f^g)(v) = \lambda y.v[f^{-1}(\{y\})]$$

where for a valuation $v: X \to M$ and a subset $X' \subseteq X$ we write

$$v[X] = \sum_{x \in X'} v(x)$$

and the sum is defined due to the finite support property.

The finitely supported monoid valuation functor $\mathbf{M}_f^-$ and its properties have been studied by Gumm and Schröder [31,32], showing that the functor preserves nonempty weak pullbacks along injective maps if the monoid is positive (the only invertible element is 0), and it preserves nonempty pullbacks if additionally the monoid is refinable (if $m_1 + m_2 = n_1 + n_2$, then there exist $l_{1,1}, l_{1,2}, l_{2,1},$ and $l_{2,2}$ such that $l_{1,1} + l_{1,2} = m_1,\ l_{2,1} + l_{2,2} = m_2,\ l_{1,1} + l_{2,1} = n_1,$ and $l_{1,2} + l_{2,2} = n_2$). The same functor was also used recently by Bonchi et al. [8,68], for deriving syntax and axioms for quantitative behaviors, where the authors show that the functor is bounded and hence has a final coalgebra.

Consider the two-valued commutative monoid $\mathbf{2} = \langle \{0, 1\}, \vee, 0 \rangle$. The functor $\mathbf{2}_f^-$ coincides with the finitary powerset functor. Note that $\mathbf{2}$ is positive and refinable, so one could also derive the weak pullback preservation of the finitary powerset functor from the results of [31,32] as well as the existence of a final coalgebra from [68]. Another instance of the monoid valuation functor $\mathbf{M}_f^-$ is the finitely supported multiset functor $\mathcal{M}$ which maps a set X to the set of all finitely supported multisets on X. Namely, $\mathcal{M}X = \{m: X \to \mathbb{N} \mid \text{supp}(m) \text{ is finite } \} = \mathbf{N}_f^X$ for the monoid of natural numbers $\mathbf{N} = \langle \mathbb{N}, +, 0 \rangle$. This monoid is positive and refinable as well.

The finitely supported (sub)probability distributions functor is not exactly an instance of the finitely supported monoid valuations functor. Rather it is an instance of its subfunctor, due to the condition that each (sub)distribution assigns probability (less than or equal to) 1 to the set on which it is defined.

Let $\mathbf{M} = \langle M, +, 0 \rangle$ be a commutative monoid, and S a subset of M. The functor of finitely supported monoid valuations in S, $\mathbf{M}_{S,f}^-: \mathbf{Sets} \to \mathbf{Sets}$ maps a set X to the set

$$\mathbf{M}_{S,f}^X = \{v: X \to M \mid \text{supp}(v) \text{ is finite and } v[X] \in S\}$$

and a function in the same way as $\mathbf{M}_f^-$ does. Clearly, $\mathbf{M}_f^- = \mathbf{M}_{M,f}^-$. The functor $\mathbf{M}_{S,f}^-$ was used by Klin [50] for deriving structural operational semantics for weighted transition systems.

Consider the commutative additive monoid of non-negative real numbers $\mathbf{R} = \langle \mathbb{R}^{\geq 0}, +, 0 \rangle$ and its subsets the interval $[0, 1]$ and the singleton $\{1\}$. The functor $\mathbf{R}_{[0,1],f}^-$ coincides with the finitely supported subprobability distribution functor, whereas $\mathbf{R}_{\{1\},f}^-$ coincides with the finitely supported probability distribution functor. The monoid $\mathbf{R}$ is also positive and refinable, so $\mathbf{R}_f^-$ preserves nonempty weak pullbacks and has a final coalgebra.

Note that (sub)probability distributions have also other properties that are not explicitly captured by a functor $\mathbf{M}_{S,f}^-$. For example, for any subdistribution μ on a set X, not only $\mu[X] \in [0, 1]$ but also $\mu[X'] \in [0, 1]$ for any subset $X' \subseteq X$. In other words, any subdistribution is a discrete subprobability measure, which is not obvious from the definition of $\mathbf{M}_{S,f}^-$. In order to make this property of subdistributions explicit, one needs to highlight some additional properties of the involved monoid: the functor $\mathbf{M}_{S,f}^-$ can be specialized further as in [45] requiring that $\mathbf{M}$ be a partially ordered monoid with the property that $x \leq x + y$ for all $x, y \in M$ and S a downward-closed set.

It is interesting to note that algebraic properties of the involved monoid may have far-reaching consequences on the behavior of the coalgebras of a monoid valuations functor. For example, as noted in [45], the fact that Boolean logic with standard modalities is expressive for bisimilarity of the finitary powerset functor in contrast to finite conjunctions being

sufficient for the expressivity of probabilistic/graded modal logic for the finitary probability (sub)distribution/multiset functor, is a consequence of an algebraic property of the involved monoids: both **R** and **N** are cancellative monoids ($x + y = x + z \Rightarrow y = z$), whereas **2** is not.

An alternative precise way to model (and generalize) the distribution functor $\mathcal{D}_f$ is via valuations to effect algebras (which are partial commutative monoids with "orthosupplement") [43]. Effect algebras capture key properties of the unit interval [0, 1].

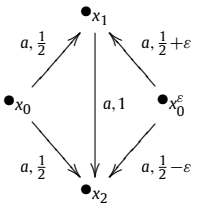
In order to capture the monad structure of both $\mathcal{P}_f$ and $\mathcal{D}_f$, additional algebraic structure is needed: instead of monoid valuations or effect-algebra valuations, one considers semiring valuations with additional properties or valuations to effect monoids [43] (effect algebras with multiplication satisfying certain properties). The unit interval [0,1] is an example of an effect monoid.

3.5. Coalgebra results on discrete probabilistic systems

In the past decade, there has been quite some research on coalgebra involving discrete probabilistic systems. Due to their variety and inductive definition, but also due to their importance in concurrency theory and verification in general, discrete probabilistic systems are popular examples in many works in the area of coalgebra. Moreover, all general results on coalgebras on **Sets** could be instantiated to probabilistic systems and in some cases they provide existing notions and results to justify the general ones, but more importantly in some cases they provide completely new results, e.g. [2,5,17, 65,52,60] that should be of interest to the probabilistic systems community. We mention a few here. In addition, there have been coalgebraic approaches to solving particular problems for probabilistic systems that are not of a generic kind, but coalgebraic notions are key ingredients in obtaining the results, as in the line of work on behavioral distances described right below.

Behavioral distances

A significant amount of work deals with behavioral distances on discrete and continuous probabilistic systems as coalgebras. We focus on the work of van Breugel and Worrell [14,13,15,16] since it is essentially coalgebraic. For more information on other work on behavioral distances we refer the reader to [15,59] that describe the situation and provide references to the literature. The main motivation behind this work is to define a quantitative analogue of bisimilarity, that will not only relate bisimilar states but provide metrics on how close the behavior of states are. The work of van Breugel and Worrell applies to reactive probabilistic systems. To start with, we borrow the following example from [15]. Consider the reactive probabilistic system below. Its states x_0 and x_0^ε



are only bisimilar if $\varepsilon = 0$. However, they behave almost the same for small ε different from 0. The authors define a pseudometric (two elements can have distance 0 even if not equal) on the states of such coalgebras, using finality in a category of coalgebras on pseudometric spaces and nonexpansive maps, for a functor describing reactive systems. Since the construction is general, i.e., it works for both continuous and discrete systems, we will get back to the underlying theory in Section 4. Here we only mention that the authors provide algorithms [13,16] for computing the behavior distances on finite-state systems, which rely on solving a particular linear programming model. Moreover, they provide a nice comparison to other existing (non-coalgebraic) notions of behavioral distance in [14].

Modal logics

The well-developed theory of coalgebraic modal logics starting from Moss, extensively expanded by Cirstea, Kurz, Pattinson, Schröder, Venema, and others, which also involves work on modular logics for inductively defined functors, frequently employs discrete probabilistic systems as examples; cf. e.g. [57,17,18,20,19,65,60]. Many of the general results when instantiated to probabilistic systems provide new insight or results that the probabilistic systems community may not be aware of, e.g. [17,65,60]. The value of the results on coalgebraic modal logics is inverse proportional to the space devoted here, the main reason for such a treatment being that more information on coalgebraic modal logics can be found in the survey article by Kupke and Pattinson [53] in this very issue.

Structural operational semantics

Coalgebra theory leads to significant results on discrete probabilistic systems in structural operational semantics. Bartels [2,3] instantiated the general categorical framework of bialgebras for structural operational semantics by Turi and Plotkin [73] to reactive probabilistic systems and simple Segala systems, providing the first formats for structural

operational semantics that guarantee that the operations are well defined and well behaved. The latter means that bisimilarity/behavioral equivalence is a congruence for each operation defined by rules of the given format. Recently in the work of Klin and Sassone [52] the same framework was instantiated to stochastic systems. More information on structural operational semantics and the work mentioned here can be found in the survey article by Klin [51] in this issue. Worth noting here is that stochastic transition systems, describing models like labeled CTMCs, are reactive versions of coalgebras of the monoid valuation functor $\mathbf{R}_f^-$, namely they are coalgebras of the functor $(\mathbf{R}_f^-)^A$.

Traces

We mentioned before that $\mathcal{D}$ (and each of its variants) is not only a functor but also a monad modeling probabilistic choice. The monad structure is important, e.g. for linear-time trace semantics. The coalgebraic trace theory of Hasuo et al. [37] applies to TF-coalgebras on **Sets** where T is a monad modeling branching, and F is a functor modeling linear behavior. Under certain conditions on T and F, the functor F lifts to the Kleisli category of T (with objects sets and morphisms $f: X \to Y$ being functions $f: X \to TY$), where branching is hidden. The main result of [37] shows that, under additional order-theoretic conditions on T and F, the initial F-algebra in **Sets** is a final coalgebra of the lifting of F in the Kleisli category. The final coalgebra semantics in the Kleisli category provides trace semantics for TF-coalgebras. The theory is applicable to the subprobability distribution monad $\mathcal{D}_{\leq 1}$ (but not to the variants) and the functor $F = 1 + A \times \_$ resulting in the usual trace semantics for generative probabilistic systems.

The powerset functor $\mathcal{P}$ is also a monad, modeling non-deterministic choice. However, the combination $\mathcal{P}\mathcal{D}$ is unfortunately not a monad [76,75] (due to nonexistence of a required distributive law). In the work of Varacca [76,75], the problem of constructing a monad for non-determinism and probability is addressed and two solutions are proposed: either to replace the distribution monad by a new monad of indexed valuations, which combined with the powerset provides a monad due to the existence of suitable distributive law, or to use one monad of convex subsets, on a different category, for the whole combination. The latter monad was recently used to describe traces of simple Segala systems by Jacobs [42].

Recent work on generic forward and backward simulations by Hasuo [34,36] also instantiates to generative probabilistic systems, providing new notions of forward and backward simulations for these systems. Forward and backward simulations are sound in the sense that they imply trace inclusion and are therefore useful as a proof method for trace equivalence. An interesting application of this method to probabilistic anonymity and probable innocence can be found in the work of Hasuo et al. [38,35].

Kleene algebras

A recent line of research in coalgebra by Silva, Bonchi et al., focuses on deriving languages of generalized regular expressions, and their sound and complete axiomatizations, for transition systems modeled as coalgebras, generalizing the results of Kleene, on regular languages and deterministic finite automata, and Milner, on regular behaviors and finite LTS. This work focuses on an inductively defined class of functors which involves the monoid valuation functor and also leads to results for discrete probabilistic systems [8,68].

Weak bisimulation

A long open problem in coalgebra theory is the problem of a coalgebraic characterization of weak bisimulation. One of the few approaches toward this [71] is actually inspired by (and somewhat tailored to) concrete work on probabilistic weak bisimulation. The approach is to transform a given system to its "double-arrow" system whose transitions are all "weak transitions" of the original system, and then define weak bisimulation as a bisimulation on the transformed system. The paper lists some required properties of such a transformation, but no generic construction is identified that actually provides the transformation. One can come up with a suitable transformation for generative probabilistic systems and for LTS.

4. Continuous probabilistic systems

By continuous probabilistic systems we mean probabilistic systems on continuous state spaces. The notion of interest is no longer a (discrete) probability distribution, but a (continuous) probability measure. As explained well by Panangaden [59], the point is that the elements of the space are no longer the "atoms" of the measure, each element may have probability zero and yet a subset may have a nonzero probability. The work on continuous probabilistic systems is in large part coalgebraic, or more precisely it is categorical and coalgebra-aware. Most of the work on continuous probabilistic systems is due to Desharnais, Panangaden [59] et al., on labeled Markov processes, to Doberkat [27,28] on stochastic relations, and to van Breugel and Worrell [15] on behavioral distances. Our aim here is to present a very brief, and as a consequence somewhat shallow, overview of continuous probabilistic systems as coalgebras. Coalgebraic treatment of continuous probabilistic systems also originates in the work of de Vink and Rutten [80], on the category of ultrametric spaces and nonexpansive functions. We start with the basic definition of a measurable space, measurable function, and (sub)probability measure.

A measurable space $\mathcal{X} = \langle X, S_X \rangle$ is a pair of a set X and a σ-algebra S_X on X, i.e., a collection of subsets of X, $S_X \subseteq \mathcal{P}X$ with the properties

(1) $\emptyset \in S_X$,
(2) $S \in S_X \Longrightarrow X \setminus S \in S_X$, and
(3) $\bigcup_i S_i \in S_X$, for any countable family of measurable sets $S_i \in S_X$.

The elements of the σ-algebra are called measurable sets.

Let $\mathcal{X} = \langle X, S_X\rangle$ and $\mathcal{Y} = \langle Y, S_Y\rangle$ be measurable spaces. A measurable function from $\mathcal{X}$ to $\mathcal{Y}$ is any function $f: X \to Y$ with the property that inverse image of a measurable set is a measurable set, i.e., $f^{-1}(S_Y) \subseteq S_X$.

A probability measure on $\mathcal{X}$ is a function $\mu: S_X \to [0, 1]$ with the properties that

(1) $\mu(\emptyset) = 0$,
(2) $\mu(X) = 1$,
(3) $\mu(\bigcup_i S_i) = \sum_i \mu(S_i)$, for any countable family of pairwise-disjoint measurable sets $S_i \in S_X$.

A function $\mu: S_X \to [0, 1]$ is a subprobability measure on $\mathcal{X}$ if it satisfies the properties (1) and (3), which means that the measure of the whole space is less than or equal to 1 but not necessarily 1.

Definition 4.1. A Markov process on a measurable space $\mathcal{X} = \langle X, S_X\rangle$ is a pair $\langle X, (\mu_x)_{x\in X}\rangle$ with X being the state set and, for each $x \in X$, $\mu_x: S_X \to [0, 1]$ is a transition subprobability measure, i.e., a subprobability measure with the additional property that for each $S \in S_X$ the function $m_S: X \to [0, 1]$ given by $m_S(x) = \mu_x(S)$ is a measurable function from $\mathcal{X}$ to the measurable space $[0, 1]$ with the σ-algebra of Borel sets, the smallest σ-algebra containing all open sets.

Hence, a Markov process is a continuous-space transition system. Given a Markov process on $\mathcal{X}$, one interprets $\mu_x(S)$ as the probability that the system starting in state x makes a transition to one of the states in S. As in the case of Markov chains, labels make the behavior of Markov processes more interesting, leading to labeled Markov processes. Labels are reactive, i.e., in a labeled Markov process on $\mathcal{X} = \langle X, S_X\rangle$, for each $x \in X$ and each label $a \in A$, $\mu_{x,a}: S_X \to [0, 1]$ is a transition subprobability measure. We borrow the following example from [23,59].

Consider a process with two labels a, b. The state space is the real plane $\mathbb{R}^2$. When the process makes an a-move from state (x_0, y_0), it jumps to (x, y_0), where the probability distribution for x is given by the density $K_\alpha \exp(-\alpha(x-x_0)^2)$, where $K_\alpha = \sqrt{\alpha/\pi}$ is the normalizing factor. When it makes a b-move it jumps from state (x_0, y_0) to (x_0, y), where the distribution for y is given by the density function $K_\beta \exp(-\beta(y - y_0)^2)$. The meaning of these densities is as follows. The probability of jumping from (x_0, y_0) to a state with x-coordinate in the interval $[s, t]$ under an a-move is $\int_s^t K_\alpha \exp(-\alpha(x - x_0)^2)dx$. Note that the probability of jumping to any given point is, of course, 0. In this process the interaction with the environment controls whether the jump is along the x-axis or along the y-axis but the actual extent of the jump is governed by a probability distribution. With a single label, the process amounts to an ordinary (time-independent) Markov process.

We refer the reader to [23–25,59] for more interesting examples and a very good explanation of the importance of modeling and verifying such systems.

4.1. Coalgebraic modeling

In order to model Markov processes as coalgebras one needs a suitable category and a suitable functor. The most natural choice for a category is **Meas**, the category of measurable spaces and measurable functions. For different reasons, some of which are explained in Section 4.2, in most of the work on Markov processes different categories were considered. In this brief survey we mainly remain in **Meas** and argue that for coalgebraic treatment of Markov processes it suffices to work with general measurable spaces unless one prefers bisimilarity to behavioral equivalence. In addition, we explain how probabilistic systems are modeled in categories of metric spaces for the purpose of studying behavioral distances.

No matter which category is considered, all works on continuous probabilistic systems agree on the functor: the Giry functor (actually monad) [29]. The initial idea to look for such a monad goes back to unpublished work by Lawvere [56].

Definition 4.2. Given a measurable space $\mathcal{X} = \langle X, S_X\rangle$ the Giry functor

$$\mathcal{G}: \mathbf{Meas} \to \mathbf{Meas}$$

maps $\mathcal{X}$ to the measurable space

$$\mathcal{G}\mathcal{X} = \langle \mathcal{G}_\mathcal{X}, \mathcal{S}\mathcal{G}\mathcal{X}\rangle$$

where $\mathcal{G}_\mathcal{X}$ is the set of subprobability measures on $\mathcal{X}$ and $\mathcal{S}\mathcal{G}\mathcal{X}$ is the smallest σ-algebra making all evaluation maps ev_S, for $S \in S_X$, measurable, where for $S \in S_X$, the evaluation map $ev_S: \mathcal{G}_\mathcal{X} \to [0, 1]$ is given by $\mu \mapsto \mu(S)$.

A morphism $f: \mathcal{X} \to \mathcal{Y}$, i.e., a measurable function $f: X \to Y$ is mapped to $\mathcal{G}f: \mathcal{G}\mathcal{X} \to \mathcal{G}\mathcal{Y}$ where $\mathcal{G}f(\mu) = \mu \circ f^{-1}$.

As noticed in [45,58], the σ-algebra $\mathcal{S}\mathcal{G}\mathcal{X}$ is generated by the collection:

$$\left\{L_r(S) \mid r \in \mathbb{Q} \cap [0, 1], S \in S_X\right\}$$

where

$$L_r(S) = \{\mu \in \mathcal{G}_\mathcal{X} \mid \mu(S) \geq r\} = ev_S^{-1}([r, 1])$$

which are the usual probabilistic modalities, providing an intrinsic connection between probabilistic modal logics and Giry coalgebras.

Moss and Viglizzo [58] show that every functor on the category **Meas** built from the identity and constant functors using products, coproducts, and the Giry functor has a final coalgebra. The construction uses modal logics: the elements of the final coalgebra are theories (sets of modal formulas) satisfied by states in all possible coalgebras. Viglizzo [79] also provided another construction of a final coalgebra for the same class of functors, avoiding the logic.

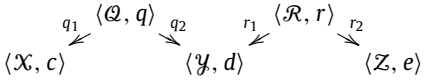
Proposition 4.3. *Markov processes are exactly the $\mathcal{G}$-coalgebras on* **Meas**.

The proposition holds since, by the construction of the σ-algebra $\mathcal{S}\mathcal{G}\mathcal{X}$ for $\mathcal{X} = \langle X, S_X\rangle$, we have that a function $c: X \to \mathcal{G}_\mathcal{X}$ is measurable if and only if $ev_S \circ c$ is measurable for all $S \in S_X$. Therefore, $c: \mathcal{X} \to \mathcal{G}\mathcal{X}$ is a Giry coalgebra if and only if $\langle X, (\mu_x)_{x\in X}\rangle$ with $\mu_x = c(x)$ is a Markov process, since $ev_S \circ c = m_S$.

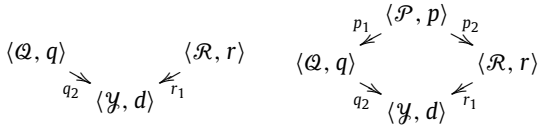
A large part of the research on continuous probabilistic systems focuses on stochastic relations and stochastic coalgebraic logic [27,28]. A stochastic relation in **Meas** is a Kleisli morphism of the Giry monad, i.e., a map $s: \mathcal{X} \to \mathcal{G}\mathcal{Y}$. Hence, every Markov process (a Giry coalgebra on **Meas**) is a stochastic relation (for $\mathcal{X} = \mathcal{Y}$), but not the other way around. Most of the research on stochastic relations is located in other categories (analytic or Polish spaces) than general measurable spaces, for reasons that we will discuss in the following section.

4.2. Bisimilarity problems and solutions

It is very difficult to show that bisimilarity is an equivalence for Markov process, in particular it is difficult to show that it is transitive [59,27,21]. The reason is the following. Assume $c \sim d$ and $d \sim e$ for three Markov processes, coalgebras on **Meas**, $c: \mathcal{X} \to \mathcal{G}\mathcal{X}$, $d: \mathcal{Y} \to \mathcal{G}\mathcal{Y}$ and $e: \mathcal{Z} \to \mathcal{G}\mathcal{Z}$. Let $\langle\langle \mathcal{Q}, q\rangle, q_1, q_2\rangle$ be a witnessing bisimulation for $c \sim d$ and $\langle\langle \mathcal{R}, r\rangle, r_1, r_2\rangle$ for $d \sim e$. Hence we have the following situation:



Then, for transitivity, it would suffice to complete the following cospan on the left, to a square on the right.



which turns out to be difficult for Markov processes. The category **Meas** has pullbacks, so if the coalgebra functor would preserve (weak) pullbacks then one can complete the square. However, as shown by Viglizzo [78], the Giry monad on **Meas** does not preserve weak pullbacks.

A large amount of research dealt with this problem and in all cases the shift to other categories was taken. In the first coalgebraic treatment of Markov processes by de Vink and Rutten [80], the category of ultrametric spaces and nonexpansive maps was considered. The main motivation for doing so was reusing a theorem that guarantees existence of a final coalgebra for locally contractive functors on ultrametric spaces [74]. The authors mention that it would be good to have weak pullback preservation which would result in a full-abstractness result implying that bisimilarity is well behaved, but leave the issue for future work. A very involved construction [23] of a so-called semi-pullback provides a way to complete the cospan above to a square, for (labeled) Markov processes on analytic spaces, showing that bisimilarity is transitive. This result was followed by a deep analysis of semi-pullbacks by Doberkat [26,27] for stochastic relations and thus Markov processes on Polish and analytic spaces. None of this proves that bisimilarity of (labeled) Markov processes on **Meas** is not transitive, but it is quite likely so, as conjectured already in [23].

Finally, we note that there is no problem with behavioral equivalence: it is always an equivalence in categories with pushouts, as is the case with **Meas**. This fact was explicitly recognized first by Danos et al. [21] where cocongruences (event bisimulations) were considered for (labeled) Markov processes. Event bisimulations provide a concrete definition of behavioral equivalence.

4.3. Comments on modal logics for continuous probabilistic systems

A significant part of this work is related to modal logic, in particular the first result showing that negation-free probabilistic modal logic (with finite conjunctions) is expressive for bisimilarity of probabilistic systems comes from the work on labeled Markov processes [7,23,59] on Polish/analytic spaces. In the context of stochastic relations, different bisimulation notions are considered in order to show expressivity of probabilistic modal logic with finite conjunctions, again on Polish/analytic spaces [27,28]. Expressivity, also called Hennessy–Milner property, of a logic for bisimilarity means that two states are bisimilar if and only if they satisfy the same formulas.

Danos et al. [21] showed that negation-free probabilistic modal logic (with finite conjunctions) is expressive for behavioral equivalence of Markov processes on **Meas**. Recently [45], Jacobs and the author of this paper have presented another proof of the expressivity of probabilistic modal logic with finite conjunctions (and no negations) with respect to behavioral equivalence for Markov processes as coalgebras of the Giry functor on **Meas**, by providing a dual adjunction between **Meas** and **MSL**, the category of meet-semilattices. The same paper also provides a proof of the expressivity of the same logic for Markov chains, via a related dual adjunction between **Sets** and **MSL**.

4.4. Behavioral distances via finality: metric vs. measurable spaces

The work on behavioral pseudometrics for probabilistic systems by van Breugel and Worrell; cf. e.g. [14,15], builds up on the work of de Vink and Rutten [80] in the sense that continuous (and as a special case also discrete) probabilistic systems are modeled as coalgebras of a variant of the Giry functor on a category of 1-bounded pseudometric spaces.

A 1-bounded pseudometric space is a pair $\langle X, d_X\rangle$ where X is a set and $d_X : X \times X \to [0, 1]$ is a pseudometric satisfying the symmetry condition and the triangle inequality. It is 1-bounded since all distances are bounded by 1, and pseudo since different elements may have distance 0. 1-bounded pseudometric spaces and nonexpansive maps (functions that do not increase distances) form a category. The authors show the existence of a final coalgebra for locally contractive functors in this category, by slightly generalizing the result of [74].

Every (pseudo)metric space is a measurable space when equipped with the Borel σ-algebra, the smallest σ-algebra containing all open sets. In order to model probabilistic systems as coalgebras on the category of pseudometric spaces, the following definition yields a functor M: it maps a metric space $\langle X, d_X\rangle$ to the set of all (tight) Borel probability measures on X, with the Hutchinson metric. On functions, M is defined just like the Giry functor. In order to model probabilistic systems, the authors modify the functor to include (reactive) labels and a so-called discount factor $c \in (0, 1)$, resulting in a functor P. The discount factor ensures that the functor P is locally contractive, and intuitively it discounts the future: the smaller the discount factor, the more the future is discounted. As a consequence, P has a final coalgebra with carrier $\Omega = \langle fix(P), d_{fix(P)}\rangle$. The space Ω is a compact metric space.

Now, by finality, this induces a metric on the states on any other P-coalgebra, as follows. Let $\mathcal{X} = \langle X, d_X\rangle$ be a 1-bounded pseudometric space and $c : \mathcal{X} \to P\mathcal{X}$ a P-coalgebra. Let $\varphi : \mathcal{X} \to \Omega$ be the unique homomorphism obtained by finality. For any two states $x, y \in X$ the behavioral distance from x to y, also called coalgebraic distance, is defined by

$$d(x, y) = d_{fix(P)}(\varphi(x), \varphi(y)).$$

The obtained behavior distance is a pseudometric: bisimilar states have distance 0. Moreover, states have distance 0 if and only if they are bisimilar. As explained by the authors [15]: the distance between states is a trade-off between the depth of observations needed to distinguish the states and the amount each observation differentiates the states.

The authors present several useful characterizations of the introduced coalgebraic pseudometric, and a comparison to existing (non-coalgebraic) behavior metrics of probabilistic systems. In particular they show that the coalgebraic pseudometric coincides (up to the discount factor) with the first behavioral pseudometric, the logical pseudometric of Desharnais et al. [25]. They also present an algorithm for computing behavioral distances on finite systems based on a linear programming problem [13,16]. Moreover, the coalgebraic distance can be approximated, i.e., the finality homomorphism φ is a fixpoint of a certain function and can be computed by a sequence of approximations φ_n each inducing a pseudometric d_n on the states of a P-coalgebra that approximate the pseudometric d. To calculate d with a prescribed degree of accuracy α, the authors show that it suffices to calculate d_i for $i = 1, \ldots, \log_c(\alpha/2)$.

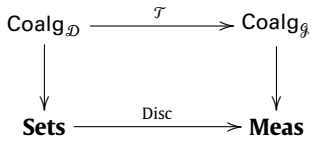
Furthermore, van Breugel et al. characterize approximate bisimilarity independently of the Hutchinson metric (independently of integration) in [10], and in [9] provide a more general final coalgebra theorem for accessible categories and accessible functors, that subsume the categories of coalgebras studied before and cover the case of no discount ($c = 1$). In a later work [12], van Breugel et al. present an algorithm for approximating the pseudometric also in case of systems that do not discount the future ($c = 1$).

4.5. Embedding discrete into continuous systems

As expected, Markov chains embed into Markov processes. Clearly, any discrete probability distribution μ on a set X extends to a probability measure, $\overline{\mu}$ on the discrete measurable space $\mathcal{X} = \langle X, \mathcal{P}X\rangle$ with the discrete σ-algebra of all subsets. We have $\overline{\mu}(X') = \mu[X']$. Therefore, any Markov chain "is" a Markov process.

However, Markov chains and Markov processes live in different categories. For a precise embedding, in line with the translation embeddings of Section 3.3, more machinery is needed. As shown in [45], there is a translation functor $\mathcal{T}$ that maps a Markov chain to a Markov process on the same state set so that behavioral equivalence is preserved and reflected. The functor is induced by a suitable injective natural transformation. It is non-trivial to show that behavioral equivalence on a Markov chain $c : X \to \mathcal{D}X$ and on the corresponding discrete Markov processes $\mathcal{T}(c)$ coincide [45] providing an embedding as in Section 3.3 from chains into processes.

The situation is as follows



where Disc denotes the functor from **Sets** to **Meas** mapping a set X to the discrete measurable space on X, and the vertical arrows represent forgetful functors mapping a coalgebra to its carrier. The functor Disc is a left adjoint of the forgetful functor from **Meas** to **Sets** mapping a measurable space $\mathcal{X} = \langle X, S_X \rangle$ to the set X. The adjunction plays an important role in the definition of $\mathcal{T}$.

Following this embedding, it may be of interest to build another floor in the hierarchy of probabilistic systems for different (labeled) continuous probabilistic systems on the category **Meas**.

5. Conclusions

We have presented a brief survey of discrete and continuous probabilistic systems as coalgebras. Via the probability distribution functor on **Sets** and the Giry functor on **Meas** (and related categories) probabilistic systems enter coalgebra. Discrete systems are inductively built and therefore present a nice class of examples for other research in coalgebra theory. Treating continuous systems is not so straightforward and requires moving to other categories than **Meas** in order that bisimilarity is an equivalence. A solution is to consider behavioral equivalence instead of bisimilarity.

Another fruitful direction in the coalgebraic treatment of probabilistic systems that we have briefly highlighted is through metric spaces, providing behavioral distances between states rather than equivalence relations on states. This work is significant for two reasons: (1) behavioral pseudometrics provide not only information about whether states of a probabilistic system behave in the same way or not, but also quantitative information about how close or how far apart the behavior of such states are; (2) part of the work on behavioral metrics is essentially coalgebraic, the pseudometrics being defined via finality.

This survey is already quite long, and we have just flown over the topic of coalgebraic treatment of probabilistic systems. Aware that in many respects we lack in explanation and detail, we hope that at least references to the literature may guide you to your particular destination topic of interest.

Acknowledgements

I am grateful to the organizers of CMCS 2010 for providing me an opportunity to give a talk and publish this work. Furthermore, I would like to thank Dirk Pattinson for discussions and pointers to references, Lutz Schröder for a comment during my talk at CMCS 2010, a prompt answer to a question, and just-in-time pointers to the literature, and Mathieu Tracol for an informative discussion on behavioral pseudometrics. Last but not least, I thank the anonymous referees for their thorough reviews, feedback, and very useful comments. The author's work is supported by the Austrian Science Funds (FWF) Project V00125.

References

[1] C. Baier, On algorithmic verification methods for probabilistic systems, Habilitationsschrift, FMI, Universitaet Mannheim, 1998.
[2] F. Bartels, GSOS for probabilistic transition systems, in: Proc. CMCS'02, Electronic Notes in Theoretical Computer Science 65 (1) (2002).
[3] F. Bartels, On generalised coinduction and probabilistic specification formats: distributive laws in coalgebraic modelling. Ph.D. Thesis, Vrije Universiteit, Amsterdam, 2004.
[4] F. Bartels, A. Sokolova, E.d. Vink, A hierarchy of probabilistic system types, in: Proc. CMCS'03, Electronic Notes in Theoretical Computer Science 82 (1) (2003).
[5] F. Bartels, A. Sokolova, E.d. Vink, A hierarchy of probabilistic system types, Theoretical Computer Science 327 (2004) 3–22.
[6] R. Bellman, A Markovian decision process, Journal of Mathematics and Mechanics 6 (1957) 679–684.
[7] R. Blute, J. Desharnais, A. Edalat, P. Panangaden, Bisimulation for labelled Markov processes, in: Proc. LICS, IEEE, 1997, pp. 149–158.
[8] F. Bonchi, M. Bonsangue, J. Rutten, A. Silva, Deriving syntax and axioms for quantitative regular behaviours, in: Proc. CONCUR, in: LNCS, vol. 5710, 2009, pp. 146–162.
[9] F.v. Breugel, C. Hermida, M. Makkai, J. Worrell, An accessible approach to behavioural pseudometrics, in: Proc. ICALP, in: LNCS, vol. 3580, 2005, pp. 1018–1030.
[10] F.v. Breugel, M. Mislove, J. Ouaknine, J. Worrell, An intrinsic characterization of approximate probabilistic bisimilarity, in: Proc. FoSSaCS, in: LNCS, vol. 2620, 2003, pp. 200–215.
[11] F.v. Breugel, M. Mislove, J. Ouaknine, J. Worrell, Domain theory, testing and simulation for labelled Markov processes, Theoretical Computer Science 333 (2005) 2239–2259.
[12] F.v. Breugel, B. Sharma, J. Worrell, Approximating a behavioural pseudometric without discount for probabilistic systems, in: Proc. FoSSaCS, in: LNCS, vol. 4423, 2007, pp. 123–137.
[13] F.v. Breugel, J. Worrell, An algorithm for quantitative verification of probabilistic transition systems, in: Proc. CONCUR, in: LNCS, vol. 2154, 2001, pp. 336–350.
[14] F.v. Breugel, J. Worrell, Towards quantitative verification of probabilistic transition systems, in: Proc. ICALP, in: LNCS, vol. 2076, 2001, pp. 421–432.
[15] F.v. Breugel, J. Worrell, A behavioural pseudometric for probabilistic transition systems, Theoretical Computer Science 331 (1) (2005) 115–142.

[16] F.v. Breugel, J. Worrell, Approximating and computing behavioural distances in probabilistic transition systems, Theoretical Computer Science 360 (1–3) (2006) 373–385.
[17] C. Cîrstea, A modular approach to defining and characterising notions of simulation, Information and Computation 204 (4) (2006) 469–502.
[18] C. Cîrstea, Modularity in coalgebra, Electronic Notes in Theoretical Computer Science 164 (1) (2006) 3–26.
[19] C. Cîrstea, A. Kurz, D. Pattinson, L. Schröder, Y. Venema, Modal logics are coalgebraic, in: BCS Int. Acad. Conf., British Computer Society, 2008, pp. 128–140.
[20] C. Cîrstea, D. Pattinson, Modular construction of complete coalgebraic logics, Theoretical Computer Science 388 (1–3) (2007) 83–108.
[21] V. Danos, J. Desharnais, F. Laviolette, P. Panangaden, Bisimulation and cocongruence for probabilistic systems, Information and Computation 204 (2006) 503–523.
[22] P. D'Argenio, H. Hermanns, J.-P. Katoen, On generative parallel composition, in: Proc. PROBMIV'98, Electronic Notes in Theoretical Computer Science 22 (1998) 105–122.
[23] J. Desharnais, A. Edalat, P. Panangaden, Bisimulation for labelled Markov processes, Information and Computation 179 (2002) 163–193.
[24] J. Desharnais, V. Gupta, R. Jagadeesan, P. Panangaden, Approximating labelled Markov processes, Information and Computation 184 (1) (2003) 160–200.
[25] J. Desharnais, V. Gupta, R. Jagadeesan, P. Panangaden, Metrics for labelled markov processes, Theoretical Computer Science 318 (3) (2004) 323–354.
[26] E.-E. Doberkat, Semi-pulbacks and bisimulations in categories of stochastic relations, in: Proc. ICALP, in: LNCS, vol. 2719, 2003, pp. 996–1070.
[27] E.-E. Doberkat, Stochastic Relations: Foundations for Markov Transition Systems, ChapmanHall, CRC, 2007.
[28] E.-E. Doberkat, Stochastic Coalgebraic Logic, Springer, 2010.
[29] M. Giry, A categorical approach to probability theory, in: Categorical Aspects of Topology and Analysis, in: LNM, vol. 915, 1982, pp. 68–85.
[30] R.v. Glabbeek, S. Smolka, B. Steffen, C. Tofts, Reactive, generative, and stratified models of probabilistic processes, in: Proc. LICS, IEEE, 1990, pp. 130–141.
[31] H.-P. Gumm, Copower functors, Theoretical Computer Science 410 (2009) 1129–1142.
[32] H.-P. Gumm, T. Schröder, Monoid-labeled transition systems, in: Proc. CMCS'01, Electronic Notes in Theoretical Computer Science 44 (1) (2001) 185–204.
[33] H. Hansson, Time and probability in formal design of distributed systems, Real-Time Safety Critical Systems 1 (1994).
[34] I. Hasuo, Generic forward and backward simulations, in: Proc. CONCUR, in: LNCS, vol. 4137, 2006, pp. 406–420.
[35] I. Hasuo, Tracing Anonymity with Coalgebras, Ph.D. Thesis, Radboud University Nijmegen, 2008.
[36] I. Hasuo, Generic forward and backward simulations II: probabilistic simulation, in: Proc. CONCUR, in: LNCS, vol. 6269, 2010, pp. 447–461.
[37] I. Hasuo, B. Jacobs, A. Sokolova, Generic trace semantics via coinduction, Logical Methods in Computer Science 3 (4:11) (2007).
[38] I. Hasuo, Y. Kawabe, H. Sakurada, Probabilistic anonymity via coalgebraic simulations, Theoretical Computer Science 411 (2007) 2239–2259.
[39] D. Hausmann, T. Mossakowski, L. Schröder, Iterative circular coinduction for cocasl in isabelle/hol, in: Proc. FASE, in: LNCS, vol. 3442, 2005, pp. 341–356.
[40] R. Howard, Dynamic Probabilistic Systems, John Wiley & Sons, Inc., New York, 1971.
[41] B. Jacobs, Exercises in coalgebraic specification, in: Algebraic and Coalgebraic Methods in the Mathematics of Program Construction, in: LNCS, vol. 2297, 2002, pp. 237–281.
[42] B. Jacobs, Coalgebraic trace semantics for combined possibilitistic and probabilistic systems, in: Proc. CMCS'08, Electronic Notes in Theoretical Computer Science 203 (5) (2008) 131–152.
[43] B.P.F. Jacobs, Probabilities, distribution monads, and convex categories, Theoretical Computer Science (2011) (doi:10.1016/j.tcs.2011.04.005).
[44] B. Jacobs, J. Rutten, A tutorial on (co)algebras and (co)induction, Bulletin of the EATCS 62 (1996) 222–259.
[45] B. Jacobs, A. Sokolova, Exemplaric expressivity of modal logics, Journal of Logic and Computation 20 (5) (2010) 1041–1068.
[46] C. Jones, Probabilistic Non-determinism, Ph.D. Thesis, University of Edinburgh, 1989.
[47] B. Jonsson, K. Larsen, Specification and refinement of probabilistic processes, in: Proc. LICS, IEEE, 1991, pp. 266–277.
[48] B. Jonsson, K. Larsen, W. Yi, Probabilistic extensions of process algebras, in: Handbook of Process Algebras, Elsevier, North Holland, 2001, pp. 685–710.
[49] J. Kemeny, J. Snell, Finite Markov Chains, Springer-Verlag, New York, 1976.
[50] B. Klin, Structural operational semantics for weighted transition systems, in: Semantics and Algebraic Specification, Essays Dedicated to Peter D. Mosses on the Occasion of His 60th Birthday, in: LNCS, vol. 5700, 2009, pp. 121–139.
[51] B. Klin, Bialgebras for structural operational semantics: An introduction, Theoretical Computer Science 412 (38) (2011) 5043–5069.
[52] B. Klin, V. Sassone, Structural operational semantics for stochastic process calculi, in: Proc. FoSSaCS, in: LNCS, vol. 4962, 2008, pp. 428–442.
[53] C. Kupke, D. Pattinson, Coalgebraic semantics of modal logics: An overview, Theoretical Computer Science 412 (38) (2011) 5070–5094.
[54] A. Kurz, Logics for coalgebras and applications to computer science, Ph.D. Thesis, Ludwig-Maximilians-Universität München, 2000.
[55] K. Larsen, A. Skou, Bisimulation through probabilistic testing, Information and Computation 94 (1991) 1–28.
[56] F. Lawvere, The category of probabilistic mappings, 1962, Preprint.
[57] L. Moss, Coalgebraic logic, Annals of Pure and Applied Logic 96 (1999) 277–317.
[58] L. Moss, I. Viglizzo, Final coalgebras for functors on measurable spaces, Information and Computation 240 (2006) 610–636.
[59] P. Panangaden, Labelled Markov Processes, Imperial College Press, 2009.
[60] D. Pattinson, L. Schröder, Modular algorithms for heterogeneous modal logics via multi-sorted coalgebra, Mathematical Structures in Computer Science 21 (2010) 235–266.
[61] A. Paz, Introduction to Probabilistic Automata (Computer Science and Applied Mathematics), Academic Press, Inc., Orlando, FL, USA, 1971.
[62] A. Pnueli, L. Zuck, Probabilistic verification, Information and Computation 103 (1993) 1–29.
[63] M. Rabin, Probabilistic automata, Information and Control 6 (1963) 230–245.
[64] J. Rutten, Universal coalgebra: a theory of systems, Theoretical Computer Science 249 (2000) 3–80.
[65] L. Schröder, Expressivity of coalgebraic modal logic: the limits and beyond, Theoretical Computer Science 390 (2–3) (2008) 230–247.
[66] R. Segala, Modeling and verification of randomized distributed real-time systems, Ph.D. Thesis, MIT, 1995.
[67] R. Segala, N. Lynch, Probabilistic simulations for probabilistic processes, in: Proc. CONCUR, in: LNCS, vol. 836, 1994, pp. 481–496.
[68] A. Silva, F. Bonchi, M.M. Bonsangue, J.J.M.M. Rutten, Quantitative Kleene coalgebras, Information and Computation 209 (5) (2011) 822–849.
[69] A. Sokolova, Coalgebraic analysis of probabilistic systems, Ph.D. Thesis, Eindhoven University of Technology, 2005.
[70] A. Sokolova, E.d. Vink, Probabilistic automata: system types, parallel composition and comparison, in: Validation of Stochastic Systems: A Guide to Current Research, in: LNCS, vol. 2925, 2004, pp. 1–43.
[71] A. Sokolova, E.d. Vink, H. Woracek, Coalgebraic weak bisimulation for action-type systems, Scientific Annals of Computer Science 19 (2009) 93–144.
[72] M. Stoelinga, An introduction to probabilistic automata, Bulletin of the EATCS 78 (2002) 176–198.
[73] D. Turi, G. Plotkin, Towards a mathematical operational semantics, in: Proc. LICS, IEEE, 1997, pp. 280–291.
[74] D. Turi, J. Rutten, On the foundations of final coalgebra semantics, Mathematical Structures in Computer Science 8 (5) (1998) 481–540.
[75] D. Varacca, Probability, nondeterminism and concurrency: two denotational models for probabilistic computation, Ph.D. Thesis, Univ. Aarhus, 2003. BRICS Dissertation Series, DS-03-14.
[76] D. Varacca, G. Winskel, Distributing probabililty over nondeterminism, Mathematical Structures in Computer Science 16 (1) (2006) 87–113.
[77] M. Vardi, Automatic verification of probabilistic concurrent finite state programs, in: Proc. FOCS, IEEE, Portland, Oregon, 1985, pp. 327–338.
[78] I. Viglizzo, Coalgebras on measurable spaces, Ph.D. Thesis, Indiana University, 2005.
[79] I. Viglizzo, Final sequences and final coalgebras for measurable spaces, in: CALCO, in: LNCS, vol. 3629, 2005, pp. 395–407.
[80] E.d. Vink, J. Rutten, Bisimulation for probabilistic transition systems: a coalgebraic approach, Theoretical Computer Science 221 (1999) 271–293.
[81] U. Wolter, On corelations, cokernels, and coequations, in: Proc. CMCS'00, Electronic Notes in Theoretical Computer Science 33 (2000).

Theoretical Computer Science 412 (2011) 3072–3089

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Information hiding in probabilistic concurrent systems

Miguel E. Andrés [a], Catuscia Palamidessi [b,*], Peter van Rossum [a], Ana Sokolova [c]

[a] *Institute for Computing and Information Sciences, Radboud University, The Netherlands*

[b] *INRIA and LIX, École Polytechnique Palaiseau, France*

[c] *Departament of Computer Sciences, University of Salzburg, Austria*

A B S T R A C T

Information hiding is a general concept which refers to the goal of preventing an adversary from inferring secret information from the observables. Anonymity and Information Flow are examples of this notion. We study the problem of information hiding in systems characterized by the coexistence of randomization and concurrency. It is well known that the presence of nondeterminism, due to the possible interleavings and interactions of the parallel components, can cause unintended information leaks. The most established approach to solve this problem is to fix the strategy of the scheduler beforehand. In this work, we propose a milder restriction on the schedulers, and we define the notion of strong (probabilistic) information hiding under various notions of observables. Furthermore, we propose a method, based on the notion of automorphism, to verify that a system satisfies the property of strong information hiding, namely strong anonymity or non-interference, depending on the context. Through the paper, we use the canonical example of the Dining Cryptographers to illustrate our ideas and techniques.

1. Introduction

The problem of information hiding consists in trying to prevent the adversary from inferring confidential information from the observables. Instances of this issue are Anonymity and Information Flow. In both fields there is a growing interest in the quantitative aspects of the problem, see for instance [25,3,40,14,15,29,30,4,17,11,12,38]. This is justified by the fact that often we have some a priori knowledge about the likelihood of the various secrets (which we can usually express in terms of a probability distribution), and by the fact that protocols often use randomized actions to obfuscate the link between secret and observable, like in the case of the anonymity protocols of DC Nets [13], Crowds [34], Onion Routing [39], and Freenet [16].

In a concurrent setting, like in the case of multi-agent systems, there is also another source of uncertainty, which derives from the fact that the various entities may interleave and interact in ways that are usually unpredictable, either because they depend on factors that are too complex to analyze, or because (in the case of specifications) they are implementation-dependent.

The formal analysis of systems which exhibit probabilistic and nondeterministic behavior usually involves the use of so-called *schedulers*, which are functions that, for each path, select only one possible (probabilistic) transition, thus delivering a purely probabilistic execution tree, where each event has a precise probability.

In the area of security, there is the problem that secret choices, like all choices, give rise to different paths. On the other hand, the decision of the scheduler may influence the observable behavior of the system. Therefore the security properties

* Corresponding author. Tel.: +33 (0)1 69333798.
E-mail address: catuscia@lix.polytechnique.fr (C. Palamidessi).

0304-3975/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.02.045

are usually violated if we admit as schedulers all possible functions of the paths: certain schedulers induce a dependence of the observables on the secrets, and protocols which would not leak secret information when running in "real" systems (where the scheduling devices cannot "see" the internal secrets of the components and therefore cannot depend on them), do leak secret information under this more permissive notion of scheduler. This is a well-known problem for which various solutions have already been proposed [7,8,10,9]. We will come back to these in the "Related work" section.

1.1. Contribution

We now list the main contribution of this work:

- We define a class of partial-information schedulers (which we call *admissible*), schedulers in this class are a restricted version of standard (full-information) schedulers. The restriction is rather flexible and has strong structural properties, thus facilitating the reasoning about security properties. In short, our systems consist of parallel components with certain restrictions on the secret choices and nondeterministic choices. The scheduler selects the next component (or components, in case of synchronization) for the subsequent step independently of the secret choices. We then formalize the notion of quantitative information flow, or degree of anonymity, using this restricted notion of scheduler.
- We propose alternative definitions to the property of strong anonymity defined in [3]. Our proposal differs from the original definition in two aspects: (1) the system should be strongly anonymous for all admissible schedulers instead of all schedulers (which is a very strong condition, never satisfied in practice), (2) we consider several variants of adversaries, namely (in increasing level of power): external adversaries, internal adversaries, and adversaries in collusion with the scheduler (in a Dolev–Yao fashion). Additionally, we use admissible schedulers to extend the notions of multiplicative and additive leakage (proposed in [38,5] respectively) to the case of a concurrent system.
- We propose a sufficient technique to prove probabilistic strong anonymity, and probabilistic noninterference, based on automorphisms. The idea is the following: In the purely nondeterministic setting, the strong anonymity of a system is often proved (or defined) as follows: take two users A and B and a trace in which user A is 'the culprit'. Now find a trace that looks the same to the adversary, but in which user B is 'the culprit' [25,22,31,26]. This new trace is often most easily obtained by *switching the behavior of A and B*. Non-interference can be proved in the same way (where A and B are high information and the trace is the low information).

  In this work, we make this technique explicit for anonymity in systems where probability and nondeterminism coexist, and we need to cope with the restrictions on the schedulers. We formalize the notion of *switching behaviors* by using automorphism (it is possible to switch the behavior of A and B if there exist an automorphism between them) and then show that the existence of an automorphism implies strong anonymity.
- We illustrate the problem with full-information schedulers in security, our solution providing admissible schedulers, and the application of our proving technique by means of the well known Dining Cryptographers anonymity protocol.

1.2. Related work

The problem of the full-information scheduler has already been extensively investigated in the literature. The works [7] and [8] consider probabilistic automata and introduce a restriction on the scheduler to the purpose of making them suitable to applications in security. Their approach is based on dividing the actions of each component of the system in equivalence classes (*tasks*). The order of execution of different tasks is decided in advance by a so-called *task scheduler*. The remaining nondeterminism within a task is resolved by a second scheduler, which models the standard *adversarial scheduler* of the cryptographic community. This second entity has limited knowledge about the other components: it sees only the information that they communicate during execution. Their notion of task scheduler is similar to our notion of admissible scheduler, but more restricted since the strategy of the task scheduler is decided entirely before the execution of the system.

Another work along these lines is [19], which uses partitions on the state-space to obtain partial-information schedulers. However that work considers a synchronous parallel composition, so the setting is rather different from ours.

The work in [10,9] is similar to ours in spirit, but in a sense *dual* from a technical point of view. Instead of defining a restriction on the class of schedulers, they provide a way to specify that a choice is transparent to the scheduler. They achieve this by introducing labels in process terms, used to represent both the states of the execution tree and the next action or step to be scheduled. They make two states indistinguishable to schedulers, and hence the choice between them private, by associating to them the same label. Furthermore, their "equivalence classes" (schedulable actions with the same label) can change dynamically, because the same action can be associated to different labels during the execution.

In [1] we have extended the framework presented in this work (by allowing internal nondeterminism and adding a second type of scheduler to resolve it) with the aim of investigating angelic vs demonic nondeterminism in equivalence-based properties.

The fact that full-information schedulers are unrealistic has also been observed in fields other than security. With the aim to cope with general properties (not only those concerning security), the first attempts used restricted schedulers in order to obtain rules for compositional reasoning [19]. The justification for those restricted schedulers is the same as for ours, namely, that not all information is available to all entities in the system. Later on, it was shown that model checking is unfeasible in its general form for the kind of restricted schedulers presented in [19]. See [24] and, more recently, [23].

To the best of our knowledge, this is the first work using automorphisms as a sound proof technique (in our case to prove strong anonymity and non-interference). The closest line of work we are aware of is in the field of model checking. There, isomorphisms can be used to identify symmetries in the system, and such symmetries can then be exploited to alleviate the state space explosion (see for instance [28]).

A notion similar to that of authomorphism, namely bisimulation, has been used in several works to define and verify anonymity and privacy properties (see for instance [20]). However, the underlying models are different: we annotate the transitions with a label identifying the active component(s), which means that an automorphism can relate different states only if they result from different *probabilistic* choices. The notion of bisimulation used in the security literature, on the contrary, usually relates states resulting from different *nondeterministic* choices. In [1] we argue that the notion of bisimulation can be too permissive in certain cases.

A preliminary version of this work, without proofs, appeared in [2].

1.3. Plan of the paper

Looking ahead, after reviewing some preliminaries (Section 2) we formalize the notions of systems and components (Section 3). In Section 4 we present admissible schedulers. We then formalize the notions of internal and external strong anonymity in a probabilistic and nondeterministic setting for admissible schedulers (Section 5). Finally, we turn our attention to the verification problem, in Section 6 we present a strong-anonymity proving technique based on automorphisms. We conclude and outline some future work in Section 7.

2. Preliminaries

In this section we gather preliminary notions and results related to probabilistic automata [37,36], information theory [18], and information leakage [38,5].

2.1. Probabilistic automata

A function $\mu\colon Q \to [0,1]$ is a *discrete probability distribution* on a set Q if $\sum_{q\in Q}\mu(q) = 1$. The set of all discrete probability distributions on Q is denoted by $\mathcal{D}(Q)$.

A *probabilistic automaton* is a quadruple $M = (Q, \Sigma, \hat{q}, \theta)$ where Q is a countable set of *states*, Σ a finite set of *actions*, $\hat{q}$ the *initial* state, and θ a *transition function* $\theta : Q \to \mathcal{P}(\mathcal{D}(\Sigma \times Q))$. Here $\mathcal{P}(X)$ is the set of all subsets of X.

If $\theta(q) = \emptyset$, then q is a *terminal* state. We write $q \to \mu$ for $\mu \in \theta(q)$, $q \in Q$. Moreover, we write $q \xrightarrow{a} r$ for $q, r \in Q$ whenever $q \to \mu$ and $\mu(a, r) > 0$. A *fully probabilistic automaton* is a probabilistic automaton satisfying $|\theta(q)| \leq 1$ for all states. In case $\theta(q) \neq \emptyset$ in a fully probabilistic automaton, we will overload notation and use $\theta(q)$ to denote the distribution outgoing from q. A *path* in a probabilistic automaton is a sequence $\sigma = q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} \cdots$ where $q_i \in Q, a_i \in \Sigma$ and $q_i \xrightarrow{a_{i+1}} q_{i+1}$. A path can be *finite* in which case it ends with a state. A path is *complete* if it is either infinite or finite ending in a terminal state. Given a path σ, $\mathit{first}(\sigma)$ denotes its first state, and if σ is finite then $\mathit{last}(\sigma)$ denotes its last state. A *cycle* is a path σ such that $\mathit{last}(\sigma) = \mathit{first}(\sigma)$. Let $\mathrm{Paths}_q(M)$ denote the set of all paths, $\mathrm{Paths}^\star_q(M)$ the set of all finite paths, and $\mathrm{CPaths}_q(M)$ the set of all complete paths of an automaton M, starting from the state q. We will omit q if $q = \hat{q}$. Paths are ordered by the prefix relation, which we denote by $\leq$. The *trace* of a path is the sequence of actions in $\Sigma^* \cup \Sigma^\infty$ obtained by removing the states, hence for the above path σ we have $\mathit{trace}(\sigma) = a_1 a_2 \ldots$. If $\Sigma' \subseteq \Sigma$, then $\mathit{trace}_{\Sigma'}(\sigma)$ is the projection of $\mathit{trace}(\sigma)$ on the elements of Σ'.

Let $M = (Q, \Sigma, \hat{q}, \theta)$ be a (fully) probabilistic automaton, $q \in Q$ a state, and let $\sigma \in \mathrm{Paths}^\star_q(M)$ be a finite path starting in q. The *cone* generated by σ is the set of complete paths $\langle\sigma\rangle = \{\sigma' \in \mathrm{CPaths}_q(M) \mid \sigma \leq \sigma'\}$. Given a fully probabilistic automaton $M = (Q, \Sigma, \hat{q}, \theta)$ and a state q, we can calculate the *probability value*, denoted by $\mathbf{P}_q(\sigma)$, of any finite path σ starting in q as follows: $\mathbf{P}_q(q) = 1$ and $\mathbf{P}_q(\sigma \xrightarrow{a} q') = \mathbf{P}_q(\sigma)\,\mu(a, q')$, where $\mathit{last}(\sigma) \to \mu$.

Let $\Omega_q \stackrel{\text{def}}{=} \mathrm{CPaths}_q(M)$ be the sample space, and let $\mathcal{F}_q$ be the smallest σ-algebra generated by the cones. Then $\mathbf{P}_q$ induces a unique *probability measure* on $\mathcal{F}_q$ (which we will also denote by $\mathbf{P}_q$) such that $\mathbf{P}_q(\langle\sigma\rangle) = \mathbf{P}_q(\sigma)$ for every finite path σ starting in q. For $q = \hat{q}$ we write $\mathbf{P}$ instead of $\mathbf{P}_{\hat{q}}$.

A (full-information) scheduler for a probabilistic automaton M is a function $\zeta : \mathrm{Paths}^\star(M) \to (\mathcal{D}(\Sigma \times Q) \cup \{\bot\})$ such that for all finite paths σ, if $\theta(\mathit{last}(\sigma)) \neq \emptyset$ then $\zeta(\sigma) \in \theta(\mathit{last}(\sigma))$, and $\zeta(\sigma) = \bot$ otherwise. Hence, a scheduler ζ selects one of the available transitions in each state, and determines therefore a fully probabilistic automaton, obtained by pruning from M the alternatives that are not chosen by ζ. Note that a scheduler is history dependent since it can take different decisions for the same state s according to the past evolution of the system.

2.2. Noisy channels

This section briefly recalls the notion of noisy channels from Information Theory [18].

A *noisy channel* is a tuple $\mathcal{C} \stackrel{\text{def}}{=} (\mathcal{X}, \mathcal{Y}, P(\cdot|\cdot))$ where $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ is a finite set of *input values*, modeling the *secrets* of the channel, and $\mathcal{Y} = \{y_1, y_2, \ldots, y_m\}$ is a finite set of *output values*, the *observables* of the channel. For $x_i \in \mathcal{X}$ and $y_j \in \mathcal{Y}$, $P(y_j|x_i)$ is the conditional probability of obtaining the output y_j given that the input is x_i. These conditional probabilities constitute the so-called *channel matrix*, where $P(y_j|x_i)$ is the element at the intersection of the i-th row and the j-th column. For any input distribution P_X on $\mathcal{X}$, P_X and the channel matrix determine a joint probability $P_\wedge$ on $\mathcal{X} \times \mathcal{Y}$, and the corresponding marginal probability P_Y on $\mathcal{Y}$ (and hence a random variable Y). P_X is also called *a priori distribution* and it is often denoted by π. The probability of the input given the output is called *a posteriori distribution*.

2.3. Information leakage

We recall here the definitions of *multiplicative leakage* proposed in [38], and of *additive leakage* proposed in [5][1]. We assume given a noisy channel $\mathcal{C} = (\mathcal{X}, \mathcal{Y}, P(\cdot|\cdot))$ and a random variable X on $\mathcal{X}$. The *a priori vulnerability* of the secrets in $\mathcal{X}$ is the probability of guessing the right secret, defined as $V(X) \stackrel{\text{def}}{=} \max_{x\in\mathcal{X}} P_X(x)$. The rationale behind this definition is that the adversary's best bet is on the secret with highest probability. The *a posteriori vulnerability* of the secrets in $\mathcal{X}$ is the probability of guessing the right secret, after the output has been observed, averaged over the probabilities of the observables. The formal definition is $V(X \mid Y) \stackrel{\text{def}}{=} \sum_{y\in\mathcal{Y}} P_Y(y) \max_{x\in\mathcal{X}} P(x \mid y)$. Again, this definition is based on the principle that the adversary will choose the secret with the highest a posteriori probability.

Note that, using the definition of conditional probability, we can write the a posteriori vulnerability in terms of the joint probability, or in terms of the channel matrix and the a priori distribution:

$$V(X \mid Y) = \sum_{y\in\mathcal{Y}} \max_{x\in\mathcal{X}} P_\wedge(x, y). = \sum_{y\in\mathcal{Y}} \max_{x\in\mathcal{X}} (P(y \mid x) P_X(x))$$

The *multiplicative* leakage is $\mathcal{L}_\times(\mathcal{C}, P_X) \stackrel{\text{def}}{=} \frac{V(X|Y)}{V(X)}$ whereas the *additive* leakage is $\mathcal{L}_+(\mathcal{C}, P_X) \stackrel{\text{def}}{=} V(X|Y) - V(X)$.

2.4. Dining cryptographers

This problem, described by Chaum in [13], involves a situation in which three cryptographers are dining together. At the end of the dinner, each of them is secretly informed by a central agency (master) whether he should pay the bill, or not. So, either the master will pay, or one of the cryptographers will be asked to pay. The cryptographers (or some external observer) would like to find out whether the payer is one of them or the master. However, if the payer is one of them, they also wish to maintain anonymity over the identity of the payer.

A possible solution to this problem, described in [13], is that each cryptographer tosses a coin, which is visible to himself and his neighbor to the left. Each cryptographer observes the two coins that he can see and announces *agree* or *disagree*. If a cryptographer is not paying, he will announce *agree* if the two sides are the same and *disagree* if they are not. The paying cryptographer will say the opposite. It can be proved that if the number of disagrees is even, then the master is paying; otherwise, one of the cryptographers is paying. Furthermore, in case one of the cryptographers is paying, neither an external observer nor the other two cryptographers can identify, from their individual information, who exactly is paying (provided that the coins are fair). The Dining Cryptographers (DC) will be a running example through the paper.

3. Systems

In this section we describe the kind of systems we are dealing with. We start by introducing a variant of probabilistic automata, that we call *tagged probabilistic automata* (TPA). These systems are parallel compositions of purely probabilistic processes, that we call *components*. They are equipped with a unique identifier, that we call *tag*, or *label*, of the component. Note that, because of the restriction that the components are fully deterministic, nondeterminism is generated only from the interleaving of the parallel components. Furthermore, because of the uniqueness of the tags, each transition from a node is associated to a different tag/pair of two tags (one in case only one component makes a step, and two in case of a synchronization step among two components).

[1] The notion proposed by Smith in [38] was given in a (equivalent) logarithmic form, and called simply *leakage*. For uniformity sake we use here the terminology and formulation of [5].

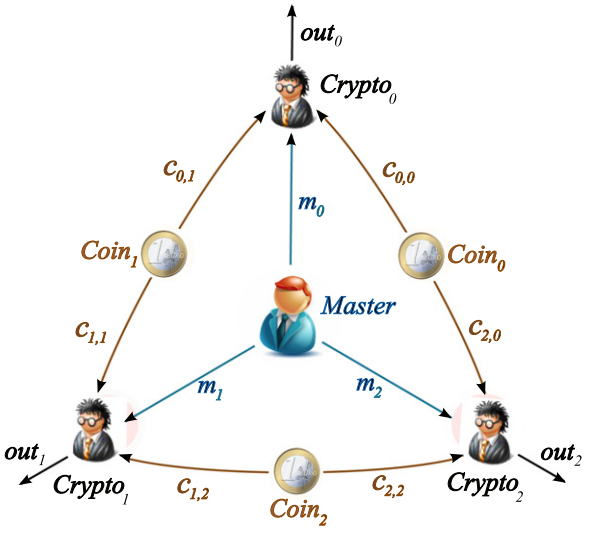


Fig. 1. Chaum's system for the Dining Cryptographers [13]..

3.1. Tagged Probabilistic automata

We now formalize the notion of TPA.

Definition 1. A *tagged probabilistic automaton* (TPA) is a tuple $(Q, L, \Sigma, \hat{q}, \theta)$, where

- Q is a set of *states*,
- L is a set of *tags*, or *labels*,
- Σ is a set of *actions*,
- $\hat{q} \in Q$ is the *initial state*,
- $\theta : Q \to \mathcal{P}(L \times D(\Sigma \times Q))$ is a *transition function*.

with the additional requirement that for every $q \in Q$ and every $\ell \in L$ there is at most one $\mu \in D(\Sigma \times Q)$ such that $(\ell, \mu) \in \theta(q)$.

A path for a TPA is a sequence $\sigma = q_0 \xrightarrow{l_1,a_1} q_1 \xrightarrow{l_2,a_2} q_2 \cdots$. In this way, the process with identifier l_i induces the system to move from q_{i-1} to q_i performing the action a_i, and it does so with probability $\mu_{l_i}(a_i, q_i)$, where μ_{l_i} is *the* distribution associated to the choice made by the component l_i. Finite paths and complete paths are defined in a similar manner.

In a TPA, the scheduler's choice is determined by the choice of the tag. We will use $enab(q)$ to denote the tags of the components that are enabled to make a transition. Namely,

$$enab(q) \stackrel{\text{def}}{=} \{\ell \in L \mid \exists \mu \in D(\Sigma \times Q) \ : \ (\ell, \mu) \in \theta(q)\} \tag{1}$$

We assume that the scheduler is forced to select a component among those which are enabled, i.e., that the execution does not stop unless all components are blocked (suspended or terminated). This is in line with the spirit of process algebra, and also with the tradition of Markov Decision Processes, but contrasts with that of the Probabilistic Automata of Lynch and Segala [37]. However, the results in this paper do not depend on this assumption; we could as well allow the (more general) notion of schedulers which may decide to terminate the execution even though there are transitions which are possible from the last state. The reason we did not do it is because it would be confusing in the setting of process algebra, and also it would complicate the notation and the proofs. On the other hand, for the purposes of this paper the generality added by the second notion of scheduler can be captured by adding a component that performs only one observable action representing termination.

Definition 2. A *scheduler* for a TPA $M = (Q, L, \Sigma, \hat{q}, \theta)$ is a function $\zeta : \text{Paths}^\star(M) \to (L \cup \{\bot\})$ such that for all finite paths σ, $\zeta(\sigma) \in enab(last(\sigma))$ if $enab(last(\sigma)) \neq \emptyset$ and $\zeta(\sigma) = \bot$ otherwise.

3.2. Components

To specify the components we use a sort of probabilistic version of CCS [32,33]. We assume a set of *secret actions* Σ_S with elements $s, s_1, s_2, \ldots$, and a disjoint set of *observable actions* Σ_O with elements $a, a_1, a_2, \ldots$. Furthermore we have a disjoint set of *communication actions* of the form $c(x)$ (receive x on channel c, where x is a formal parameter), or $\bar{c}\langle v \rangle$ (send v on channel c, where v is a value on some domain V). Sometimes we need only to synchronize without transmitting any value, in which case we will use simply c and $\bar{c}$. We denote the set of channel names by C.

A component q is specified by the following grammar:

Components

q	::=	0	termination
	|	$a.q$	observable prefix
	|	$\sum_i p_i : q_i$	blind choice
	|	$\sum_i p_i : s_i.q_i$	secret choice
	|	*if* $x = v$ *then* q_1 *else* q_2	conditional
	|	A	process call

Observables

a	::=	$c \mid \bar{c}$	simple synchronization
	|	$c(x) \mid \bar{c}\langle v \rangle$	synchronization and communication

The p_i, in the blind and secret choices, represents the probability of the i-th branch and must satisfy $0 \leq p_i \leq 1$ and $\sum_i p_i = 1$. When no confusion arises, we use simply $+$ for a binary choice. The process call A is a simple process identifier. For each of them, we assume a corresponding unique process declaration of the form $A \stackrel{\text{def}}{=} q$. The idea is that, whenever A is executed, it triggers the execution of q. Note that q can contain A or another process identifier, which means that our language allows (mutual) recursion.

Note that each component contains only probabilistic and sequential constructs. In particular, there is no internal parallelism nor nondeterminism (apart from the input nondeterminism, which disappears in the definition of a system). Hence each component corresponds to a purely probabilistic automaton, as described by the operational semantics below. The main reason to dismiss the use of internal parallelism and nondeterminism is verification: as mentioned in the introduction we will present a proving technique for the different definitions of anonymity proposed in this work. This result would not be possible without such restriction on the components (see Example 5).

For an extension of this framework allowing the use of internal parallelism and nondeterminism we refer to [1]. There, the authors combine *global nondeterminism* (arising from the interleaving of the components) and *local nondeterminism* (allowed as a primitive and also arising from the internal parallelism of the components). The authors use such (extended) framework for a different purpose than ours, namely to define a notion of equivalence suitable for security analysis. No verification mechanisms are provided in [1].

Components' semantics: The operational semantics consists of probabilistic transitions of the form $q \rightarrow \mu$ where $q \in Q$ is a process, and $\mu \in \mathcal{D}(\Sigma \times Q)$ is a distribution on actions and processes. They are specified by the following rules:

$$\text{PRF1}\ \frac{v \in V}{c(x).q \rightarrow \delta(c(v), q[v/x])} \qquad \text{PRF2}\ \frac{}{a.q \rightarrow \delta(a, q)}\ \text{if } a \neq c(x)$$

$$\text{INT}\ \frac{}{\sum_i p_i : q_i \rightarrow \sum\!\!\!\!\!\!\bigcirc_i\ p_i \cdot \delta(\tau, q_i)} \qquad \text{SECR}\ \frac{}{\sum_i p_i : s_i.q_i \rightarrow \sum\!\!\!\!\!\!\bigcirc_i\ p_i \cdot \delta(s_i, q_i)}$$

$$\text{CND1}\ \frac{}{\begin{array}{l}\textit{if } v = v \textit{ then } q_1 \\ \qquad \textit{else } q_2 \rightarrow \delta(\tau, q_1)\end{array}} \qquad \text{CND2}\ \frac{v \neq v'}{\begin{array}{l}\textit{if } v = v' \textit{ then } q_1 \\ \qquad \textit{else } q_2 \rightarrow \delta(\tau, q_2)\end{array}}$$

$$\text{CALL}\ \frac{q \rightarrow \mu}{A \rightarrow \mu}\ \text{if } A \stackrel{\text{def}}{=} q$$

$\sum\!\!\!\!\!\!\bigcirc_i\ p_i \cdot \mu_i$ is the distribution μ such that $\mu(x) = \sum_i p_i \mu_i(x)$. We use $\delta(x)$ to represent the delta of Dirac, which assigns probability 1 to x. The silent action, τ, is a special action different from all the observable and the secret actions. $q[v/x]$ stands for the process q in which any occurrence of x has been replaced by v. To shorten the notation, in the examples throughout the paper, we omit writing explicit termination, i.e., we omit the symbol 0 at the end of a term.

3.3. Systems

A system consists of n processes (components) in parallel, restricted at the top-level on the set of channel names C:

$(C)\ q_1 \parallel q_2 \parallel \cdots \parallel q_n$.

The restriction on C enforces synchronization (and possibly communication) on the channel names belonging to C, in accordance with the CCS spirit. Since C is the set of all channels, all of them are forced to synchronize. This is to eliminate, at the level of systems, the nondeterminism generated by the rule for the receive prefix, PRF1.

Systems' semantics: The semantics of a system gives rise to a TPA, where the states are terms representing systems during their evolution. A transition now is of the form $q \xrightarrow{\ell} \mu$ where $\mu \in (\mathcal{D}(\Sigma \times Q))$ and $\ell \in L$ is either the identifier of the component which makes the move, or a two-element set of identifiers representing the two partners of a synchronization. The following two rules provide the operational semantics rules in the case of interleaving and synchronisation/communication, respectively.

Interleaving.

$$\frac{q_i \to \sum_j p_j \cdot \delta(a_j, q_{ij})}{(C)\ q_1 \parallel \cdots \parallel q_i \parallel \cdots \parallel q_n \xrightarrow{i} \sum_j p_j \cdot \delta(a_j, (C)\ q_1 \parallel \cdots \parallel q_{ij} \parallel \cdots \parallel q_n)} \quad \text{if } a_j \notin C$$

where i indicates the tag of the component making the step.

Synchronization/Communication.

$$\frac{q_i \to \delta(\bar{c}\langle v\rangle, q_i') \qquad q_j \to \delta(c(v), q_j')}{(C)\ q_1 \parallel \cdots \parallel q_i \parallel \cdots \parallel q_n \xrightarrow{\{i,j\}} \delta(\tau, (C)\ q_1 \parallel \cdots \parallel q_i' \parallel \cdots \parallel q_j' \parallel \cdots \parallel q_n)}$$

here $\{i, j\}$ is the tag indicating that the components making the step are i and j. For simplicity we write $\xrightarrow{i,j}$ instead of $\xrightarrow{\{i,j\}}$. The rule for synchronization without communication is similar, the only difference is that we do not have $\langle v\rangle$ and (v) in the actions. Note that c can only be an observable action (neither a secret nor τ), by the assumption that channel names can only be observable actions.

We note that both interleaving and synchronization rules generate nondeterminism. The only other source of nondeterminism is PRF1, the rule for a receive prefix $c(x)$. However the latter is not real nondeterminism: it is introduced in the semantics of the components but it disappears in the semantics of the systems, given that the channel c is restricted at the top-level. In fact the restriction enforces communication, and when communication takes place, only the branch corresponding to the actual value v transmitted by the corresponding send action is maintained, all the others disappear.

Proposition 1. *The operational semantics of a system is a TPA with the following characteristics:*

(a) *Every step* $q \xrightarrow{\ell} \mu$ *is either*

a blind choice: $\mu = \sum_i p_i \cdot \delta(\tau, q_i)$, *or*
a secret choice: $\mu = \sum_i p_i \cdot \delta(s_i, q_i)$, *or*
a delta of Dirac: $\mu = \delta(\alpha, q')$ *with* $\alpha \in \Sigma_O$ *or* $\alpha = \tau$.

(b) *If* $q \xrightarrow{\ell} \mu$ *and* $q \xrightarrow{\ell} \mu'$ *then* $\mu = \mu'$.

Proof.

(a) The rules for the components and the rule for synchronization/communication can only produce blind choices, secret choices, or deltas of Dirac. Furthermore, because of the restriction on all channels, the transitions at the system level cannot contain communication actions. Finally, observe that the interleaving rule maintains these properties.
(b) At the component level, the only source of nondeterminism is PRF1, the rule for a receive prefix $c(x)$. At the system level, this action is forced to synchronize with a corresponding send action, and, in a component, there can be only one such action available at a time. Hence the tag determines the value to be sent, which in turn determines the selection of exactly one branch in the receiving process. The only other sources of nondeterminism are the interleaving and the synchronization/communication rules, and they induce a different tag for each alternative. □

Example 1. We now present the components for the Dining Cryptographers using the introduced syntax. They correspond to Fig. 1 and to the automata depicted in Fig. 3. As announced before, we omit the symbol 0 for explicit termination at the end of each term. The secret actions s_i represent the choice of the payer. The operators $\oplus$, $\ominus$ represent the sum modulo 2 and the difference modulo 2, respectively. The test $i == n$ returns 1 (true) if $i = n$, and 0 otherwise. The set of restricted channel names is $C = \{c_{0,0}, c_{0,1}, c_{1,1}, c_{1,2}, c_{2,0}, c_{2,2}, m_0, m_1, m_2\}$.

The operation $pay \oplus coin_1 \oplus coin_2$ in Fig. 2 is syntactic sugar, it can be defined using the *if-then-else* operator. Note that, in this way, if a cryptographer is not paying (pay $= 0$), then he announces 0 if the two coins are the same (agree) and 1 if they are not (disagree).

$$\begin{aligned}
\text{Master} &\stackrel{\text{def}}{=} p : \overline{m}_0\langle 0\rangle \,.\, \overline{m}_1\langle 0\rangle \,.\, \overline{m}_2\langle 0\rangle + (1-p) : \textstyle\sum_{i=0}^{2} p_i : s_i \,. \\
&\qquad \overline{m}_0\langle i == 0\rangle \,.\, \overline{m}_1\langle i == 1\rangle \,.\, \overline{m}_2\langle i == 2\rangle \\
\text{Crypt}_i &\stackrel{\text{def}}{=} m_i(pay) \,.\, c_{i,i}(coin_1) \,.\, c_{i,i\oplus 1}(coin_2) \,.\, \overline{out}_i\langle pay \oplus coin_1 \oplus coin_2\rangle \\
\text{Coin}_i &\stackrel{\text{def}}{=} 0.5 : \bar{c}_{i,i}\langle 0\rangle \,.\, \bar{c}_{i\ominus 1,i}\langle 0\rangle \;+\; 0.5 : \bar{c}_{i,i}\langle 1\rangle \,.\, \bar{c}_{i\ominus 1,i}\langle 1\rangle \\
\text{System} &\stackrel{\text{def}}{=} (C)\ \text{Master} \parallel \textstyle\prod_{i=0}^{2} \text{Crypt}_i \parallel \prod_{i=0}^{2} \text{Coin}_i
\end{aligned}$$

Fig. 2. Dining cryptographers CCS.

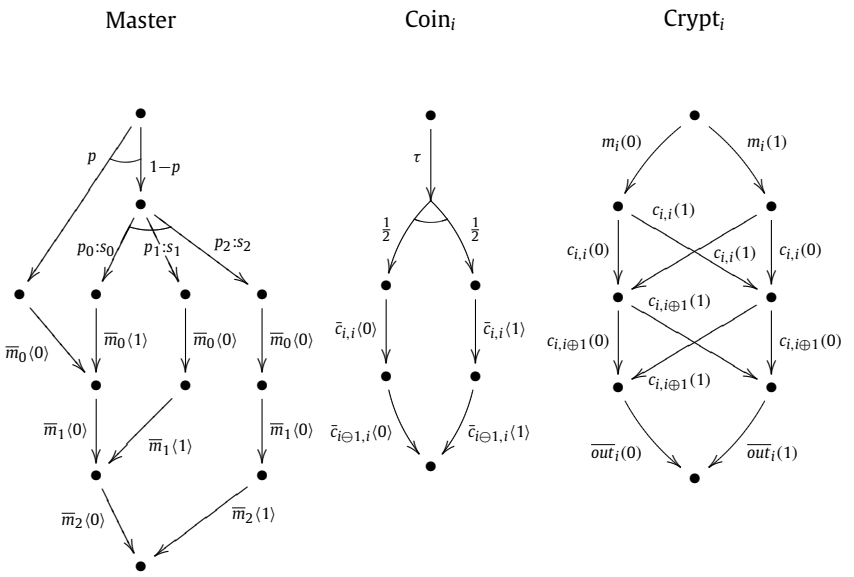


Fig. 3. Dining cryptographers automata.

4. Admissible schedulers

We now introduce the class of admissible schedulers.

Standard (full-information) schedulers have access to all the information about the system and its components, and in particular the secret choices. Hence, such schedulers can leak secrets by making their decisions depend on the secret choice of the system. This is the case with the Dining Cryptographers protocol of Section 2.4: among all possible schedulers for the protocol, there are several that leak the identity of the payer. In fact the scheduler has the freedom to decide the order of the announcements of the cryptographers (interleaving), so a scheduler could choose to let the payer announce lastly. In this way, the attacker learns the identity of the payer simply by looking at the interleaving of the announcements.

4.1. The screens intuition

Let us first describe admissible schedulers informally. As mentioned in the introduction, admissible schedulers can base their decisions only on partial information about the evolution of the system, in particular admissible schedulers cannot base their decisions on information concerned with the internal behavior of components (such as secret choices).

We follow the subsequent intuition: admissible schedulers are entities that have access to a screen with buttons, where each button represents one (current) available option. At each point of the execution the scheduler decides the next step among the available options (by pressing the corresponding button). Then the output (if any) of the selected component becomes available to the scheduler and the screen is refreshed with the new available options (the ones corresponding to the system after making the selected step). We impose that the scheduler can base its decisions only on such information, namely: the screens and outputs he has seen up to that point of the execution (and, of course, the decisions he has made).

Example 2. Consider $S \stackrel{\text{def}}{=} (\{c_1, c_2\})\, r \parallel q \parallel t$, where

$$r \stackrel{\text{def}}{=} 0.5 : s_1.\overline{c}_1.\overline{c}_2 + 0.5 : s_2.\overline{c}_1.\overline{c}_2,$$
$$q \stackrel{\text{def}}{=} c_1.(0.5 : a_1 + 0.5 : b_1), \qquad t \stackrel{\text{def}}{=} c_2.(0.5 : a_2 + 0.5 : b_2).$$

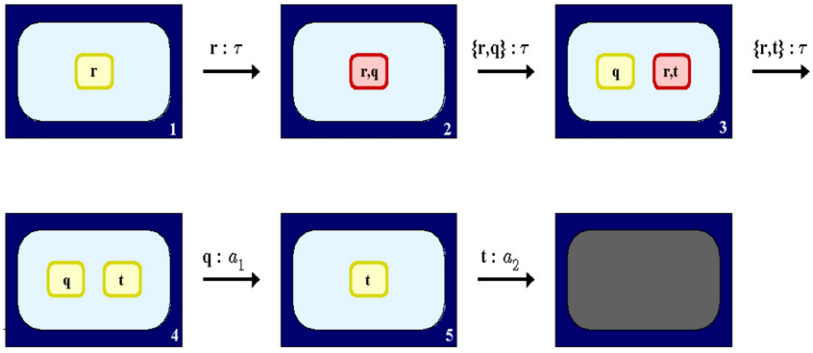


Fig. 4. Screens intuition.

Fig. 4 shows the sequence of screens corresponding to a particular sequence of choices taken by the scheduler.[2] Interleaving and communication options are represented by yellow and red buttons, respectively. An arrow between two screens represents the transition from one to the other (produced by the scheduler pressing a button), additionally, the decision taken by the scheduler and corresponding outputs are depicted above each arrow.

Note that this system has exactly the same problem as the DC protocol: a full-information scheduler could reveal the secret by basing the interleaving order (q first or t first) on the secret choice of the component r. However, the same does not hold any more for admissible schedulers (the scheduler cannot deduce the secret choice by just looking at the screens and outputs). This is also the case for the DC protocol, i.e., admissible schedulers cannot leak the secret of the protocol.

4.2. The formalization

Before formally defining admissible schedulers we need to formalize the ingredients of the screens intuition. The buttons on the screen (available options) are the enabled options given by the function *enab* (see (1) in Section 3), the decision made by the scheduler is the tag of the selected enabled option, observable actions are obtained by sifting the secret actions to the schedulers by means of the following function:

$$sift(\alpha) \stackrel{\text{def}}{=} \begin{cases} \alpha & \text{if } \alpha \in \Sigma_O \cup \{\tau\}, \\ \tau & \text{if } \alpha \in \Sigma_S. \end{cases}$$

The partial information of a certain evolution of the system is given by the map t defined as follows.

Definition 3. Let $\hat{q} \stackrel{\ell_1,\alpha_1}{\longrightarrow} \cdots \stackrel{\ell_n,\alpha_n}{\longrightarrow} q_{n+1}$ be a finite path of the system, then we define t as:

$$t\left(\hat{q} \stackrel{\ell_1,\alpha_1}{\longrightarrow} \cdots \stackrel{\ell_n,\alpha_n}{\longrightarrow} q_{n+1}\right) \stackrel{\text{def}}{=} (enab(\hat{q}), \ell_1, sift(\alpha_1)) \cdots (enab(q_n), \ell_n, sift(\alpha_n)) \cdot enab(q_{n+1}).$$

Finally, we have all the ingredients needed to define admissible schedulers.

Definition 4 (*Admissible Schedulers*)**.** A scheduler ζ is admissible if for all $\sigma, \sigma' \in \text{Paths}^\star$

$$t(\sigma) = t(\sigma') \quad \text{implies} \quad \zeta(\sigma) = \zeta(\sigma').$$

In this way, admissible schedulers are forced to take the same decisions on paths that they cannot tell apart. Note that this is a restriction on the original definition of (full-information) schedulers where t is the identity map over finite paths (and consequently the scheduler is free to choose differently).

In the kind of systems we consider (the TPAs) the only source of nondeterminism are the interleaving and interactions of the parallel components. Consequently, in a TPA the notion of scheduler is quite simple: its role, indeed, is to select, at each step, the component or pair of components which will perform the next transition. In addition, the TPA model allows us to express in a simple way the notion of admissibility: in fact the transitions available in the last state of σ are determined by the set of components enabled in the last state of σ, and $t(\sigma)$ gives (among other information) such set. Therefore $t(\sigma) = t(\sigma')$ implies that the last states of σ and σ' have the same possible transitions, hence it is possible to require that $\zeta(\sigma) = \zeta(\sigma')$ without being too restrictive or too permissive. In more general systems, where the sources of nondeterminism can be arbitrary, it is difficult to impose that the scheduler "does not depend on the secret choices", because different secret choices in general may give rise to states with different sets of transitions, and it is unclear whether such difference should be ruled out as "inadmissible", or should be considered as part of what a "real" scheduler can detect.

[2] The transitions from screens 4 and 5 represent 2 steps each (for simplicity we omit the τ-steps generated by blind choices).

5. Information-hiding properties in presence of nondeterminism

In this section we revise the standard definition of information flow and anonymity in our framework of controlled nondeterminism.

We first consider the notion of adversary. We consider three possible notions of adversaries, increasingly more powerful.

5.1. Adversaries

External adversaries: Clearly, an adversary should be able, by definition, to see at least the observable actions. For an adversary external to the system S, it is natural to assume that these are also the only actions that he is supposed to see. We also assume that an external adversary has no way of knowing whether an internal synchronization has taken place. To this purpose, we need to abstract from the τ actions. Therefore, we define the observation domain, for an external adversary, as the set of the (finite) sequences of observable actions, namely:

$$\mathcal{O}_{\mathrm{e}} \stackrel{\mathrm{def}}{=} \Sigma_O^*.$$

Correspondingly, we need a function $t_{\mathrm{e}} : \mathrm{Paths}^\star(S) \to \mathcal{O}_{\mathrm{e}}$ that extracts the observables from the executions:

$$t_{\mathrm{e}}\left(q_0 \stackrel{\ell_1,\alpha_1}{\longrightarrow} \cdots \stackrel{\ell_n,\alpha_n}{\longrightarrow} q_{n+1}\right) \stackrel{\mathrm{def}}{=} \mathit{sieve}(\alpha_1)\cdots\mathit{sieve}(\alpha_n)$$

where

$$\mathit{sieve}(\alpha) \stackrel{\mathrm{def}}{=} \begin{cases} \alpha & \text{if } \alpha \in \Sigma_O, \\ \epsilon & \text{if } \alpha \in \Sigma_S \cup \{\tau\}. \end{cases}$$

Note that the difference between *sift* and *sieve* is that the latter transforms the τ actions in empty string, thus making completely invisible to the external adversary whether an internal synchronization has taken place or not.

Internal adversaries: An internal adversary may be able to see, besides the observables, also the interleaving and synchronizations of the various components, i.e. which component(s) are active, at each step of the execution. Hence it is natural to define the observation domain, for an internal adversary, as the sequence of pairs of observable action and tag (i.e. the identifier(s) of the active component(s)), namely:

$$\mathcal{O}_{\mathrm{i}} \stackrel{\mathrm{def}}{=} (L \times (\Sigma_O \cup \{\tau\}))^*.$$

Analogously to the case of the external adversaries, we need a function $t_{\mathrm{i}} : \mathrm{Paths}^\star(S) \to \mathcal{O}_{\mathrm{i}}$ that extracts the observables from the executions:

$$t_{\mathrm{i}}\left(q_0 \stackrel{\ell_1,\alpha_1}{\longrightarrow} \cdots \stackrel{\ell_n,\alpha_n}{\longrightarrow} q_{n+1}\right) \stackrel{\mathrm{def}}{=} (\ell_1, \mathit{sieve}(\alpha_1))\cdots(\ell_n, \mathit{sieve}(\alpha_n)).$$

Note that, in contrast to the case of t_{e}, in the definition of t_{i} we could have used, equivalently, *sift* instead of *sieve*.

Adversaries in collusion with the scheduler: Finally, we consider the case in which the adversary is in collusion with the scheduler, or possibly the adversary *is* the scheduler. To illustrate the difference between these kinds of adversaries and internal adversaries, consider the scheduler of an operating system. In such a scenario an internal adversary is able to see which process has been scheduled to run next (process in the "running state") whereas an adversary in collusion with the scheduler can see as much as the scheduler, thus being able to see (in addition) which processes are in the "ready state" and which processes are in the "waiting / blocked" state. We will show later that such additional information does not help the adversary to leak information (see Proposition 4). The observation domain of adversaries in collusion with the scheduler coincides with the one of the scheduler:

$$\mathcal{O}_{\mathrm{s}} \stackrel{\mathrm{def}}{=} (\mathcal{P}(L) \times L \times (\Sigma_O \cup \{\tau\}))^*.$$

The corresponding function

$$t_{\mathrm{s}} : \mathrm{Paths}^\star(S) \to \mathcal{O}_{\mathrm{s}}$$

is defined as the one of the scheduler, i.e. $t_{\mathrm{s}} = t$.

5.2. Information leakage

In the fields of information flow and anonymity there is a converging consensus for formalizing the notion of leakage as the difference or the ratio between the a priori uncertainty that the adversary has about the secret, and the a posteriori uncertainty, that is, the residual uncertainty of the adversary once it has seen the outcome of the computation. The uncertainty can be measured in different ways. One popular approach is the information-theoretic one, according to which the system is seen as a noisy channel between the secret inputs and the observable output, and uncertainty corresponds to the entropy of the system (see preliminaries—Section 2). In this approach, the leakage is represented by the so-called mutual information, which expresses the correlation between the input and the output.

In most of the approaches in the information flow literature the notion of entropy uses Shannon entropy. However Cachin, in his Ph.D. thesis [6], had already argued that the right notion of entropy should depend on the notion of adversary, and on the way we measure its success. More recently, Köpf and Basin have considered again this intuition, and have developed an information-theoretic schema to define leakage in terms of mutual information for a large class of adversaries [27].

In his recent paper, Smith has considered again the question of the most suitable notion of entropy, focusing on the particular case of one-try attacks [38]. He has argued that Shannon entropy is not suitable to represent the security threats in the case in which the adversary is interested in figuring out the secret in one try, and he has proposed to use Rényi's min entropy instead, or equivalently, the average probability of guessing the secret in one try. This leads to interpret the uncertainty in terms of the notion of *vulnerability* defined in the preliminaries (Section 2). The corresponding notion of leakage, in the pure probabilistic case, has been investigated in [38] (multiplicative case) and in [5] (additive case).

Here we adopt the vulnerability-based approach to define the notion of leakage in our probabilistic and nondeterministic context. The Shannon-entropy-based approach could be extended to our context as well, because in both cases we only need to specify how to determine the conditional probabilities which constitute the channel matrix, and the marginal probabilities that constitute the input and the output distribution.

We will denote by S the random variable associated to the set of secrets $\mathcal{S} = \Sigma_S$, and by O_x the random variables associated to the set of observables $\mathcal{O}_x$, where $x \in \{e, i, s\}$. So, $\mathcal{O}_x$ represents the observation domains for the various kinds of adversaries defined above.

As mentioned before, our results require some structural properties for the system: we assume that there is a single component in the system containing a secret choice and this component contains a single secret choice. This hypothesis is general enough to allow expressing protocols like the Dining Cryptographers, Crowds, voting protocols, etc., where the secret is chosen only once.

Assumption 1. A system contains exactly one component with a syntactic occurrence of a secret choice, and such a choice does not occur in the scope of a recursive call.

Note that the assumption implies that the choice appears exactly once in the operational semantics of the component. It would be possible to relax the assumption and allow more than one secret choice in a component, as long as there are no observable actions between the secret choices. But for the sake of simplicity in this paper we impose the more restrictive requirement. As a consequence, we have that the operational semantics of systems satisfies the following property:

Proposition 2. *If $q \xrightarrow{\ell} \mu$ and $q' \xrightarrow{\ell'} \mu'$ are both secret choices, then $\ell = \ell'$ and there exist p_i's, q_i's and q_i''s such that:*

$$\mu = \sum_i p_i \cdot \delta(s_i, q_i) \quad \text{and} \quad \mu' = \sum_i p_i \cdot \delta(s_i, q_i')$$

i.e., μ and μ' differ only for the continuation states.

Proof. Because of Assumption 1, there is only one component that can generate a secret choice, and it generates only one such choice. Due to the different possible interleavings, this choice can appear as an outgoing transition in more than one state of the TPA, but the probabilities are always the same, because the interleaving rule does not change them. □

Given a system, each scheduler ζ determines a fully probabilistic automaton, and, as a consequence, the probabilities

$$\mathbf{P}_\zeta\,(s, o) \stackrel{\text{def}}{=} \mathbf{P}_\zeta\left(\bigcup \{\langle\sigma\rangle \mid \sigma \in \text{Paths}^\star(S), t_x(\sigma) = o, secr(\sigma) = s\}\right)$$

for each secret $s \in \mathcal{S}$ and observable $o \in \mathcal{O}_x$, where $x \in \{e, i, s\}$. Here *secr* is the map from paths to their secret action. From these we can derive, in standard ways, the marginal probabilities $\mathbf{P}_\zeta\,(s)$, $\mathbf{P}_\zeta\,(o)$, and the conditional probabilities $\mathbf{P}_\zeta\,(o \mid s)$.

Every scheduler leads to a (generally different) noisy channel, whose matrix is determined by the conditional probabilities as follows:

Definition 5. Let $x \in \{e, i, s\}$. Given a system and a scheduler ζ, the corresponding channel matrix $\mathcal{C}_\zeta^x$ has rows indexed by $s \in \mathcal{S}$ and columns indexed by $o \in \mathcal{O}_x$. The value in (s, o) is given by

$$\mathbf{P}_\zeta\,(o \mid s) \stackrel{\text{def}}{=} \frac{\mathbf{P}_\zeta\,(s, o)}{\mathbf{P}_\zeta\,(s)}$$

Given a scheduler ζ, the multiplicative leakage can be defined as $\mathcal{L}_\times(\mathcal{C}^{\mathrm{x}}_\zeta, P_\zeta)$, while the additive leakage can be defined as $\mathcal{L}_+(\mathcal{C}^{\mathrm{x}}_\zeta, P_\zeta)$ where P_ζ is the a priori distribution on the set of secrets (see preliminaries, Section 2). However, we want a notion of leakage independent from the scheduler, and therefore it is natural to consider the worst case over all possible admissible schedulers.

Definition 6 (*x-Leakage*)**.** Let $x \in \{\mathrm{e}, \mathrm{i}, \mathrm{s}\}$. Given a system, the multiplicative leakage is defined as

$$\mathcal{ML}^{\mathrm{x}}_\times \stackrel{\mathrm{def}}{=} \max_{\zeta \in Adm} \mathcal{L}_\times(\mathcal{C}^{\mathrm{x}}_\zeta, P_\zeta),$$

while the additive leakage is defined as

$$\mathcal{ML}^{\mathrm{x}}_+ \stackrel{\mathrm{def}}{=} \max_{\zeta \in Adm} \mathcal{L}_+(\mathcal{C}^{\mathrm{x}}_\zeta, P_\zeta),$$

where *Adm* is the class of admissible schedulers defined in the previous section.

We have that the classes of observables e, i, and s determine an increasing degree of leakage:

Proposition 3. *Given a system, for the multiplicative leakage we have*

1. *For every scheduler* ζ, $\mathcal{L}_\times(\mathcal{C}^{\mathrm{e}}_\zeta, P_\zeta) \leq \mathcal{L}_\times(\mathcal{C}^{\mathrm{i}}_\zeta, P_\zeta) \leq \mathcal{L}_\times(\mathcal{C}^{\mathrm{s}}_\zeta, P_\zeta)$
2. $\mathcal{ML}^{\mathrm{e}}_\times \leq \mathcal{ML}^{\mathrm{i}}_\times \leq \mathcal{ML}^{\mathrm{s}}_\times$

Similarly for the additive leakage.

Proof.

1. The property follows immediately from the fact that the domain $\mathcal{O}_{\mathrm{e}}$ is an abstraction of $\mathcal{O}_{\mathrm{i}}$, and $\mathcal{O}_{\mathrm{i}}$ is an abstraction of $\mathcal{O}_{\mathrm{s}}$.
2. Immediate from previous point and from the definition of $\mathcal{ML}^{\mathrm{x}}_\times$ and $\mathcal{ML}^{\mathrm{x}}_+$. □

5.3. Strong anonymity (revised)

We consider now the situation in which the leakage is the minimum for all possible admissible schedules. In the purely probabilistic case, we know that the minimum possible multiplicative leakage is 1, and the minimum possible additive one is 0. We also know that this is the case for all possible input distributions if and only if the capacity of the channel matrix is 0, which corresponds to the case in which the rows of the matrix are all the same. This corresponds to the notion of strong probabilistic anonymity defined in [3]. In the framework of information flow, it would correspond to probabilistic non-interference. Still in [3], the authors considered also the extension of this notion in presence of nondeterminism, and required the condition to hold under all possible schedulers. This is too strong in practice, as we have argued in the introduction: in most cases we can build a scheduler that leaks the secret by changing the interleaving order. We therefore tune this notion by requiring the condition to hold only under the admissible schedulers.

Definition 7 (*x-Strongly Anonymous*)**.** Let $x \in \{\mathrm{e}, \mathrm{i}, \mathrm{s}\}$. We say that a system is *x-strongly-anonymous* if for all admissible schedulers ζ we have

$$\mathbf{P}_\zeta(o \mid s_1) = \mathbf{P}_\zeta(o \mid s_2)$$

for all $s_1, s_2 \in \Sigma_S$, and $o \in \mathcal{O}_{\mathrm{x}}$.

The following corollary is an immediate consequence of Proposition 3.

Corollary 8.

1. *If a system is s-strongly-anonymous, then it is also i-strongly-anonymous.*
2. *If a system is i-strongly-anonymous, then it is also e-strongly-anonymous.*

The converse of point (2), in the previous corollary, does not hold, as shown by the following example:

Example 3. Consider the system $S \stackrel{\mathrm{def}}{=} (\{c_1, c_2\})\, P \,||\, Q \,||\, T$ where

$$P \stackrel{\mathrm{def}}{=} (0.5 : s_1 \,.\, \overline{c}_1) + (0.5 : s_2 \,.\, \overline{c}_2) \qquad Q \stackrel{\mathrm{def}}{=} c_1 \,.\, o \qquad T \stackrel{\mathrm{def}}{=} c_2 \,.\, o$$

It is easy to check that S is *e*-strongly anonymous but not *i*-strongly anonymous, showing that (as expected) internal adversaries can "distinguish more" than external adversaries.

On the contrary, for point (1) of Corollary 8, also the other direction holds:

Proposition 4. *A system is s-strongly-anonymous if and only if it is i-strongly-anonymous.*

Proof. Corollary 8 ensures the only-if part. For the if part, we proceed by contradiction. Assume that the system is i-strongly-anonymous but that $\mathbf{P}_\zeta\,(o \mid s_1) \neq \mathbf{P}_\zeta\,(o \mid s_2)$ for some admissible scheduler ζ and observable $o \in \mathcal{O}_s$. Let $o = (enab(\hat{q}), \ell_1, sift(\alpha_1)) \ldots (enab(q_n), \ell_n, sift(\alpha_n))$ and let o' be the projection of o on $\mathcal{O}_i$, i.e. $o' = (\ell_1, sift(\alpha_1)) \cdots (\ell_n, sift(\alpha_n))$. Since the system is i-strongly-anonymous, $\mathbf{P}_\zeta\left(o' \mid s_1\right) = \mathbf{P}_\zeta\left(o' \mid s_2\right)$, which means that the difference in probability with respect to o must be due to at least one of the sets of available processes. Let us consider the first set L in o which exhibits a difference in the probabilities, and let o'' be the prefix of o up to the tuple containing L. Since the probabilities are determined by the distributions on the probabilistic choices which occur in the individual components, the probability of each $\ell \in L$ to be available (given the trace o'') is independent of the other labels in L. At least one such ℓ must therefore have a different probability, given the trace o'', depending on whether the secret choice was s_1 or s_2. And, because of the assumption on L, we can replace the conditioning on trace o'' with the conditioning on the projection o''' of o'' on $\mathcal{O}_i$. Consider now an admissible scheduler ζ' that acts like ζ up to o'', and then selects ℓ if and only if it is available. Since the probability that ℓ is not available depends on the choice of s_1 or s_2, we have $\mathbf{P}_\zeta\left(o''' \mid s_1\right) \neq \mathbf{P}_\zeta\left(o''' \mid s_2\right)$, which contradicts the hypothesis that the system is i-strongly-anonymous. □

Intuitively, this result means that an *s*-adversary can leak information if and only if an *i*-adversary can leak information or, in other words, *s*-adversaries are as powerful as *i*-adversaries (even when the former can observe more information).

6. On the verification of strong anonymity: a proving technique based on automorphisms

As mentioned in the introduction, several problems involving restricted schedulers have been shown undecidable (including computing maximum/minimum probabilities for the case of standard model checking [24], [23]). These results are discouraging in the aim to find algorithms for verifying strong anonymity/non-interference using our notion of admissible schedulers (and most definitions based on restricted schedulers). Despite the fact that the problem seems to be undecidable in general, in this section we present a sufficient (but not necessary) anonymity proving technique: we show that the existence of automorphisms between each pair of secrets implies strong anonymity. We conclude this section illustrating the applicability of our proving technique by means of the DC protocol, i.e., we prove that the protocol does not leak information by constructing automorphisms between pairs of cryptographers. It is worth mentioning that our proving technique is general enough to be used for the analysis of leakage information of a broad family of protocols, namely any protocol that can be modeled in our framework.

6.1. The proving technique

In practice proving anonymity often happens in the following way. Given a trace in which user *A* is the 'culprit', we construct an observationally equivalent trace in which user *B* is the 'culprit' [25,22,31,26]. This new trace is typically obtained by 'switching' the behavior of users *A* and *B*. We formalize this idea by using the notion of automorphism, cf. e.g. [35].

Definition 9 (*Automorphism*)**.** Given a TPA $(Q, L, \Sigma, \hat{q}, \theta)$ we say that a bijection $f : Q \to Q$ is an *automorphism* if it satisfies $f(\hat{q}) = \hat{q}$ and

$$q \xrightarrow{\ell} \sum_i p_i \cdot \delta(\alpha_i, q_i) \iff f(q) \xrightarrow{\ell} \sum_i p_i \cdot \delta(\alpha_i, f(q_i)).$$

In order to prove anonymity it is sufficient to prove that the behaviors of any two 'culprits' can be exchanged without the adversary noticing. We will express this by means of the existence of automorphisms that exchange a given pair of secret s_i and s_j.

Before presenting the main theorem of this section we need to introduce one last definition. Let $S = (C)\ q_1 || \cdots || q_n$ be a system and M its corresponding TPA. We define M_τ as the automaton obtained after "hiding" all the secret actions of M. The idea is to replace every occurrence of a secret s in M by the silent action τ. Note that this can be formalized by replacing the secret choice by a blind choice in the corresponding component q_i of the system S.

We now formalize the relation between automorphisms and strong anonymity. We will first show that the existence of automorphisms exchanging pairs of secrets implies *s*-strong anonymity (Theorem 1). After, we will show that the converse does not hold, i.e., *s*-strongly-anonymous systems are not necessarily automorphic (Example 4).

Theorem 1. *Let S be a system satisfying Assumption 1 and M its tagged probabilistic automaton. If for every pair of secrets $s_i, s_j \in \Sigma_S$ there exists an automorphism f of M_τ such that for any state q we have*

$$q \xrightarrow{\ell, s_i}_M q' \Longrightarrow f(q) \xrightarrow{\ell, s_j}_M f(q'), \tag{2}$$

then S is s-strongly-anonymous.

Proof. Assume that for every pair of secrets s_i, s_j we have an automorphism f satisfying the hypothesis of the theorem. We have to show that, for every admissible scheduler ζ we have:

$$\forall o \in \mathcal{O}_s : \ \mathbf{P}_\zeta\ (o \mid s_1) = \mathbf{P}_\zeta\ (o \mid s_2)\ .$$

We start by observing that for s_i, by Proposition 2, there exists a unique p_i such that, for all transitions $q \xrightarrow{l} \mu$, if μ is a (probabilistic) secret choice, then $\mu(s_i, -) = p_i$. Similarly for s_j, there exists a unique p_j such that $\mu(s_j, -) = p_j$ for all secret choices μ.

Let us now recall the definition of $\mathbf{P}_\zeta\ (o \mid s)$:

$$\mathbf{P}_\zeta\ (o \mid s) \stackrel{\text{def}}{=} \frac{\mathbf{P}_\zeta\ (o \wedge s)}{\mathbf{P}_\zeta\ (s)}$$

where

$$\mathbf{P}_\zeta\ (o \wedge s) \stackrel{\text{def}}{=} \mathbf{P}_\zeta\ (\{\pi \in \text{CPaths} \mid t_s(\pi) = o \wedge secr(\pi) = s\})$$

with $secr(\pi)$ being the (either empty or singleton) sequence of secret actions of π, and

$$\mathbf{P}_\zeta\ (s) \stackrel{\text{def}}{=} \mathbf{P}_\zeta\ (\{\pi \in \text{CPaths} \mid secr(\pi) = s\})\ .$$

Note that, since a secret appears at most once on a complete path, we have:

$$\begin{aligned}
\mathbf{P}_\zeta\ (s_i) &= \mathbf{P}_\zeta\left(\{\pi \xrightarrow{\ell, s_i} \sigma \in \text{CPaths} \mid \pi, \sigma\}\right) \\
&= \sum_{\pi \xrightarrow{\ell, s_i} q_i \in \text{Paths}^\star} \mathbf{P}_\zeta\left(\pi \xrightarrow{\ell, s_i} q_i\right) \\
&= \sum_{\substack{last(\pi) \xrightarrow{\ell} \mu \\ \mu \text{ secret choice}}} \mathbf{P}_\zeta\ (\pi) \cdot p_i
\end{aligned}$$

and analogously

$$\begin{aligned}
\mathbf{P}_\zeta\ (s_j) &= \mathbf{P}_\zeta\left(\{\pi \xrightarrow{\ell, s_j} \sigma \in \text{CPaths} \mid \pi, \sigma\}\right) \\
&= \sum_{\pi \xrightarrow{\ell, s_j} q_j \in \text{Paths}^\star} \mathbf{P}_\zeta\left(\pi \xrightarrow{\ell, s_j} q_j\right) \\
&= \sum_{\substack{last(\pi) \xrightarrow{\ell} \mu \\ \mu \text{ secret choice}}} \mathbf{P}_\zeta\ (\pi) \cdot p_j
\end{aligned}$$

Let us now consider $\mathbf{P}_\zeta\ (o \mid s_i)$ and $\mathbf{P}_\zeta\ (o \mid s_j)$. We have:

$$\begin{aligned}
\mathbf{P}_\zeta\ (o \wedge s_i) &= \mathbf{P}_\zeta\left(\left\{\pi \xrightarrow{\ell, s_i} \sigma \in \text{CPaths} \mid t_s(\pi \xrightarrow{\ell, s_i} \sigma) = o\right\}\right) \\
&= \sum_{\substack{\pi \\ last(\pi) \xrightarrow{\ell} \mu \\ \mu \text{ secret choice}}} \mathbf{P}_\zeta\ (\pi) \cdot p_i \cdot \sum_{\substack{\sigma \\ \pi \xrightarrow{\ell, s_i} \sigma \in \text{Paths}^\star \\ t_s(\pi \xrightarrow{\ell, s_i} \sigma) = o \wedge last(t_e(\sigma)) \neq \tau}} \mathbf{P}_\zeta\ (\sigma)
\end{aligned}$$

again using that a secret appears at most once on a complete path. Moreover, note that we have overloaded the notation $\mathbf{P}_\zeta$ by using it for different measures when writing $\mathbf{P}_\zeta\ (\sigma)$, since σ need not start in the initial state $\hat{q}$.
Analogously we have:

$$\begin{aligned}
\mathbf{P}_\zeta\ (o \wedge s_j) &= \mathbf{P}_\zeta\left(\left\{\pi \xrightarrow{\ell, s_j} \sigma \in \text{CPaths} \mid t_s(\pi \xrightarrow{\ell, s_j} \sigma) = o\right\}\right) \\
&= \sum_{\substack{\pi \\ last(\pi) \xrightarrow{\ell} \mu \\ \mu \text{ secret choice}}} \mathbf{P}_\zeta\ (\pi) \cdot p_j \cdot \sum_{\substack{\sigma \\ \pi \xrightarrow{\ell, s_j} \sigma \in \text{Paths}^\star \\ t_s(\pi \xrightarrow{\ell, s_j} \sigma) = o \wedge last(t_e(\sigma)) \neq \tau}} \mathbf{P}_\zeta\ (\sigma)
\end{aligned}$$

Therefore, we derive

$$\mathbf{P}_\zeta\,(o \mid s_i) = \frac{\displaystyle\sum_{\substack{\pi \\ last(\pi)\xrightarrow{\ell}\mu \\ \mu \text{ secret choice}}} \sum_{\substack{\sigma \\ \pi \xrightarrow{\ell,s_i} \sigma \in \mathrm{Paths}^\star \\ t_s(\pi \xrightarrow{\ell,s_i} \sigma)=o \wedge last(t_e(\sigma))\neq\tau}} \mathbf{P}_\zeta\,(\pi)\cdot\mathbf{P}_\zeta\,(\sigma)}{\displaystyle\sum_{\substack{last(\pi)\xrightarrow{\ell}\mu \\ \mu \text{ secret choice}}} \mathbf{P}_\zeta\,(\pi)} \tag{3}$$

$$\mathbf{P}_\zeta\,\big(o \mid s_j\big) = \frac{\displaystyle\sum_{\substack{\pi \\ last(\pi)\xrightarrow{\ell}\mu \\ \mu \text{ secret choice}}} \sum_{\substack{\sigma \\ \pi \xrightarrow{\ell,s_j} \sigma \in \mathrm{Paths}^\star \\ t_s(\pi \xrightarrow{\ell,s_j} \sigma)=o \wedge last(t_e(\sigma))\neq\tau}} \mathbf{P}_\zeta\,(\pi)\cdot\mathbf{P}_\zeta\,(\sigma)}{\displaystyle\sum_{\substack{last(\pi)\xrightarrow{\ell}\mu \\ \mu \text{ secret choice}}} \mathbf{P}_\zeta\,(\pi)} \tag{4}$$

Observe that the denominators of both formulae (3) and (4) are the same. Also note that, since f is an automorphism, for every path π, $f(\pi)$ obtained by replacing each state in π with its image under f is also a path. Moreover, since f satisfies (2), for every path $\pi \xrightarrow{\ell,s_i} \sigma$ we have that $f(\pi) \xrightarrow{\ell,s_j} f(\sigma)$ is also a path. Furthermore f induces a bijection between the sets

$$\{(\pi,\sigma) \mid last(\pi) \xrightarrow{\ell'} \mu \text{ s.t. } \mu \text{ secret choice}, \pi \xrightarrow{\ell,s_i} \sigma \in \mathrm{Paths}^\star t_s(\pi \xrightarrow{\ell,s_i} \sigma) = o, last(t_e(\sigma)) \neq \tau\}$$

and

$$\{(\pi,\sigma) \mid last(\pi) \xrightarrow{\ell'} \mu \text{ s.t. } \mu \text{ secret choice}, \pi \xrightarrow{\ell,s_j} \sigma \in \mathrm{Paths}^\star t_s(\pi \xrightarrow{\ell,s_j} \sigma) = o, last(t_e(\sigma)) \neq \tau\}$$

given by $(\pi,\sigma) \leftrightarrow (f(\pi), f(\sigma))$.

Finally, since ζ is admissible, $t_s(\pi) = t_s(f(\pi))$, and f is an automorphism, it is easy to prove by induction that $\mathbf{P}_\zeta\,(\pi) = \mathbf{P}_\zeta\,(f(\pi))$. Similarly, $\mathbf{P}_\zeta\,(\sigma) = \mathbf{P}_\zeta\,(f(\sigma))$. Hence the numerators of (3) and (4) coincide which concludes the proof. □

Note that, since s-strong anonymity implies i-strong anonymity and e-strong anonymity, the existence of such an automorphism implies all the notions of strong anonymity presented in this work. We now proceed to show that the converse does not hold, i.e., strongly-anonymous systems are not necessarily automorphic.

Example 4. Consider the following (single component) system

$$\begin{array}{c} 0.5 : s_1.(0.5 : (p : a + (1-p) : b) + 0.5 : ((1-p) : a + p : b)) \\ + \\ 0.5 : s_2.(0.5 : (q : a + (1-q) : b) + 0.5 : ((1-q) : a + q : b)) \end{array}$$

It is easy to see that such a system is s-strongly-anonymous, however if $p \neq q$ and $p \neq 1-q$ there does not exist an automorphism for the pair of secrets (s_1, s_2).

The following example demonstrates that our proving technique does not carry over to systems whose components admit internal parallelism.

Example 5. Consider $S \stackrel{\text{def}}{=} (\{c_1, c_2\})\, r \,\|\, q \,\|\, t$, where

$$\begin{array}{l} r \stackrel{\text{def}}{=} 0.5 : s_1.\bar{c}_1 + 0.5 : s_2.\bar{c}_2, \\ q \stackrel{\text{def}}{=} c_1.(a \mid b), \qquad t \stackrel{\text{def}}{=} c_2.(a \mid b). \end{array}$$

where $q_1|q_2$ represents the parallel composition of q_1 and q_2. It is easy to show that there exists an automorphism for s_1 and s_2. However, admissible schedulers are able to leak such secrets. This is due to the fact that component r synchronizes with q and t on different channels, thus a scheduler of S is not restricted to select the same transitions on the branches associated to s_1 and s_2 (remember that schedulers can observe synchronization).

For the same reason, our proving technique does not extend to systems whose components contain nondeterminism. An example proving this point can be obtained from the previous one by replacing the parallel composition $q_1|q_2$ with the nondeterministic composition $q_1 + q_2$.

We now show that the definition of x-strong-anonymity is independent of the particular distribution over secrets, i.e., if a system is x-strongly-anonymous for a particular distribution over secrets, then it is x-strongly-anonymous for all distributions over secrets. This result is useful because it allows us to prove systems to be strongly anonymous even when their distribution over secrets is not known.

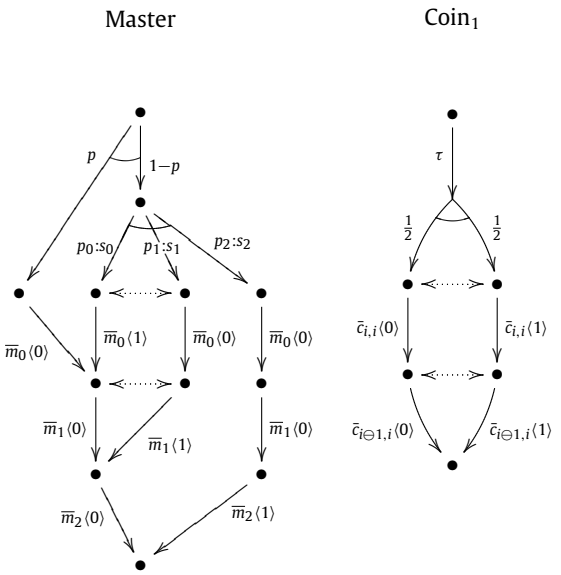


Fig. 5. Automorphism between Crypt$_0$ and Crypt$_1$.

Theorem 2. *Consider a system* $S = (C)\ q_1 \parallel \cdots \parallel q_i \parallel \cdots \parallel q_n$. *Let* q_i *be the component which contains the secret choice, and assume that it is of the form* $\sum_j p_j : s_j \,.\, q_j$. *Consider now the system* $S' = (C)\ q_1 \parallel \cdots \parallel q'_i \parallel \cdots \parallel q_n$, *where* q'_i *is identical to* q_i *except for the secret choice, which is replaced by* $\sum_j p'_j : s_j \,.\, q_j$. *Then we have that:*

1. *For every* s_i, s_j *there is an automorphism on* S *satisfying the assumption of Theorem 1 if and only if the same holds for* S'.
2. S *is x-strongly-anonymous if and only if* S' *is x-strongly-anonymous.*

Note: (1) does not imply (2), because in principle neither S not S' may have the automorphism, and still one of the two could be strongly anonymous.

Proof. We note that the PAs generated by S and S' coincide except for the probability distribution on the secret choices. Since the definition of automorphism and the assumption of Theorem 1 do not depend on these probability distributions, (1) is immediate. As for (2), we observe that x-strong anonymity only depends on the conditional probabilities $\mathbf{P}_\zeta\ (o \mid s)$. By looking at the proof of Theorem 1, we can see that in the computation of $\mathbf{P}_\zeta\ (o \mid s)$ the probabilities on the secret choices (i.e. the p_j's) are eliminated. Namely $\mathbf{P}_\zeta\ (o \mid s)$ does not depend on the p_j's, which means that the value of the p_j's has no influence on whether the system is x-strong anonymous or not. □

6.2. An Application: dining cryptographers

Now we show how to apply the proving technique presented in this section to the Dining Cryptographers protocol. Concretely, we show that there exists an automorphism f exchanging the behavior of the Crypt$_0$ and Crypt$_1$; by symmetry, the same holds for the other two combinations.

Consider the automorphisms of Master and Coin$_1$ indicated in Fig. 5. The states that are not explicitly mapped (by a dotted arrow) are mapped to themselves.

Also consider the identity automorphism on Crypt$_i$ (for $i = 0, 1, 2$) and on Coin$_i$ (for $i = 0, 2$). It is easy to check that the product of these seven automorphisms is an automorphism for Crypt$_0$ and Crypt$_1$.

7. Conclusion and future work

We have defined a class of partial-information schedulers which can only base their decisions on the information they have available. In particular they cannot base their decisions on the internal behavior of the components. We have used admissible schedulers to resolve nondeterminism in a realistic way, and to tune the definition of strong anonymity proposed in [3].

Furthermore, we have presented a technique to prove the various definitions of strong anonymity proposed in the paper. This is particularly interesting considering that many problems related to restricted schedulers have been shown to be undecidable. In particular we have shown how to use the technique to prove that the DC protocol is strongly anonymous when considering admissible schedulers, in contrast to the situation when considering full-information schedulers.

We think that the results of this paper would hold also if we considered probabilistic schedulers (instead than deterministic ones). In a sense, the issue of probabilistic versus deterministic schedulers is orthogonal to the issue of partial versus full information schedulers, and to the problem of "taming" nondeterminism to avoid leakage.

For future work, we plan to investigate the decidability problem for the various definitions of strong anonymity we have proposed. Another interesting direction for future work is to extend well-known isomorphism-checking algorithms and tools (see [21] for a survey) to our setting in order to verify automatically strong anonymity (in case an automorphism exists — recall that this is not a necessary condition).

Moreover, we plan to investigate the type of protocols that can be captured by our framework. We believe that we can also cope with systems which do not satisfy the restriction of secret choices being generated by a single component (Assumption 1). More precisely, we believe that our framework can be applied also to the case of systems in which secret choices can occur in more than one component, provided that they occur at the beginning of the code (as is often the case). The idea is to automatically transform the system into a new one satisfying Assumption 1, whose secret choice corresponds to the combination of the original ones, and in such a way that the new system is equivalent to the original one w.r.t. anonymity. The following example illustrates our intuition.

Example 6. Consider an electronic voting system EVS consisting of two components, P and Q, who can vote for two candidates 0 and 1. Each component selects the candidate by making a probabilistic choice:

$$\text{EVS} \;=\; (p : 0.P[0]) + ((1-p) : 1.P[1]) \parallel (q : 0.Q[0]) + ((1-q) : 1.Q[1]).$$

We can transform EVS in the following system EVS′ where the probabilistic choices are "centralized" and combined into one single choice, as follows:

$$\text{EVS}' \;=\; (c, d)\ \text{VotesGenerator} \parallel c(v).P[v] \parallel d(v).Q[v]$$

where

$$\begin{aligned}\text{VotesGenerator} \;=\;& (p\ q : \bar{c}\langle 0\rangle.\bar{d}\langle 0\rangle) + (p\ (1-q) : \bar{c}\langle 0\rangle.\bar{d}\langle 1\rangle) \\ &+ \\ & ((1-p)\ q : \bar{c}\langle 1\rangle.\bar{d}\langle 0\rangle) + ((1-p)\ (1-q) : \bar{c}\langle 1\rangle.\bar{d}\langle 1\rangle)\end{aligned}$$

We believe that EVS and EVS′ are equivalent, in the sense that EVS satisfies strong anonymity iff EVS′ satisfies strong anonymity.

Acknowledgements

The authors wish to thank Flavio Garcia, Pedro D'Argenio, Sergio Giro, and Mariëlle Stoelinga for useful comments on an earlier version of this paper, as well as the anonymous reviewers for thoroughly reading the paper and providing thoughtful recommendations. The first author is supported by NWO project 612.000.526. The last author is supported by the Austrian Science Fund (FWF), Project V00125.

References

[1] M.S. Alvim, M.E. Andrés, C. Palamidessi, P. van Rossum, Safe equivalences for security properties, in: C.S. Calude, V. Sassone (Eds.), Proceedings of the 6th IFIP International Conference on Theoretical Computer Science, TCS 2010, in: IFIP Advances in Information and Communication Technology, vol. 323, Springer, 2010, pp. 55–70.
[2] M.E. Andrés, C. Palamidessi, P. van Rossum, A. Sokolova, Information hiding in probabilistic concurrent systems, in: Proceedings of the 7th IEEE International Conference on Quantitative Evaluation of SysTems, (QEST 2010), IEEE Computer Society, 2010, pp. 17–26.
[3] M. Bhargava, C. Palamidessi, Probabilistic anonymity, in: M. Abadi, L. de Alfaro (Eds.), Proceedings of CONCUR, in: Lecture Notes in Computer Science, vol. 3653, Springer, 2005, pp. 171–185.
[4] C. Braun, K. Chatzikokolakis, C. Palamidessi, Compositional methods for information-hiding, in: R. Amadio (Ed.), Proceedings of FOSSACS, in: Lecture Notes in Computer Science, vol. 4962, Springer, 2008, pp. 443–457.
[5] C. Braun, K. Chatzikokolakis, C. Palamidessi, Quantitative notions of leakage for one-try attacks, in: Proceedings of the 25th Conf. on Mathematical Foundations of Programming Semantics, in: Electronic Notes in Theoretical Computer Science, vol. 249, Elsevier B.V, 2009, pp. 75–91.
[6] C. Cachin, Entropy measures and unconditional security in cryptography, Ph.D. Thesis, ETH, Zurich, 1997. Reprint as vol.1 of ETH Series in Information Security and Cryptography, ISBN 3-89649-185-7, Hartung-Gorre Verlag, Konstanz, 1997.
[7] R. Canetti, L. Cheung, D. Kaynar, M. Liskov, N. Lynch, O. Pereira, R. Segala, Task-structured probabilistic i/o automata, in: Proceedings the 8th International Workshop on Discrete Event Systems, WODES'06, Ann Arbor, Michigan, 2006.
[8] R. Canetti, L. Cheung, D.K. Kaynar, M. Liskov, N.A. Lynch, O. Pereira, R. Segala, Time-bounded task-PIOAs: a framework for analyzing security protocols, in: S. Dolev (Ed.), Proceedings of the 20th International Symposium in Distributed Computing, DISC'06, in: Lecture Notes in Computer Science, vol. 4167, Springer, 2006, pp. 238–253.
[9] K. Chatzikokolakis, G. Norman, D. Parker, Bisimulation for demonic schedulers, in: L. de Alfaro (Ed.), Proc. of the Twelfth Int. Conf. on Foundations of Software Science and Computation Structures, FOSSACS 2009, in: Lecture Notes in Computer Science, vol. 5504, Springer, York, UK, 2009, pp. 318–332.
[10] K. Chatzikokolakis, C. Palamidessi, Making random choices invisible to the scheduler, in: L. Caires, V.T. Vasconcelos (Eds.), Proceedings of the 18th International Conference on Concurrency Theory, CONCUR 2007, in: Lecture Notes in Computer Science, vol. 4703, Springer, 2007, pp. 42–58.
[11] K. Chatzikokolakis, C. Palamidessi, P. Panangaden, Anonymity protocols as noisy channels, Information and Computation 206 (2–4) (2008) 378–401.
[12] K. Chatzikokolakis, C. Palamidessi, P. Panangaden, On the Bayes risk in information-hiding protocols, Journal of Computer Security 16 (5) (2008) 531–571.
[13] D. Chaum, The dining cryptographers problem: unconditional sender and recipient untraceability, Journal of Cryptology 1 (1988) 65–75.

[14] D. Clark, S. Hunt, P. Malacaria, Quantified interference for a while language, in: Proceedings of the Second Workshop on Quantitative Aspects of Programming Languages, QAPL 2004, in: Electronic Notes in Theoretical Computer Science, vol. 112, Elsevier Science B.V, 2005, pp. 149–166.
[15] D. Clark, S. Hunt, P. Malacaria, Quantitative information flow, relations and polymorphic types, Journal of Logic and Computation 18 (2) (2005) 181–199.
[16] I. Clarke, O. Sandberg, B. Wiley, T.W. Hong, Freenet: a distributed anonymous information storage and retrieval system, in: International Workshop on Design Issues in Anonymity and Unobservability, in: Lecture Notes in Computer Science, vol. 2009, Springer, 2000, pp. 44–66.
[17] M.R. Clarkson, A.C. Myers, F.B. Schneider, Belief in information flow, Journal of Computer Security 17 (5) (2009) 655–701.
[18] T.M. Cover, J.A. Thomas, Elements of Information Theory, second edition, John Wiley & Sons, Inc., 2006.
[19] L. de Alfaro, T.A. Henzinger, R. Jhala, Compositional methods for probabilistic systems, in: K.G. Larsen, M. Nielsen (Eds.), Proceedings of the 12th International Conference on Concurrency Theory, CONCUR 2001, in: Lecture Notes in Computer Science, vol. 2154, Springer, 2001.
[20] S. Delaune, S. Kremer, M. Ryan, Verifying privacy-type properties of electronic voting protocols, Journal of Computer Security 17 (4) (2009) 435–487.
[21] P. Foggia, C. Sansone, M. Vento, A performance comparison of five algorithms for graph isomorphism, in: Proc. of the IAPR TC-15 Ws on Graph-based Representations in Pattern Recognition, 2001, pp. 188–199.
[22] F.D. Garcia, I. Hasuo, P. van Rossum, W. Pieters, Provable anonymity, in: R. Küsters, J. Mitchell (Eds.), Proceedings of the 2005 ACM Workshop on Formal Methods in Security Engineering, (FMSE'05), ACM, 2005, pp. 63–72.
[23] S. Giro, Undecidability results for distributed probabilistic systems, in: M.V.M. Oliveira, J. Woodcock (Eds.), 12th Brazilian Symposium on Foundations and Applications of Formal Methods, SBMF, in: Lecture Notes in Computer Science, vol. 5902, Springer, 2009, pp. 220–235.
[24] S. Giro, P.R. D'Argenio, Quantitative model checking revisited: neither decidable nor approximable, in: J.-F. Raskin, P.S. Thiagarajan (Eds.), Proceedings of the 5th International Conference on Formal Modeling and Analysis of Timed Systems, FORMATS, in: Lecture Notes in Computer Science, vol. 4763, Springer, 2007, pp. 179–194.
[25] J.Y. Halpern, K.R. O'Neill, Anonymity and information hiding in multiagent systems, Journal of Computer Security 13 (3) (2005) 483–512.
[26] I. Hasuo, Y. Kawabe, Probabilistic anonymity via coalgebraic simulations, in: Proceedings of the European Symposium on Programming, in: Lecture Notes in Computer Science, vol. 4421, Springer, Berlin, 2007, pp. 379–394.
[27] B. Köpf, D.A. Basin, An information-theoretic model for adaptive side-channel attacks, in: P. Ning, S.D.C. di Vimercati, P.F. Syverson (Eds.), Proceedings of the 2007 ACM Conference on Computer and Communications Security, CCS 2007, Alexandria, Virginia, USA, October 28–31, 2007, ACM, 2007, pp. 286–296.
[28] M.Z. Kwiatkowska, G. Norman, D. Parker, Symmetry reduction for probabilistic model checking, in: T. Ball, R.B. Jones (Eds.), Proceedings of the 18th International Conference on Computer Aided Verification, CAV 2006, in: Lecture Notes in Computer Science, vol. 4144, Springer, 2006, pp. 234–248.
[29] P. Malacaria, Assessing security threats of looping constructs, in: M. Hofmann, M. Felleisen (Eds.), Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2007, Nice, France, January 17–19, 2007, ACM, 2007, pp. 225–235.
[30] P. Malacaria, H. Chen, Lagrange multipliers and maximum information leakage in different observational models, in: Úlfar Erlingsson, Marco Pistoia (Eds.), Proceedings of the 2008 Workshop on Programming Languages and Analysis for Security, PLAS 2008, ACM, Tucson, AZ, USA, 2008, pp. 135–146.
[31] S. Mauw, J. Verschuren, E. de Vink, A formalization of anonymity and onion routing, in: P. Samarati, P. Ryan, D. Gollmann, R. Molva (Eds.), Proceedings of the European Symposium on Research in Computer Security, in: Lecture Notes in Computer Science, vol. 3193, 2004, pp. 109–124.
[32] R. Milner, Communication and Concurrency, in: International Series in Computer Science, Prentice Hall, 1989.
[33] R. Milner, Communicating and Mobile Systems: The π-Calculus, Cambridge University Press, 1999.
[34] M.K. Reiter, A.D. Rubin, Crowds: anonymity for Web transactions, ACM Transactions on Information and System Security 1 (1) (1998) 66–92.
[35] J.J. Rutten, Universal coalgebra: a theory of systems, Theoretical Computer Science 249 (2000) 3–80.
[36] R. Segala, Modeling and Verification of Randomized Distributed Real-Time Systems, Ph.D. Thesis, June 1995. Tech. Rep., MIT/LCS/TR-676.
[37] R. Segala, N. Lynch, Probabilistic simulations for probabilistic processes, Nordic Journal of Computing 2 (2) (1995) 250–273. An extended abstract appeared in Proceedings of CONCUR'94, Lecture Notes in Computer Science, vol. 836, pp. 481–496.
[38] G. Smith, On the foundations of quantitative information flow, in: L. de Alfaro (Ed.), Proc. of the 12th Int. Conf. on Foundations of Software Science and Computation Structures, in: Lecture Notes in Computer Science, vol. 5504, Springer, York, UK, 2009, pp. 288–302.
[39] P. Syverson, D. Goldschlag, M. Reed, Anonymous connections and onion routing, in: IEEE Symposium on Security and Privacy, Oakland, California, 1997, pp. 44–54.
[40] Y. Zhu, R. Bettati, Anonymity vs. information leakage in anonymity systems, in: Proc. of ICDCS, IEEE Computer Society, 2005, pp. 514–524.

Quantitative Relaxation of Concurrent Data Structures

Thomas A. Henzinger* Christoph M. Kirsch+ Hannes Payer+ Ali Sezgin* Ana Sokolova+

*IST Austria
{tah,asezgin}@ist.ac.at

+University of Salzburg
firstname.lastname@cs.uni-salzburg.at

Abstract

There is a trade-off between performance and correctness in implementing concurrent data structures. Better performance may be achieved at the expense of relaxing correctness, by redefining the semantics of data structures. We address such a redefinition of data structure semantics and present a systematic and formal framework for obtaining new data structures by quantitatively relaxing existing ones. We view a data structure as a sequential specification containing all "legal" sequences over an alphabet of method calls. Relaxing the data structure corresponds to defining a distance from *any* sequence over the alphabet to the sequential specification: the k-relaxed sequential specification contains all sequences over the alphabet within distance k from the original specification. In contrast to other existing work, our relaxations are semantic (distance in terms of data structure states). As an instantiation of our framework, we present two simple yet generic relaxation schemes, called out-of-order and stuttering relaxation, along with several ways of computing distances. We show that the out-of-order relaxation, when further instantiated to stacks, queues, and priority queues, amounts to tolerating bounded out-of-order behavior, which cannot be captured by a purely syntactic relaxation (distance in terms of sequence manipulation, e.g. edit distance). We give concurrent implementations of relaxed data structures and demonstrate that bounded relaxations provide the means for trading correctness for performance in a controlled way. The relaxations are monotonic, which further highlights the trade-off: increasing k increases the number of permitted sequences, which as we demonstrate can lead to better performance. Finally, since a relaxed stack or queue also implements a pool, we obtain new concurrent pool implementations that outperform the state-of-the-art ones.

Categories and Subject Descriptors D.3.1 [*Programming languages*]: Formal definitions and theory—semantics; E.1 [*Data Structures*]: Lists, stacks, and queues; D.1.3 [*Programming languages*]: Programming techniques—concurrent programing

General Terms Theory, Algorithms, Design, Performance

Keywords (concurrent) data structures, relaxed semantics, quantitative models, costs

POPL'13, January 23–25, 2013, Rome, Italy.

1. Introduction

Concurrent data structures may be a performance and scalability bottleneck and thus prevent effective use of increasingly parallel hardware [18]. There is a trade-off between scalability (performance) and correctness in implementing concurrent data structures. A remedy to the scalability problem is to relax the semantics of concurrent data structures. The semantics is given by some notion of equivalence with sequential behavior. The equivalence is determined by a consistency condition, most commonly linearizability [7], and the sequential behavior is inherited from the sequential version of the data structure (e.g., the sequential behavior of a concurrent stack is a regular stack). Therefore, relaxing the semantics of a concurrent data structure amounts to either weakening the consistency condition (linearizability being replaced with sequential consistency or quiescent consistency) or redefining (relaxing) its sequential specification. In this paper, we present a framework for relaxing sequential specifications in a quantitative manner.

For an example of a relaxation, imagine a k-stack in which each `pop` removes one of the most recent k elements and an operation `size` which returns a value that is at most k away from the correct size. It is intuitively clear that such a k-stack relaxes a regular stack, but current theory does not provide means to quantify the relaxation. Our framework does, it provides a way to formally describe and quantitatively assess such relaxations.

We view a data structure as a sequential specification S consisting of all semantically correct sequences of method calls. We identify the sequential specification with a particular labeled transition system (LTS) whose states are sets of sequences in S with indistinguishable future behavior and transitions are labeled by method calls. A sequence is in the sequential specification if and only if it is a finite trace of this LTS.

Our framework for quantitative relaxation of concurrent data structures amounts to specifying costs of transitions and paths. In the LTS, only *correct* transitions are allowed, e.g., a transition labeled by `pop(a)` is only possible in a state of a stack with `a` as top element. In a relaxation, we are exactly interested in allowing the *wrong* transitions, but they will have to incur cost. Our framework makes this possible in a controlled quantitative way.

The framework is instantiated through specifying two cost functions: A local function, *transition cost*, that assigns a penalty to each wrong transition, and a global function, *path cost*, that accumulates the local costs (using, e.g., maximum, sum, or average) to obtain the overall distance of a sequence. Via this local-global dichotomy, we are able to achieve a separation of concerns, modularity and flexibility: Different transition costs can be used with the same path cost, or vice versa, leading to different relaxations. Once the distance of a sequence from the original sequential specification S is defined in this way, a k-relaxation of the data structure becomes the set of all sequences within distance k from S.

Returning to the stack example above, we can set the transition cost of a `pop` transition at a state to be the number of elements that

are between the popped element and the top of the stack. We can define the path cost to be the maximum transition cost that occurs along a sequence. Then, the corresponding k-relaxation precisely captures what we intuitively described.

We instantiate the framework on two levels. On the abstract level, we present two generic relaxations called out-of-order and stuttering relaxation, which provide a way to assign transition costs, together with several different path cost functions for *any* data structure. On the concrete level, we instantiate the out-of-order relaxation to stacks, queues, and priority queues. We spell out the effects of the relaxation in these concrete cases and prove that they indeed correspond to the intuitive idea of bounded relaxed out-of-order behavior. We also instantiate the stuttering relaxation to a Compare-And-Swap (CAS) object and a shared counter and prove correspondence results as well.

We show that the relaxation framework is indeed of practical value: we give an efficient new implementation of an out-of-order stack and fit an existing efficient implementation of an out-of-order queue [11] in our framework as well. The experimental results demonstrate ideal behavior: linear scalability and performance. In particular, the relaxed stack implementation we present outperforms and outscales state-of-the-art stack, queue, and pool algorithms on various workloads. We also present implementations for a stuttering CAS, a stuttering shared counter using this stuttering CAS and a different stuttering shared counter, all of which further demonstrate increased scalability and performance.

The main contributions of this paper are: (1) the framework for quantitative relaxation of data structures, and (2) efficient concurrent implementations. The way to the framework is paved by formally capturing the semantics of a data structure. Other contributions made possible by the framework are: the generic out-of-order and stuttering relaxations of data structures; characterizations of the out-of-order relaxation in concrete terms for stacks, queues, and priority queues; characterization of the stuttering relaxation in concrete terms for CAS and shared counters.

The structure of the paper is as follows. In the remainder of this section, we provide motivation for the main features of our work. We present the formal view on data structures in Section 2, followed by the framework for quantitative relaxation in Section 3. Throughout the formal part we use a stack as running example. We present the two generic instances, out-of-order and stuttering relaxations, in Section 4 and instantiate them further to concrete data structures in Section 5 and Section 6, respectively. We discuss related work in Section 7. In Section 8 we present implementation details and in Section 9 experimental results confirming our original scalability and performance goal. We wrap up with concluding remarks in Section 10.

In the related work survey, we put special emphasis on quasi-linearizability [2], the only other work we are aware of that also tackled the problem of quantitatively relaxing sequential data structures for better performance in the concurrent setting. As opposed to our semantic (state-based) approach in assigning distances to sequences, the relaxation of [2] is syntactic (permutation-based). We argue that (1) the semantic approach is more expressive than the syntactic one, and (2) it allows the designer of a data structure to formally capture the intent of a specific relaxation more easily and naturally.

Highlights

Relaxation improves performance. A relaxation of the sequential specification of a data structure can lead to a distribution of contention points, diminishing the need for, and thus, the cost of synchronization. For instance, instead of requiring that each pop operation updates the top pointer of a concurrent stack, allowing a relaxation which sets the size of the window from which a removal is deemed acceptable to some $k > 1$ (most recent elements) effectively reduces contention for the top pointer. In Section 9, we show that even such a simple relaxation for stacks with $k = 80$ on a 40-core (2 hyperthreads per core) server machine can lead to an eight-fold increase in performance compared to the existing state-of-the-art implementations of strict stacks.

Note that a larger sequential specification increases the potential for better performance. Since our relaxations are monotonic, increasing k increases the performance potential. However, the extent to which this potential can be utilized in practice depends on many factors among which is the choice of hardware.

Generality. Consider three different sequences belonging to three different data structures, stack, queue, and priority queue, respectively:

```
push(a)push(b)push(c)push(d)
enq(a)enq(c)enq(b)enq(d)
ins(a)ins(b)ins(d)ins(c)
```

where for the priority queue `b` has top priority, followed by `a` and `c` that have the same medium priotity, and `d` has low priority.

If these sequences are extended with a removal operation (`pop`, `deq`, `rem`, respectively), the expected return values are `d` (element at the top of the stack), `a` (element at the head of the queue), and `b` (element with the highest priority).

Imagine instead, that the removal operation returns `c` for all of these three sequences. At first sight, that `c` is returned seems to be arbitrary. However, a careful examination reveals a common pattern: In each sequence, `c` is not the current, but *next* (possible) value to be removed. That is, in the stack it is the element immediately below the top element; in the queue it is the element immediately after the head element; in the priority queue it is an element with the second highest priority. It then seems natural to view all these relaxations as an instantiation of a common relaxation scheme.

Our framework allows one to precisely express this and other common types of relaxations. For instance, our generic out-of-order relaxation provides exactly this for data structures in which information is retrieved according to some order, temporal in the case of queue and stack, logical in the case of a priority queue. The generic relaxation removes the need of relaxing each data structure separately.

Modularity. Let us now consider the situation immediately following the removal of `c` from the stack as was depicted above. We have the following sequence:

```
push(a)push(b)push(c)push(d)pop(c)
```

The stack now contains the elements `a`, `b`, and `d`, the last of which is on top. One might desire a particular relaxation where two consecutive out-of-order removals are not allowed, and hence, the next removal has to return `d`. Yet another might find it acceptable that at all times one of the top two elements are removed; it does not matter how long the top element remains on the stack. Our framework allows one to express both. Each transition incurs a transition cost. Observe that in both relaxations the same cost (out-of-order removal cost) is assigned to each transition. Each sequence of transition costs incurs a path cost, and this is what distinguishes the two relaxations. The first will require that there are no two consecutive transitions with non-zero cost; the second will require that the maximum of any transition cost is at most 1. We thus obtain a modular framework in which existing relaxations can be tailored by modifying transition costs, path costs, or both.

Measurability. The code given in Figure 1(a) represents a CAS-based strict shared counter. The shared variable `c` is a counter, and each thread tries to increment the value of the counter. Representative of many concurrent implementations, this code leads to poor scalability as all threads trying to increment the counter will compete for access to `c`.

```
1 while(true):
2  x = c;
3  if (CAS(&c,x,x+1)):
4   return x+1;
```
(a) Single counter.

```
1 while(true):
2  x,v = getMaxAndValueAt(c, f(t));
3  if (CAS(&c[f(t)],v,x+1)):
4   return x+1;
```
(b) Distributed counter.

Figure 1. Shared counters

Next, consider a modified version of this shared counter, given in Figure 1(b). Unlike the strict implementation, here we use an array `c` of k counters and the logical value of the shared counter is taken to be the maximum value among all the counters in `c`. Each thread `t` can write only to the slot with index `f(t)`. Each attempt of `t` incrementing the counter starts by reading the value contained in `f(t)` (stored in `v`) and the maximum value of all the counters in `c` (stored in `x`). Then, it tries to update its counter to `x` + 1 provided that `f(t)` is not updated by a concurrent thread.

The behavior of this code depends crucially on the value chosen for the size k of the array. For instance, if $k = 1$, then this implementation will be behaviorally equivalent to the single counter code. For other values of k, it is evident that there will be a discrepancy between the behaviors of the two codes.

We go beyond this qualitative notion (existence vs. absence of relaxation) and provide a measure for any relaxation defined in our framework. The distributed counter given in Figure 1(b) is in fact a k-stuttering relaxation of the shared counter of Figure 1(a). This way an application developer using a quantitatively relaxed data structure can evaluate the gain in performance for k-relaxation vs. the effort of modifying an application that uses it, and try to optimize k. For instance, if the relaxed shared counter is used as a performance counter counting the occurrence of a given event, e.g. context switches in a multi-processor scheduler, not all occurrences of events will be registered. Knowing that the number of unregistered event occurrences within one counter increment can not exceed k (the size of `c`) is a crucial information for the application designer.

Transparency. Now consider the code given in Figure 2. This code is very similar to the strict shared counter code of Figure 1(a) except for the call to the method `kCAS` instead of `CAS`. The `kCAS` is a relaxed version of `CAS` such that up to at most k many concurrent threads trying to update the value of the `CAS` object can complete with false positive (see Section 6 and Section 8 for details).

Although the `kCAS` and the distributed counter of Figure 1(b) take fundamentally different approaches in relaxing the strict semantics of a counter, they both implement a k-stuttering relaxation. This illustrates another use of our framework: It can be used as a simpler way to establish abstract equivalence, thus providing transparency for a higher-level application. If one shows that two implementations implement the same relaxation, then a client application using either implementation will observe the same behavior, regardless of the differences in actual implementation details.

```
1 while(true):
2  x = c;
3  if (kCAS(&c,x,x+1)):
4   return x+1;
```

Figure 2. `kCAS` counter.

2. Data structures, specifications, states

Let Σ be a set of methods including input and output values. We will refer to Σ as the sequential alphabet. A *sequential history* $\mathbf{s}$ is an element of Σ^*, i.e., a sequence over Σ. As usual, by ε we denote the empty sequence in Σ^*. A *data structure* is a *sequential specification* S which is a prefix-closed set of sequential histories, $S \subseteq \Sigma^*$.

EXAMPLE 2.1. The set of methods of a stack, with data in a set D, is

$$\Sigma_S = \{\texttt{push(d)} \mid \texttt{d} \in D\} \cup \{\texttt{pop(d)} \mid \texttt{d} \in D \cup \{\texttt{null}\}\}.$$

The sequential specification S_S consists of all stack-valid sequences, i.e., sequences in which each `pop` pops the top of the stack and each `push` pushes an element at the top. For instance, the sequence $\mathbf{s}_S = \texttt{push(a)pop(a)push(b)}$ is in the sequential specification S_S, whereas the sequence $\mathbf{t}_S = \texttt{push(a)push(b)pop(a)}$ is not.

The following definition is the core of our way of capturing semantics. Let S be a sequential specification.

DEFINITION 2.2. *Two sequential histories* $\mathbf{s}, \mathbf{t} \in S$ *are* S-equivalent, *written* $\mathbf{s} \equiv_S \mathbf{t}$, *if for any sequence* $\mathbf{u} \in \Sigma^*$, $\mathbf{su} \in S$ *if and only if* $\mathbf{tu} \in S$.

It is clear that $\equiv_S$ is an equivalence relation. By $[\mathbf{s}]_S$ we denote the S-equivalence class of $\mathbf{s}$. Intuitively, two sequences in the sequential specification are S-equivalent if they lead to the same "state". The following simple property follows directly from the definition of S-equivalence.

LEMMA 2.3. *If* $\mathbf{s} \equiv_S \mathbf{t}$ *and* $\mathbf{su} \in S$, *then* $\mathbf{su} \equiv_S \mathbf{tu}$.

The intuition about states is made explicit in the next definition. In addition, we point out particular "minimal" representatives of a state.

DEFINITION 2.4. *A state of a data structure with sequential specification* S *is an equivalence class* $[\mathbf{s}]_S$ *with respect to* $\equiv_S$. *For a state* $q = [\mathbf{s}]_S$, *the kernel of* q *is the set*

$$\ker(q) = \{\, \mathbf{t} \in [\mathbf{s}]_S \mid \mathbf{t} \textit{ has minimal length } \}.$$

A sequence $\mathbf{s} \in S$ *is a kernel sequence if* $\mathbf{s} \in \ker([\mathbf{s}]_S)$.

EXAMPLE 2.5. One can easily show that kernel sequences of a stack are all sequences in $\{\texttt{push(d)} \mid d \in D\}^*$. Moreover, for any state $q = [\mathbf{s}]_{S_S}$ of a stack, there is a unique sequence in $\ker(q)$, i.e., $|\ker(q)| = 1$. This implies that different sequences in $\mathbf{s} \in \{\texttt{push(d)} \mid \texttt{d} \in D\}^*$ represent different states.

Having identified states, a data structure corresponds to a labeled transition system (LTS) that we define next.

DEFINITION 2.6. *Let* S *be a (sequential specification of a) data structure. Its corresponding LTS is* $\mathrm{LTS}(S) = (Q, \Sigma, \rightarrow, q_0)$ *with*

- *set of states* $Q = S/\equiv_S = \{\,[\mathbf{s}]_S \mid \mathbf{s} \in S\}$,
- *set of labels* Σ,
- *transition relation* $\rightarrow \subseteq Q \times \Sigma \times Q$ *given by*

$$[\mathbf{s}]_S \xrightarrow{m} [\mathbf{s}m]_S \textit{ if and only if } \mathbf{s}m \in S, \textit{ and}$$

- *initial state* $q_0 = [\varepsilon]_S$.

Note that the transition relation is well defined (independent of the choice of a representative) due to Lemma 2.3. Also q_0 is well defined since S is prefix closed. We write $q \xrightarrow{m}$ if there is an m-labeled transition from q to some state; $q \overset{m}{\nrightarrow}$ if there is no m-labeled transition from q. We also write $q \xrightarrow{\mathbf{u}}$ if there is a $\mathbf{u}$-labeled path of transitions starting from q, and $q \overset{\mathbf{u}}{\nrightarrow}$ if it is not the case that $q \xrightarrow{\mathbf{u}}$. The following immediate observation provides the exact correspondence between the sequential specification of a data structure and its LTS: S is the set of finite traces of the initial state of $LTS(S)$.

LEMMA 2.7. *Let* S *be a sequential specification with* $\mathrm{LTS}(S) = (Q, \Sigma, \rightarrow, q_0)$. *Then for any* $\mathbf{u} \in \Sigma^*$ *we have* $\mathbf{u} \in S$ *if and only if* $q_0 \xrightarrow{\mathbf{u}}$.

EXAMPLE 2.8. Since different stack-kernel sequences represent different states, cf. Example 2.5, the transitions of $LTS(S_S)$ are fully described by

$$[\mathbf{s}]_{S_S} \xrightarrow{\texttt{push(a)}} [\mathbf{s}\cdot\texttt{push(a)}]_{S_S}$$
$$[\mathbf{s}]_{S_S} \xrightarrow{\texttt{pop(a)}} [\mathbf{s}']_{S_S} \quad \text{if } \mathbf{s} = \mathbf{s}'\cdot\texttt{push(a)}, \text{ and}$$
$$[\mathbf{s}]_{S_S} \xrightarrow{\texttt{pop(null)}} [\varepsilon]_{S_S} \text{ if } \mathbf{s} = \varepsilon$$

where $\mathbf{s}$ is a kernel sequence in $\{\texttt{push(d)} \mid d \in D\}^*$. Note that if $\mathbf{s} = \mathbf{s}'\cdot\texttt{push(a)}$, then $[\mathbf{s}\cdot\texttt{pop(a)}]_{S_S} = [\mathbf{s}']_{S_S}$.

3. Framework for quantitative relaxations

We are now ready to present the framework for quantitatively relaxing data structures. Let $S \subseteq \Sigma^*$ be a data structure with $\mathrm{LTS}(S) = (Q, \Sigma, \to, q_0)$. Our goal is to relax S to a so-called k-relaxed specification $S_k \subseteq \Sigma^*$ in a bounded way, with k providing the bound.

Giving a relaxation for a data structure S amounts to the following three steps:

1. **Completion.** From $\mathrm{LTS}(S) = (Q, \Sigma, \to, q_0)$ we construct the completed labeled transition system
$$\mathrm{LTS}_c(S) = (Q, \Sigma, Q \times \Sigma \times Q, q_0)$$
with transitions from any state to any other state by any method.
2. **Transition costs.** From $\mathrm{LTS}_c(S)$ a quantitative labeled transition system $\mathrm{QLTS}(S) = (Q, \Sigma, Q \times \Sigma \times Q, q_0, C, \mathrm{cost})$ is constructed. Here C is a well-ordered cost domain, hence it has a minimum that we denote by 0, and $\mathrm{cost}\colon Q \times \Sigma \times Q \to C$ is the transition cost function satisfying
$$\mathrm{cost}(q, m, q') = 0 \quad \text{if and only if} \quad q \xrightarrow{m} q' \text{ in } \mathrm{LTS}(S).$$
We write $q \xrightarrow{m,k} q'$ for the quantitative transition with $\mathrm{cost}(q, m, q') = k$. A quantitative path of $\mathrm{QLTS}(S)$ is a sequence
$$\kappa = q_1 \xrightarrow{m_1,k_1} q_2 \xrightarrow{m_2,k_2} q_3 \ldots q_n \xrightarrow{m_n,k_n} q_{n+1}.$$
The sequence $\tau = (m_1,k_1)(m_2,k_2)\ldots(m_n,k_n) \in (\Sigma \times C)^*$ is the quantitative trace of κ, notation $\mathrm{qtr}(\kappa)$, and the sequence $\mathbf{u} = m_1 \ldots m_n$ is the trace of the quantitative path κ and of the quantitative trace $\mathrm{qtr}(\kappa)$, notation $\mathrm{tr}(\kappa) = \mathrm{tr}(\mathrm{qtr}(\kappa)) = \mathbf{u}$. By $\mathrm{qtr}(\mathbf{u})$ we denote the set of all quantitative traces of quantitative paths starting in the initial state with trace $\mathbf{u}$ and by $\mathrm{qtr}(S)$ the set of all quantitative traces of quantitative paths starting in the initial state.
3. **Path cost function.** We choose a monotone path cost function $\mathrm{pcost}\colon \mathrm{qtr}(S) \to C$. Monotonicity here is with respect to prefix order: if a quantitative trace τ is a prefix of a quantitative trace τ', then $\mathrm{pcost}(\tau) \leqslant \mathrm{pcost}(\tau')$.

Having performed these three steps, we can define the k-relaxed specification.

DEFINITION 3.1. *The k-relaxed specification S_k for $k \in C$ contains all sequences that have a distance at most k from S,*

$$S_k = \{\mathbf{u} \in \Sigma^* \mid d_S(\mathbf{u}) \leqslant k\}$$

where $d_S(\mathbf{u})$ is the distance of $\mathbf{u}$ to the sequential specification S given by

$$d_S(\mathbf{u}) = \min\{\mathrm{pcost}(\tau) \mid \tau \in \mathrm{qtr}(\mathbf{u})\}.$$

REMARK 3.2. Both the distance d_S and the relaxed specification S_k are actually parametric in the transition cost function as well as in the path cost function. For simplicity, we prefer a light, overloaded notation that does not explicitly mention these parameters. Also, for some applications one may wish for two different cost domains, one for the transition, one for the path cost, of which only the second one needs to be well ordered. Again for simplicity, we restrict the presentation to a single cost domain.

Some obvious properties of the quantified relaxations resulting from our framework are:

- $S_0 = S$, ensured by the condition on the transition cost function.
- Every relaxation S_k is prefix closed, ensured by the monotonicity of the path cost function.
- The relaxations are monotone, i.e., if $k \leqslant m$, then $S_k \subseteq S_m$.

To conclude, in order to relax a data structure all that one needs is a cost domain C, a transition cost for each transition in the completed LTS (item 2. above), and a path cost function (item 3. above).

REMARK 3.3. The current framework does not allow for relaxations that leave the original state space of $LTS(S)$. An example of such is a prophetic relaxation of e.g. stack, where sequences with $\texttt{pop(a)}$ preceding $\texttt{push(a)}$ need to be assigned finite distance. This can be done by slightly changing the definition of $LTS_c(S)$: Instead of keeping the original states $S/\equiv_S$, one can take as set of states the quotient $\Sigma^*/\sim$ where $\sim$ is an equivalence that coincides with $\equiv_S$ when restricted to S. For simplicity and since we do not use such relaxations in this paper, our current definition of $LTS_c(S)$ keeps the states unchanged.

4. Generic relaxations

In this section we illustrate the relaxation framework on two generic examples. The value and generality of these particular examples become evident in Section 5 and Section 6 when we instantiate them to concrete data structures. Let $S \subseteq \Sigma^*$ be a data structure with $\mathrm{LTS}(S) = (Q, \Sigma, \to, q_0)$. We first fix the cost domain to $C = \mathbb{N} \cup \{\infty\}$.

4.1 Out-of-order relaxation

For the out-of-order generic relaxation we define a transition cost function $scost\colon Q \times \Sigma \times Q \to C$, called *segment cost*, and mention two other related transition cost functions.

DEFINITION 4.1. *Let $t = (q, m, q')$ be a transition in $\mathrm{LTS}_c(S)$. Let $\mathbf{v}$ be a sequence with minimal length satisfying one of the following two conditions:*

(1) There exist sequences $\mathbf{u}$, $\mathbf{w}$ such that $\mathbf{uvw} \in \ker(q)$ and $\mathbf{uw}$ is a kernel sequence and either
(i) $[\mathbf{uw}]_S \xrightarrow{m} [\mathbf{u'w}]_S$ and $q' = [\mathbf{u'vw}]_S$, or
(ii) $[\mathbf{uw}]_S \xrightarrow{m} [\mathbf{uw'}]_S$ and $q' = [\mathbf{uvw'}]_S$.
(2) There exist sequences $\mathbf{u}$, $\mathbf{w}$ such that $\mathbf{uw} \in \ker(q)$ and $\mathbf{uvw}$ is a kernel sequence and either
(i) $[\mathbf{uvw}]_S \xrightarrow{m} [\mathbf{u'vw}]_S$ and $q' = [\mathbf{u'w}]_S$, or
(ii) $[\mathbf{uvw}]_S \xrightarrow{m} [\mathbf{uvw'}]_S$ and $q' = [\mathbf{uw'}]_S$.

Then the segment cost is given by the length of $\mathbf{v}$, $scost(t) = |\mathbf{v}|$. If such a sequence $\mathbf{v}$ does not exist for t, then $scost(t) = \infty$.

Intuitively, segment cost of a relaxed transition is the length of the shortest subword ($\mathbf{v}$) whose removal (1) or insertion (2) into the kernel sequence enables a transition. Observe that the transition can be taken in $\mathrm{LTS}(S)$ if and only if its segment cost is 0, obtained by setting $\mathbf{v} = \varepsilon$. We will see in the next section that this cost quantifies *out-of-order* updates or observations, such as returning an element other than the top element in a stack or removing an element other than the head of a queue. We note that segment cost just as any

transition cost can also be used per method, i.e., some methods may be relaxed, some not.

4.2 Stuttering relaxation

For the stuttering generic relaxation, we define the so-called *stuttering cost*.

DEFINITION 4.2. *Let $t = (q,m,q')$ be a transition in* $\mathrm{LTS}_c(S)$. *Then, the* stuttering cost*, stcost is defined as*

$$stcost(q,m,q') = \begin{cases} 0 & \text{if } q \xrightarrow{m} q' \\ 1 & \text{if } q = q' \wedge q \overset{m}{\nrightarrow} q' \wedge q \xrightarrow{m} \\ \infty & \text{otherwise} \end{cases}$$

where $\rightarrow$ is the transition relation of $LTS(S)$.

Intuitively, the stuttering relaxation allows for (already enabled) transitions to have no effect on the state. If, in the specification S, q goes to $\hat{q}$ with method m and $q \neq \hat{q}$, then the stuttering cost of applying m at q *and* staying at q after the transition is 1. All other transitions which are not part of the original specification are set to have infinite cost.

An example of an unbounded (except in the maximal size of the queue) stuttering relaxation is presented in [15], where workers are allowed to work on the same task by a relaxed queue semantics and an element in the queue can be dequeued a number of times (up to the maximal size of the queue). In favor of bounded stuttering we note that, typically, implementations can benefit from retiring (completing rather than retrying) mutator method calls when there is too much contention and have the client handle false positives.

4.3 Path cost functions

Let $S \subseteq \Sigma^*$ be a data structure and $\tau = (m_1,k_1)(m_2,k_2)\dots(m_n,k_n)$ a quantitative trace in $\mathrm{qtr}(S)$. We define the following generic path cost functions (to be used with any transition cost):

- The *maximal cost*, $\mathrm{pcost}_{max}\colon \mathrm{qtr}(S) \to \mathbb{N} \cup \{\infty\}$, maps τ to the maximal transition cost along it. Formally,
$$\mathrm{pcost}_{max}(\tau) = \max\{k_i \mid 1 \leqslant i \leqslant n\}.$$
- The φ-*interval cost*, $\mathrm{pcost}_{[\varphi]}\colon \mathrm{qtr}(S) \to \mathbb{N} \cup \{\infty\}$, for φ a binary predicate (first order formula with two free variables making statements about positions in the quantitative trace), maps τ to the length of a maximal consecutive quantitative subtrace that satisfies φ. Hence, we have
$$\mathrm{pcost}_{[\varphi]}(\tau) = \max\{j-i+1 \mid \varphi(i,j) \text{ and } 1 \leqslant i \leqslant j \leqslant n\}.$$
- The φ-*interval restricted maximal cost*, $\mathrm{pcost}_{max|[\varphi]}\colon \mathrm{qtr}(S) \to \mathbb{N} \cup \{\infty\}$, for φ as in the φ-interval cost, is given by
$$\mathrm{pcost}_{max|[\varphi]}(\tau) = \max\{l_{i,j} \mid \varphi(i,j) \text{ and } 1 \leqslant i \leqslant j \leqslant n\},$$
where
$$l_{i,j} = \max\{k_r + (r-i+1) \mid i \leqslant r \leqslant j\}.$$

We instantiate the out-of-order relaxation along with the maximal, φ-interval, and φ-interval restricted maximal cost on stacks, queues, and priority queues in Section 5. The φ-interval restricted maximal cost is more complex and less intuitive than the other path cost functions, but when instantiated it provides valuable relaxation examples that are efficiently implementable. In Section 6 we apply the stuttering relaxation along with the φ-interval cost on a CAS object and on a shared counter. Note that the other two cost functions do not make much sense together with the stuttering cost (the maximal cost is two-valued and the φ-interval restricted maximal cost amounts to the φ-interval cost plus one).

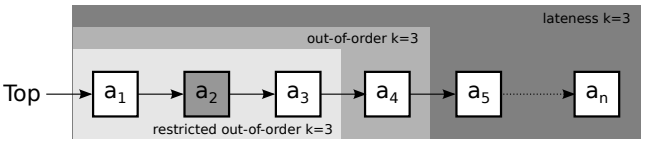


Figure 3. The ranges of elements which may be returned by a pop operation of a k-stack with restricted out-of-order, out-of-order, and lateness relaxation with $k = 3$. The element a_2 is already removed.

5. Out-of-order stacks, queues, and priority queues

In this section we apply the relaxation of Section 4.1 to stacks, FIFO queues, and priority queues. Due to lack of space, here we leave out some common methods, e.g., `top` (for stack), `head` (for queue), `size` (for all). Inclusion of these methods does not change the results, in particular Propositions 5.1-5.3, presented in this section.

Stack. We have already given the set of methods of a stack, its states, and its LTS in Example 2.1, Example 2.5, and Example 2.8. Let us recall that the sequential specification S_S consists of all stack-valid sequences, i.e., sequences in which each `pop` pops the top of the stack, and each `push` pushes an element at the top.
Let $\mathbf{s}$ be a kernel sequence. A kernel sequence $\mathbf{s}'$ is

- $\mathtt{push(a)}$-out-of-order-k from $\mathbf{s}$ if $\mathbf{s}' = \mathbf{u} \cdot \mathtt{push(a)} \cdot \mathbf{v}$ where $\mathbf{s} = \mathbf{uv}$, $\mathbf{v}$ is minimal, and $|\mathbf{v}| = k$;
- $\mathtt{pop(a)}$-out-of-order-k from $\mathbf{s}$ if $\mathbf{s}' = \mathbf{uv}$ where $\mathbf{s} = \mathbf{u} \cdot \mathtt{push(a)} \cdot \mathbf{v}$, $\mathbf{v}$ is minimal, and $|\mathbf{v}| = k$;
- $\mathtt{pop(null)}$-out-of-order-k from $\mathbf{s}$ if $\mathbf{s} = \mathbf{s}'$ and $|\mathbf{s}| = k$;

By inspecting all cases, we can show the following proposition.

PROPOSITION 5.1. *Let $\mathbf{s}$ and $\mathbf{s}'$ be two kernel sequences of a stack. Then $[\mathbf{s}]_{S_S} \overset{m,k}{\rightarrow} [\mathbf{s}']_{S_S}$ in the out-of-order relaxation with segment cost if and only if $\mathbf{s}'$ is m-out-of-order-k from $\mathbf{s}$.*

As mentioned in Section 4.1, the relaxations can be applied method-wise. We implemented k-relaxed stacks with only `push` and `pop` methods, of which only `pop` is relaxed according to the segment cost.The interpretation of the path cost functions from Section 4.3 and the corresponding relaxations are as follows:

- The maximal cost represents the maximal distance from the top of a popped element, leading to an *out-of-order k-stack*. Hence, in an out-of-order k-stack, each `pop` pops an element that is at most k away from the top.

Let $\varphi(i,j)$ be the following first order formula with free variables i and j:

$$\forall r \in [i,j].\, k_r \neq 0$$

- The φ-interval cost represents lateness, i.e., the maximal number of consecutive pops needed to pop the top, leading to a *lateness k-stack*. Hence, in a lateness k-stack at most the k-th consecutive pop pops the top.
- The φ-interval restricted maximal cost represents the maximal size of a "shrinking window" starting from the top from which elements can be popped, leading to a *restricted out-of-order k-stack*. In a restricted out-of-order k-stack, each pop removes an element at most $k-l$ away from the top, where l is the current lateness of the top.

Figure 3 presents a snapshot of a relaxed stack in each of the three out-of-order relaxations. It shows a state of a stack in which the element a_2, marked in grey, has been removed after the last

removal of the top or the last push had happened. The ranges show which elements may be returned by a pop operation applied to this state in each out-of-order relaxed version for $k = 3$.

FIFO queue. We now briefly describe the out-of-order relaxation of a queue.The set of methods for a FIFO queue, with data set D, is

$$\Sigma_Q = \{\texttt{enq(d)} \mid \texttt{d} \in D\} \cup \{\texttt{deq(d)} \mid \texttt{d} \in D \cup \{\texttt{null}\}\}.$$

The sequential specification S_Q consists of all queue-valid sequences, i.e., sequences in which each `deq` deques the head of the queue and each `enq` enqueues at the tail of the queue. For instance, the following sequence $\mathbf{s}_Q = \texttt{enq(a)enq(b)deq(a)}$ is in the sequential specification S_Q, whereas the sequence $\mathbf{t}_Q = \texttt{enq(a)enq(b)deq(b)}$ is not.

One can easily show that kernel sequences of a FIFO queue are all sequences in $\{\texttt{enq(d)} \mid \texttt{d} \in D\}^*$. Moreover, also here, for any state $q = [\mathbf{s}]_{S_Q}$ of the FIFO queue, there is a unique sequence in $\ker(q)$, i.e., $|\ker(q)| = 1$. Hence different sequences in $\mathbf{s} \in \{\texttt{enq(d)} \mid \texttt{d} \in D\}^*$ represent different states. As a consequence, the transition relation of $\mathrm{LTS}(S_Q)$ can be described in a concise way. Let $\mathbf{s}$ be a kernel sequence of a queue. We have,

$$\begin{aligned}
&[\mathbf{s}]_{S_Q} \xrightarrow{\texttt{enq(a)}} [\mathbf{s} \cdot \texttt{enq(a)}]_{S_Q}, \\
&[\mathbf{s}]_{S_Q} \xrightarrow{\texttt{deq(a)}} [\mathbf{s}']_{S_Q} \quad \text{if } \mathbf{s} = \texttt{enq(a)} \cdot \mathbf{s}', \text{ and} \\
&[\mathbf{s}]_{S_Q} \xrightarrow{\texttt{deq(null)}} [\varepsilon]_{S_Q} \text{ if } \mathbf{s} = \varepsilon.
\end{aligned}$$

In a similar way as for stack, we can define when a queue kernel sequence is m-out-of-order-k from another kernel sequence, for m being a queue method. Furthermore, the analogue of Proposition 5.1 (obtained by replacing "stack" by "FIFO queue") holds for queues as well which we state below.

PROPOSITION 5.2. *Let* $\mathbf{s}$ *and* $\mathbf{s}'$ *be two kernel sequences of a queue. Then* $[\mathbf{s}]_{S_Q} \xrightarrow{m,k} [\mathbf{s}']_{S_Q}$ *in the out-of-order relaxation with segment cost if and only if* $\mathbf{s}'$ *is m-out-of-order-k from* $\mathbf{s}$*.*

The maximal path cost function leads to analogous out-of-order k-queue. For lateness and restricted out-of-order k-queues we need to employ slightly different path cost functions.

Priority queue. The data set of a priority queue needs to be well-ordered, since data items carry priority as well. We take the data set to be $\mathbb{N}$. The smaller the number, the higher the priority. The set of methods is

$$\Sigma_{\mathcal{P}} = \{\texttt{ins(n)} \mid \texttt{n} \in \mathbb{N}\} \cup \{\texttt{rem(n)} \mid \texttt{n} \in \mathbb{N} \cup \{\texttt{null}\}\}.$$

The sequential specification $S_{\mathcal{P}}$ consists of all priority-queue-valid sequences, i.e., sequences in which each `rem` removes an element with highest available priority.

Kernel sequences of a priority queue are all sequences in $\{\texttt{ins(n)} \mid \texttt{n} \in \mathbb{N}\}^*$. Unlike for stack and queue, there may be more than one sequence representing a state of a priority queue. For a state q, if $\mathbf{s} \in \ker(q)$, then also any permutation of $\mathbf{s}$ is in $\ker(q)$. Nevertheless, the order provides a canonical representative of a state: the unique kernel sequence ordered in non-increasing priority[1]. Let $\mathbf{s}$ be a canonical kernel sequence. The transitions of $LTS(S_{\mathcal{P}})$ are fully described by

$$\begin{aligned}
&[\mathbf{s}]_{S_{\mathcal{P}}} \xrightarrow{\texttt{ins(n)}} [\mathbf{s} \cdot \texttt{ins(n)}]_{S_{\mathcal{P}}}, \\
&[\mathbf{s}]_{S_{\mathcal{P}}} \xrightarrow{\texttt{rem(n)}} [\mathbf{s}']_{S_{\mathcal{P}}} \quad \text{if } \mathbf{s} = \texttt{ins(n)} \cdot \mathbf{s}', \text{ and} \\
&[\mathbf{s}]_{S_{\mathcal{P}}} \xrightarrow{\texttt{rem(null)}} [\varepsilon]_{S_{\mathcal{P}}} \text{ if } \mathbf{s} = \varepsilon.
\end{aligned}$$

Again, we define when a canonical kernel sequence is m-out-of-order-k from another canonical kernel sequence, where m is a priority queue method. We have the following result.

PROPOSITION 5.3. *Let* $\mathbf{s}$ *and* $\mathbf{s}'$ *be two kernel sequences of a priority queue. Then* $[\mathbf{s}]_{S_{\mathcal{P}}} \xrightarrow{m,k} [\mathbf{s}']_{S_{\mathcal{P}}}$ *in the out-of-order relaxation with segment cost if and only if* $\mathbf{s}'$ *is m-out-of-order-k from* $\mathbf{s}$*.*

Analogous relaxations are again possible, only the path cost functions are more complex since they need to capture when an element with higher priority than all existing elements in the priority queue is inserted.

6. Stuttering relaxed CAS and shared counter

In this section, we instantiate the relaxation from Section 4.2 to two concrete examples.

CAS. The set of methods for a Compare-And-Swap (CAS) object with a data set D and an initial data value $\texttt{init} \in D$ can, for our purposes, be modeled as

$$\Sigma_{CAS} = \{\texttt{cas(d,d',b)} \mid \texttt{d,d'} \in D, \texttt{b} \in \{\texttt{T},\texttt{F}\}\}.$$

The sequential specification S_{CAS} is defined inductively as follows: The empty sequence ε is in S_{CAS} and any sequence of length one and shape $\texttt{cas(init,d,T)}$ is in S_{CAS} for $\texttt{d} \in D$. If $\mathbf{s} \in S_{CAS}$, let $\mathbf{t}$ be the maximal prefix of $\mathbf{s}$ such that $\mathbf{s} = \mathbf{t} \cdot m \cdot \mathbf{u}$ for $m = \texttt{cas(d,d',T)}$. Then $\mathbf{s} \cdot m' \in S_{CAS}$ if either

(1) $m' = \texttt{cas(d',d'',T)}$, or

(2) $m' = \texttt{cas(d'',d''',F)}$ and $\texttt{d}'' \neq \texttt{d}'$.

Let $\mathbf{s} \in S_{CAS}$ and let as before $\mathbf{t}$ be the maximal prefix of $\mathbf{s}$ such that $\mathbf{s} = \mathbf{t} \cdot m \cdot \mathbf{u}$ for $m = \texttt{cas(d,d',T)}$. Then, it is not difficult to show that, $s \equiv \texttt{cas(init,d',T)}$. Hence, there is a unique kernel sequence in each equivalence class and it has length one. The transitions of $LTS(S_{CAS})$ are given by

$$\begin{aligned}
&[\varepsilon]_{S_{CAS}} \xrightarrow{\texttt{cas(init,d,T)}} [\texttt{cas(init,d,T)}]_{S_{CAS}}, \\
&[\texttt{cas(init,d,T)}]_{S_{CAS}} \xrightarrow{\texttt{cas(d,d',T)}} [\texttt{cas(init,d',T)}]_{S_{CAS}}, \text{ and} \\
&[\texttt{cas(init,d,T)}]_{S_{CAS}} \xrightarrow{\texttt{cas(d',d'',F)}} [\texttt{cas(init,d,T)}]_{S_{CAS}} \text{ if } \texttt{d} \neq \texttt{d}'.
\end{aligned}$$

Intuitively, the state of the CAS object is given by one data value `d`, initially set to `init`. In such a state, a transition by method $\texttt{cas(d,d',T)}$ is enabled since the comparison of the first argument and the current value succeeds (returns `T`, true) leading to the new state value $\texttt{d}'$, that is, a successful comparison results in a swapped value. A transition by method $\texttt{cas(d',d'',F)}$ in which the comparison fails (returns `F`, false) is enabled if indeed $\texttt{d} \neq \texttt{d}'$ after which no swap happens and the state value remains `d`. The state with data value `d` is formally represented by the equivalence class of a sequence with a single method $\texttt{cas(init,d,T)}$.

Now let us formalize the notion of allowing invisible failures for CAS object updates. For this purpose we define another object called failCAS over the same set of methods, with a somewhat different set of legal sequences. We call a sequence

$$\texttt{cas(d,d',T)} \cdot \mathbf{y} \cdot \texttt{cas(d,d'',T)}$$

over Σ_{CAS} a *false positive sequence* if $\texttt{cas(d,d',T)} \cdot \mathbf{y} \in S_{CAS}$, $\mathbf{y}$ is a sequence of symbols of the form $\texttt{cas(d''',}-,\texttt{F)}$ with $\texttt{d}''' \neq \texttt{d}$, and $\texttt{d} \neq \texttt{d}'$. Then, $\mathbf{s} = s_0 \ldots s_n \in S_{failCAS}$ if there exists a set of

[1] The canonical representative is a matter of choice. Equally justified is using the unique kernel sequence ordered in non-decreasing priority, in which case the transitions of a priority queue resemble more the transitions of a stack, highlighting the duality between FIFO queues and stacks.

positions $0 = i_0 < i_1 < \ldots < i_r = n$ for $\mathbf{s}$ such that each sequence $s_{i_{j-1}} \ldots s_{i_j}$, for $1 \leqslant j \leqslant r$, is either a false positive sequence or is in S_{CAS}. Let the *failure count* of $\mathbf{x} \in S_{failCAS}$ be the maximum number of consecutive false positive sequences $\mathbf{x}$ contains.

The corresponding relaxation in our framework is obtained by using stuttering cost on the CAS object and φ-interval cost for the predicate $\varphi(i,j)$ given by

$$\forall r \in [i,j].(k_r \neq 0 \vee \exists \mathtt{d}, \mathtt{d}' \in D.(m_r = \mathtt{cas}(\mathtt{d}, \mathtt{d}', \mathtt{F}))),$$

leads to a k-CAS in which up to k methods may stutter at the same state (fail to perform a swap even though the data values match). It is then easy to show the following correspondence result.

PROPOSITION 6.1. *A sequence $\mathbf{x} \in S_{failCAS}$ has failure count k if and only if $\mathbf{x}$ is in the specification of k-CAS.*

Shared counter. The set of methods of a shared counter is

$$\Sigma_{SC} = \{\mathtt{get\&Inc(n)} \mid \mathtt{n} \in \mathbb{N}\}.$$

The sequential specification S_{SC} of a shared counter contains the empty sequence ε and a sequence $\mathbf{s}$ of length $n > 0$ is in S_{SC} if and only if $\mathbf{s}(\mathtt{i}) = \mathtt{get\&Inc(i)}$, for all $1 \leqslant \mathtt{i} \leqslant n$.

One can easily show that each state of a shared counter is a singleton, i.e., for $\mathbf{s}, \mathbf{t} \in S_{SC}$ we have $\mathbf{s} \equiv \mathbf{t}$ if and only if $\mathbf{s} = \mathbf{t}$. The unique sequence representing a state is automatically a kernel sequence. The transitions of $LTS(S_{SC})$ are obviously given by

$$[\varepsilon]_{S_{SC}} \xrightarrow{\mathtt{get\&Inc(1)}} [\mathtt{get\&Inc(1)}]_{S_{SC}}, \text{ and}$$

$$[\mathbf{s}]_{S_{SC}} \xrightarrow{\mathtt{get\&Inc(n+1)}} [\mathbf{s} \cdot \mathtt{get\&Inc(n+1)}]_{S_{SC}},$$

if $\mathbf{s}(\mathtt{i}) = \mathtt{get\&Inc(i)}$, for all $1 \leqslant \mathtt{i} \leqslant n$.

We define a failing shared counter analogously to a failing CAS object. A sequence $\mathbf{s}$ is a behavior of failSC if either $\mathbf{s} \in \{\varepsilon, \mathtt{get\&Inc(1)}\}$, or $\mathbf{t} = \mathbf{u} \cdot \mathtt{get\&Inc(n)}$ is a behavior of failSC and $\mathbf{s} = \mathbf{t} \cdot \mathtt{get\&Inc(n+}a\mathtt{)}$, for $a \in \{0,1\}$. The *failure count* of $\mathbf{s} \in S_{failSC}$ is one less than the length of a maximal subsequence of identical symbols in $\mathbf{s}$.

The corresponding relaxation of the shared counter is obtained by the stuttering cost and the φ-interval cost, for $\varphi(i,j) = \forall r \in [i,j].k_r \neq 0$. Thus, we get the k-stuttering shared counter, k-SC, in which a method can stutter (fail to produce a new, incremented by one, value) at most k times. For instance, the sequence

$$\mathtt{get\&Inc(1)get\&Inc(2)get\&Inc(2)get\&Inc(2)get\&Inc(3)}$$

is in the sequential specification of the 2-stuttering shared counter, 2-SC. We again have the following correspondence result.

PROPOSITION 6.2. *A sequence $\mathbf{x} \in S_{failSC}$ has failure count k if and only if $\mathbf{x}$ is in the specification of k-SC.*

7. Related work

The general topic of this paper is part of a recent trend towards scalable but semantically weaker concurrent data structures [18]. We first discuss work related to our framework and then focus on work related to our implementations.

Framework. The relaxation framework generalizes previous work on so-called semantical deviation and k-FIFO queues [10, 12] which correspond to restricted out-of-order k-FIFO queues here.

Our work is closely related to relaxing the semantics of concurrent data structures through quasi-linearizability [2]. Just like quasi-linearizability, we provide quantitative relaxations of concurrent data structures. Unlike quasi-linearizability which uses syntactic distances, our relaxations are based on semantical distances from a sequence to the sequential specification. We briefly present the quasi-linearizability approach, identify two main issues, and how our method overcomes these.

We call two sequences $\mathbf{x}$, $\mathbf{x}'$, both of length n, permutation equivalent, written $\mathbf{x} \sim \mathbf{x}'$, if there exists a permutation p on $\{1,\ldots,n\}$ such that for all $1 \leqslant i \leqslant n$, $\mathbf{x}(i) = \mathbf{x}'(p(i))$. We write $\mathbf{x} \sim_p \mathbf{x}'$ to emphasize the permutation witnessing $\mathbf{x} \sim \mathbf{x}'$. In such a case, the permutation distance between $\mathbf{x}$ and $\mathbf{x}'$ is given as $\max\{|i - p(i)| \mid 1 \leqslant i \leqslant n\}$.

Let S be a sequential specification over Σ. In [2], the distance of a sequence $\mathbf{x} \in \Sigma^*$ to S is defined via a collection D of subsets of Σ. Let $\mathbf{y} \in \Sigma^*$ be a sequence such that $\mathbf{z} = \mathbf{xy}$ has a permutation equivalent $\mathbf{z}' \in S$. Then, for $A \in D$, the A-cost of obtaining $\mathbf{z}'$ from $\mathbf{z}$ is the permutation distance between $\mathbf{z}|A$ and $\mathbf{z}'|A$, where $|$ denotes restriction. Let k_A denote the minimal A-cost over all $\mathbf{y}$. Then, $\mathbf{x}$ is quasi-linearizable with quasi-linearization factor $Q\colon D \to \mathbb{N}$, if for all $A \in D$, $k_A \leqslant Q(A)$. Observe that the distance for $\mathbf{x}$ is obtained by quantifying over all possible extensions of $\mathbf{x}$ whose permutations are in S. We now show that this definition fails to capture desired relaxation distances.

1. *Not precise.* Consider the following sequence:

$$\mathtt{enq(1)enq(2)enq(3)deq(1)deq(2)enq(4)deq(4)}$$

In order to assign a relaxation cost of 1 to this sequence belonging to an out-of-order queue, quasi-linearizability employs a scheme where only enqueue operations are allowed to commute. Formally, quasi-linearizability uses $D = \{Enq, Deq\}$ with $Q(Enq) = k$ and $Q(Deq) = 0$, where Enq (resp., Deq) contains all `enq` (resp., `deq`) symbols. However, with this scheme the sequence $\mathtt{deq(i)}^n$ which removes elements from an empty queue will always be in any k-relaxation of the queue because setting

$$\mathbf{z} = \mathtt{deq(i)}^n\mathtt{enq(i)}^n, \quad \mathtt{z}' = \mathtt{enq(i)}^n\mathtt{deq(i)}^n$$

will give $k_{Enq} = k_{Deq} = 0$, independent of the value n. This means that the following implementation is a 0-relaxation of queue.

```
enq(x): { while(true);}  deq(): { return random();}
```

This implementation is clearly not implementing a queue, nor any intended bounded relaxation of a queue, but all the sequences it generates will have zero distance relative to D as given above. Thus, quasi-linearizability cannot exactly capture intended relaxations and might allow wrong behaviors. Observe that we have already shown in Proposition 5.2 that out-of-order k-queues can never generate such erroneous behavior.

2. *Not general.* For a stack, consider the sequence

$$\begin{aligned} \mathbf{x} \quad = \quad & \mathtt{push(a)}[\mathtt{push(i)pop(i)}]^n \\ & \mathtt{push(b)}[\mathtt{push(j)pop(j)}]^m \\ & \mathtt{pop(a)}[\mathtt{push(1)pop(1)}]^r \end{aligned}$$

where all symbols, $\mathtt{a}, \mathtt{b}, \ldots$ have distinct values. Prior to the removal of `a`, the stack contains `a` and `b`, with the latter at the top position. The distance in the out-of-order relaxation induced by maximal path cost and segment cost in this case is 1 since the element popped is immediately after the top entry. However, with quasi-linearization factor Q, it is impossible to precisely capture out-of-order penalty for data structures like stacks. First, consider the case where we pick $\mathbf{z} = \mathbf{x}$, which we can do since $\mathbf{x}$ has a permutation equivalent valid stack sequence. In order to get a permutation $\mathbf{x}'$ of $\mathbf{x}$ such that $\mathbf{x}'$ is a valid sequence of a stack, either one of $\mathtt{pop(a)}$ or $\mathtt{push(b)}$ has to move over m copies of `push` and `pop` operations, or one of $\mathtt{push(a)}$ or $\mathtt{push(b)}$ has to move over n copies of `push` and `pop` operations. So either D is empty which allows for any sequence to be in the relaxation or it is always possible to pick the values for n and m such that the penalty is arbitrarily large. Second, consider

the case where we extend **x** with **y** such that **xy** has a permutation equivalent valid stack behavior. But because of the $2r$-long suffix of **x**, a similar reasoning as in the previous case applies to this case as well.

Similarly, a stuttering relaxation will not have a finite quasi-linearizability distance, since no permutation of (an extension of) a stuttering sequence is in the original sequential specification.

Consistency conditions. As opposed to relaxing the sequential specification of a concurrent data structure, one may also relax the consistency condition, e.g., quiescent consistency [4] instead of linearizability. We note that linearizable out-of-order relaxation of a stack is incomparable to a quiescently consistent stack. To see this, first, consider a concurrent history **c** with two threads t_1 and t_2. The history **c** starts with the invocation of `push(a)` by t_1, followed by a sequence $\texttt{pop(i)}^n\texttt{push(i)}^n$ all executed by t_2. This history is quiescently consistent for stack because the reordering of methods (even those that do not overlap in time) is allowed as long as they are not separated by a quiescent state. On the other hand, any linearization of **c** will have to observe out-of-order pop operations since the operations of t_2 do not overlap. So, for each k, there exists a history which is quiescently consistent for stack but is not in the specification of an out-of-order k-stack. Second, consider the sequential history `push(a)push(b)pop(a)` which has an out-of-order relaxation distance of 1. Since the history is sequential, quiescent consistency will not allow any reordering. Thus, for any k, there exists a history which is in the specification of an out-of-order k-stack but not quiescently consistent. A comprehensive overview of variants of weaker and stronger consistency conditions than linearizability can be found in [6].

Implementations. Work related to our implementations and experiments is discussed in more detail in Section 8 and Section 9. Here we briefly refer to all the work considered. Our relaxed stack implementation is closely related to the very recent efficient lock-free implementation of a relaxed k-FIFO queue [11] (by some of us and a third coauthor), but the change from queue to stack semantics imposes a significant difference as well. The k-FIFO queue [11] is in turn related to implementations of relaxed FIFO queues such as the Random Dequeue and Segment Queue [2] as well as Scal queues [10, 12]. Both the Random Dequeue Queue and the Segment Queue implement the restricted out-of-order relaxation. The Segment Queue [2] and the k-FIFO queue [11] implement a queue of segments. However, the implementations are quite different with significant impact on performance, see Section 9. Our relaxed stack implementation implements a stack of segments. Scal queues are relaxed queues with, in general, unbounded relaxation. Since any relaxed stack or queue implementation also implements a pool, we compare our work also to state-of-the-art pool implementations [1, 3, 19].

In [5], the authors show that implementing deterministic data structure semantics requires expensive synchronization mechanisms which may prohibit scalability in high contention scenarios. We agree with that and show in our implementations and experiments that the non-determinism introduced in the sequential specification provides scalability and performance benefits. In [15], the authors present a work-stealing queue with relaxed semantics where queue elements may be returned any number of times instead of just once. In comparison to other state-of-the-art work-stealing queues with non-relaxed semantics this may provide better performance and scalability. Again the introduced non-determinism pays off. Overview of different relaxations on hardware and software level is presented in [9, 18].

8. Implementations of relaxed data structures

In this section we present the new implementation of a restricted out-of-order stack (k-stack, for short) and present the two new implementations of a stuttering shared counter. It is interesting to note that the "restricted" relaxation seems to be crucial for obtaining performance[2], which is why we focus on it.

8.1 k-Stack

The top pointer of a concurrent stack may become a scalability bottleneck under high contention [20]. The main idea behind our k-stack implementation is to reduce contention on the top pointer by maintaining a stack of so-called k-segments[3]. We implemented the stack that holds the k-segments similarly to the lock-free stack of [20] with the difference that there is always at least one k-segment, even if it is empty, on the stack. This avoids unnecessary removal and adding of a k-segment, e.g. in the empty state. A k-segment (or just segment, when no confusion arises) contains k "atomic values" (see next paragraph) which may either point to `null` indicating an empty slot or may hold a so-called item. Both push and pop operations are served by the top segment. Hence, up to k stack operations may be performed in parallel. A push operation tries to insert an element in the top segment. It adds a new segment to the stack if the top segment is full. A pop operation tries to remove an element from the top segment. It removes the top segment from the stack if it is empty and is not the only segment on the stack. Additionally, each segment contains an atomic counter `remove` that counts how many threads are trying to remove it from the stack. The counter is initially set to zero.

The pseudo code of the lock-free k-stack algorithm with $k > 0$ is depicted in Figure 4. The occurrence of the ABA problem is made unlikely through version numbers. We refer to values enhanced with version numbers as atomic values. Hence an atomic value has two fields, the actual value `val` and its version number `ver`.

The methods `init`, `try_add_new_ksegment`, and `try_remove_ksegment` implement the stack of segments. In the latter, the atomic counter `remove` is updated and the method `empty` that performs an empty check is called. We discuss the method `empty` within the `pop` method, as it is also called there.

Let `item` represent an element to be pushed on the k-stack. The `push` method first tries to find an empty slot for the `item` using the `find_empty_slot` method (line 45). The `find_empty_slot` method randomly selects an index in the top k-segment and then linearly searches for an empty slot starting with the selected index and wrapping around at index k. Then the `push` method checks if the k-stack state has been consistently observed by testing whether `top` changed in the meantime (line 46) which would trigger a retry. If an empty slot is found (line 47) the method tries to insert the `item` at the location of the empty slot using a compare-and-swap (`CAS`) operation (line 49). If the insertion is successful the method verifies whether the insertion is also valid by calling the `committed` method (line 50), as discussed below. If any of these steps fails, a retry is performed. If no empty slot is found in the current top segment, the `push` method tries to add a new segment to the stack of segments (line 53) and then retries.

The `committed` method (line 24) validates an insertion, it ensures that the inserted element is really inserted *on the stack*. This method is the core and the main novelty of the algorithm. It returns `true` when the insertion is valid and `false` when it is not valid. An insertion is invalidated if a concurrent thread removes the segment to which the element was inserted before the effect of the

[2] For an efficient implementation one needs a sequential specification that fits the properties of the hardware that it will run on.

[3] The same high-level idea is used in the Segment Queue [2] and the k-FIFO queue [11] discussed in the next subsection.

```
1 global top;
2
3 void init():
4   new_ksegment = calloc(sizeof(ksegment));
5   top = atomic_value(new_ksegment, 0);
6
7 void try_add_new_ksegment(top_old):
8   if top_old == top:
9     new_ksegment = calloc(sizeof(ksegment));
10    new_ksegment->next = top_old;
11    top_new = atomic_value(new_ksegment, top_old.ver+1);
12    CAS(&top, top_old, top_new);
13
14 void try_remove_ksegment(top_old):
15  if top_old == top:
16    if top_old->next != null:
17      atomic_increment(&top_old->remove);
18      if empty(top_old):
19        top_new = atomic_value(top_old->next,
              top_old.ver+1);
20        if CAS(&top, top_old, top_new):
21          return;
22      atomic_decrement(&top_old->remove);
23
24 bool committed(top_old, item_new, index):
25  if top_old->s[index] != item_new:
26    return true;
27  else if top_old->remove == 0:
28    return true;
29  else:  //top_old->remove >= 1
30    item_empty = atomic_value(EMPTY, item_new.ver+1);
31    if top_old != top:
32      if !CAS(&top_old->s[index], item_new, item_empty):
33        return true;
34    else:
35      top_new = atomic_value(top_old.val, top_old.ver+1);
36      if CAS(&top, top_old, top_new):
37        return true;
38      if !CAS(&top_old->s[index], item_new, item_empty):
39        return true;
40  return false;
41
42 void push(item):
43  while true:
44    top_old = top;
45    item_old, index = find_empty_slot(top_old);
46    if top_old == top:
47      if item_old.val == EMPTY:
48        item_new = atomic_value(item, item_old.ver+1);
49        if CAS(&top_old->s[index], item_old, item_new):
50          if committed(top_old, item_new, index):
51            return true;
52      else:
53        try_add_new_ksegment(top_old);
54
55 item pop():
56  while true:
57    top_old = top;
58    item_old, index = find_item(top_old);
59    if top_old == top:
60      if item_old.val != EMPTY:
61        item_empty = atomic_value(EMPTY, item_old.ver+1);
62        if CAS(&top_old->s[index], item_old, item_empty):
63          return item_old.val;
64      else:
65        if only_ksegment(top_old):
66          if empty(top_old):
67            if top_old == top:
68              return null;
69        else:
70          try_remove_ksegment(top_old);
```

Figure 4. Lock-free *k*-stack algorithm

insertion took place. Therefore, an insertion is valid if the inserted item already got popped at validation time by a concurrent thread (line 25, 32, 38) or the segment where the item was inserted was not removed by a concurrent thread (line 27). A `remove` counter larger than zero indicates that the segment has been removed or concurrent threads are trying to remove the segment from the stack (line 29). If the current top segment is not equal to the segment where the item was inserted we have to conservatively assume that the segment was removed from the stack (line 31) and undo the insertion (line 32). If the current top segment is equal to the segment where the item was inserted, a race with concurrent popping threads may occur which may not have observed the insertion of the item and may try to remove the *k*-segment from the stack in the meantime. This would result in loss of the inserted item. To prevent that, the method tries to increment the version number in the `top` atomic value using `CAS` (line 36) forcing threads that concurrently try to remove that *k*-segment to retry. If this fails, a concurrent pop operation may have changed `top` (line 20) which would make the insertion potentially invalid. Hence, in case of losing the race, the method tries to undo the insertion using `CAS` (line 38).

The `pop` method returns an item if the *k*-stack is not empty. Otherwise it returns `null`. Similar to the `push` method, the `pop` method first tries to find an item in the top segment using the `find_item` method (line 58). The `find_item` method randomly selects an index in the top *k*-segment and then linearly searches for an item starting with the selected index and wrapping around at index *k*. Then, the `pop` method checks if the *k*-stack state has been consistently observed by checking whether `top` changed in the meantime (line 59) which would trigger a retry. If an item was found (line 60) the method tries to remove it using `CAS` (line 62) and returns it if the removal was successful (line 63). Otherwise a retry is performed. If no item is found and the current segment is the only segment on the stack (line 65) an empty check is performed using the method `empty` (line 66). This method stores the values of the *k* slots of the segment in a local array (if they are empty) and subsequently checks in another pass over the segment slots whether the values in the slots changed in the meantime. If a non-empty slot was found, the empty method immediately returns `false`. If the empty check succeeded and the top did not change in the meantime (line 67), `null` is returned (line 68). Otherwise, if no item is found in the current segment and there is more than one segment in the stack, the method tries to remove the segment (line 70) and performs a retry.

Correctness: *k*-Stack

We now prove that the *k*-stack implementation is correct for the relaxed stack semantics.

PROPOSITION 8.1. *The k-stack algorithm is linearizable with respect to restricted out-of-order k-stack.*

Proof. Without loss of generality, we assume that each item pushed on the stack is unique. A segment `s'` is *reachable* from a segment `s` if either `s'=s` or `s'` is reachable from `s->next`. An item `i` is *on the stack*, if `push(i)` has already committed and there exists a segment reachable from the top segment containing a slot whose value is `i`. Note that reachability is important, i.e., only having a slot containing the item is not enough to guarantee that the item is logically on the stack, because the slot could be in a segment (to be) removed by a concurrent pop operation.

We begin by identifying a linearization point of each method call. The goal is to show that the sequential history obtained from a concurrent history by ordering methods according to their linearization points is in the specification of a restricted out-of-order *k*-stack. The linearization point of `push` is the reading of the empty slot (line 45) in the last iteration (successful insertion) of the main loop. The linearization point of `pop` that does not return `null` is the reading of a non-empty slot (line 58) in the last iteration (successful removal) of the main loop. The linearization point of a `null`-returning `pop` is the point after the first pass of the segment in the call to `empty` method (line 66) which returns `true`.

The correctness argument is based on the following facts.

1. An item is pushed on the stack exactly once. This is a consequence of our unique-items assumption and the control flow of `push(i)`, the only method that can modify a slot to contain `i`.

2. An item is popped at most once. If an item `i` is on the stack, then it can only be removed once, because of 1. and the existence of a unique statement which replaces `i` with empty. If `i` is in some slot but not on the stack, then `push(i)` will erase `i` and retry insertion before committing. We have to show that while `i` is in some slot but not on the stack, no pop operation can return `i`. Clearly, the call to method `committed` by `push(i)` must return `false`. This implies that until `committed` completes, the slot where `i` resides is not modified by any other thread. Otherwise, either after the first `if` statement (line 25) or following failed `CAS` attempts (lines 32 and 38) of replacing `i` with empty will lead to returning true. Furthermore, when control reaches the only exit point for returning `false`, it is guaranteed that there is no slot containing `i`. Thus, if `i` is not on the stack, no pop operation could have replaced it with empty.

3. If a pop operation returns `null`*, then during its execution, there must be a state at which there are no items on the stack.* Since returning `null` is without any side-effect, it suffices to prove the existence of a state which corresponds to a logically empty stack. The call to `empty` is only done when the top segment is the only segment in the stack. In the `empty` method, the value of `top` is checked at the beginning and after the first pass to ensure that the pointer is not updated by concurrent operations. Hence, the stack is indeed logically empty at the linearization point since the second pass succeeds.

4. An item `j` *cannot be popped before an item* `i`*, if they are both on the stack, and* `i`*,* `j` *are in segments* `s`*,* `s'`*, respectively, with* `s'` *reachable from* `s` *and* `s'` $\neq$ `s`. The segment `s'` can become a top segment only after the segment `s` has been removed by some pop operation. Moreover, if a segment becomes unreachable from the top segment, it remains unreachable. These two observations imply that the linearization point of `pop(i)` must precede the linearization point of `pop(j)` which can only happen when `s'` is a top segment.

5. An item `i` *on the stack is popped only if it is one of the* $k-l$ *youngest items on the stack, where* l *is the current lateness of the youngest item.* By youngest we mean most recently pushed. Recall that lateness is the number of pops that were performed after the pop of the previous youngest item or the push of the current youngest one. Assume `i` is popped from the stack at the current moment in time and at that point the youngest item is `t` with current lateness l. This means that ever since `t` is the youngest item on the stack, no push operation was performed and there have been l pop operations performed none of which removed `t`. Let `j` be any of these l popped items. Since `t` is the last item pushed and it is still on the stack, `t` is in the top segment. Since `j` is removed before `t`, by 4. it must have also resided in the top segment. For the same reason, also `i` is in the top segment prior to its removal. Hence, at the moment in which `pop(i)` happens, there are at most $k-l$ items in the top segment.

Now, 5. shows that the sequential behavior obtained by ordering methods according to their linearization points satisfies the restricted-out-of-order k-stack. Moreover, 3. shows an even stricter behavior, a linearizable empty check.[4] Thus, any concurrent execution generated by the given algorithm is linearizable for restricted out-of-order k-stack. Observe that already 1. and 2. show that the k-stack has pool semantics. □

[4] We could easily relax the linearizable empty check to fit the restricted out-of-order specification, by removing line 66 in the code. However, a linearizable empty check is a valuable feature of a concurrent implementation.

```
1 struct blk_original {
2   pidtype X;
3   bool Y;
4   val_t V;
5   bool C;
6   val_t D;
7 };
```

(a) CAS.

```
1 struct blk_modified {
2   pidtype X[k];
3   bool Y[k];
4   val_t V;
5   bool C[k];
6   val_t D;
7 };
```

(b) k-CAS.

Figure 5. Wait-free CAS and k-CAS state structures.

Without any particular difficulty, but with a somewhat lengthy argument, one can show that the k-stack algorithm is lock-free by showing that whenever a thread retries an operation, another thread completes its operation ensuring progress of at least one thread.

8.2 k-Stuttering shared counters

We implemented the two versions of a stuttering k-shared counter, as discussed in the introduction. The first version is based on a k-relaxed stuttering version of a wait-free software CAS operation [13] (k-CAS for short). It uses a structure `blk_original`, shown in Figure 5(a), to keep track of the state of concurrent CAS operations. The atomic value is located in field `V` and the CAS operation uses the decision fields `X`, `Y`, and `C` to determine which thread gets permission to change `V`. We modified the `blk_original` structure so that the fields `X`, `Y`, and `C` are arrays of size k depicted in structure `blk_modified` in Figure 5(b). We keep the main CAS operation unmodified but use a balancing function that maps threads to array indices i smaller than k (the thread ID modulo k). A thread determines the state of its CAS operation by just accessing position i in the arrays `X`, `Y`, and `C`. On success, a thread writes the new value into `V`. Hence, up to k concurrent threads may perform the k-CAS operation in parallel and change `V` resulting in a loss of at most $k-1$ state changes, which further results in at most $k-1$ lost shared counter updates.

The second version of the k-shared counter is the k-distributed counter depicted in Figure 1(b).

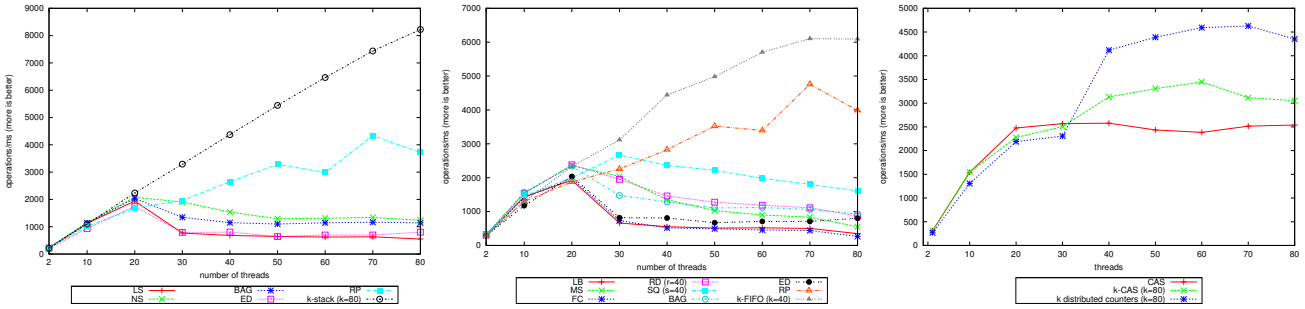
It is not difficult to show that both implementations are linearizable with respect to the k-stuttering shared counter and they are lock-free.

9. Experiments

We evaluate the performance and scalability of the k-stack, several existing quantitatively relaxed FIFO queues, and the k-stuttering shared counter implementations. All experiments ran on an Intel-based server machine with four 10-core 2.0GHz Intel Xeon processors (40 cores, 2 hyperthreads per core), 24MB shared L3-cache, and 128GB of memory running Linux 2.6.39. We implemented a benchmarking framework to analyze our k-stack and k-shared counter implementations, as well as the implementations of relaxed queues. The benchmarking framework can be configured for a different number of threads (n), number of operations each thread performs (o), and the computational load performed between each operation (c). The computational load between two consecutive operations is created by iteratively calculating π and c is the number of iterations performed. We use $c = 2000$ which takes a total of $4600ns$ on average in our experiments. The framework uses static preallocation for memory used at runtime with touching each page before running the benchmark to avoid paging issues.

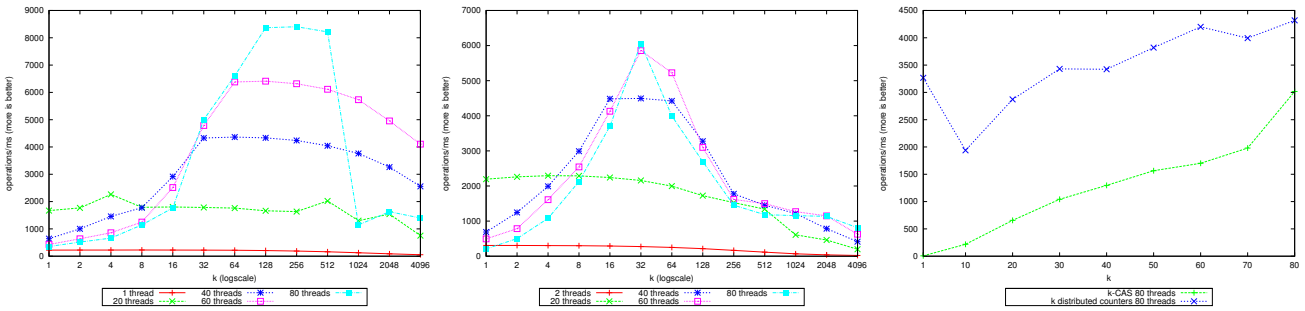
9.1 k-Stack

We compare our k-stack implementation with a standard lock-based stack (LS), which acquires a global lock for each stack operation, and a non-blocking stack (NS) [20], which uses a CAS operation to manipulate the top pointer of the stack. Moreover, we compare our k-stack with different pools. The lock-free (BAG) [19]



(a) Stack (b) FIFO queue (c) Shared counter

Figure 6. Benchmarks on a 40-core (2 hyperthreads per core) server with an increasing number of threads



(a) Stack (b) FIFO queue (c) Shared counter

Figure 7. Benchmarks on a 40-core (2 hyperthreads per core) server with increasing k

pool is based on thread-local lists of elements. Threads put elements on their local list and take elements from their local list if it is not empty. If it is empty they take elements from the lists of other threads. The lock-free elimination-diffraction pool (ED) [1] uses a set of FIFO queues to store elements. Access to these queues is balanced using elimination arrays and a diffraction tree. The synchronous rendezvousing pool (RP) [3] implements a single elimination array using a ring buffer. A get operation marks a slot identified by its thread id and waits for a take operation to insert an element. Take operations scan the array for pending get operations.

Figure 6(a) depicts the performance analysis of our k-stack. We configure $k = 80$, which is a good k (on average) for a broad range of thread combinations and workloads on our server machine. This is no surprise since the server machine has (logically) 80 cores. The analysis is done on a producer-consumer workload where half of the threads are producers and half are consumers. The k-stack outscales and outperforms all considered stack and pool implementations. Figure 7(a) shows the effect of k on performance and scalability. There exists an optimal k with respect to performance, which is also robust in the sense that there exists only a single range of close-to-optimal k. For large k performance decreases due to higher sequential overhead, e.g. scanning for elements in almost empty k-segments. Note that an increase in performance above $k = 80$ is not to be expected on the given architecture.

9.2 Quantitatively relaxed FIFO queues

We also evaluate the existing implementations of a k-FIFO queue [11] and different FIFO queues, quasi-linearizable FIFO queues, and the pools introduced in the previous section. The k-FIFO queue [11] implements a restricted out-of-order k-queue as a lock-free linked list of k-segments. An enqueue operation is served by the tail k-segment and a dequeue operation is served by the head k-segment. Hence, up to k enqueue and k dequeue operations may be performed in parallel. The k-FIFO queue is empty if head and tail point to the same k-segment which does not contain any elements. The lock-based (strict) FIFO queue (LB) locks a global lock for each queue operation. The lock-free Michael-Scott (strict) FIFO queue (MS) [14] uses `CAS` operations to change head, tail, and next pointer in a linked list of elements. The flat-combining (strict) FIFO queue (FC) [8] is based on the approach that a single thread performs the queue operations of multiple threads by locking the whole queue, collecting pending queue operations, and applying them to the queue. The Random Dequeue Queue (RD) [2] implements a quasi-linearizable FIFO with quasi-factor r where r defines the range $[0, r-1]$ of a random number. It actually implements a restricted out-of-order r-FIFO queue. RD is based on MS where the dequeue operation was modified in a way that the random number determines which element is returned starting from the oldest element. The Segment Queue (SQ) [2] is a quasi-linearizable FIFO queue with quasi-factor s. It is logically similar (both implement a queue of segments and hence a restricted out-of-order queue) to the k-FIFO queue but does not provide a linearizable empty check, i.e., it may return `null` in the not-empty state. Also, SQ comes with a different segment management strategy than the k-FIFO queue, which results on average in significantly more CAS operations.

Figure 6(b) depicts the performance analysis of the queues. We configure $k = r = s = 40$, which turns out be a good k (on average) for a broad range of thread combinations and workloads on our server machine. This is no surprise since then the level of possible parallelism is $2k = 80$, the number of (logical) cores. We use a producer-consumer workload where half of the threads are producers and half consumers. The k-FIFO queue outscales and outperforms all considered FIFO queue, quasi-linearizable FIFO queue, and pool implementations. Figure 7(b) shows the effect of k on performance and scalability. Again, there exists a robust

and optimal k with respect to performance. Also here, for large k performance decreases due to larger sequential overhead.

9.3 k-Shared counter

We compare our k-CAS-based shared counter and distributed shared counter implementations with a shared counter implementation based on a regular CAS operation depicted in Figure 6(c). The threads perform in total one million counter increment operations in each benchmark run. The CAS version performs best until 30 threads. After that the k-CAS-based shared counter and the distributed shared counter version outperform it. Figure 7(c) shows the effect of k on performance and scalability. In the k-CAS version performance monotonically increases with larger k, whereas in the distributed shared counter version performances decreases until $k = 10$, but monotonically increases after that. Our educated guess is that this is caused by the trade-off between two possible sources of contention: (1) CAS on the same memory location, and (2) bad caching, i.e., accessing many different locations in memory. The distributed shared counter decreases (1) but increases (2). However, except for small values of k, we observe that the gain is larger than the loss.

10. Final remarks

We have presented a framework for quantitative relaxation of concurrent data structures together with generic as well as further concrete instances of it. Our main motivation is the belief that relaxed data structures may decrease contention and thus provide the potential for scalable and well-performing implementations. Indeed, the potential advantage which we demonstrate utilizable is striking. The lessons learned can be summarized as follows: The way from a sequential implementation to efficient concurrent implementation is always hard. Just because a sequential specification is relaxed, it does not necessarily mean that an efficient implementation immediately follows. However, efficient implementations that benefit from quantitative relaxations are possible, as we demonstrate in this paper. In our opinion, the framework provides a firm formal ground for quantitative relaxation of concurrent data structures and paves the road to designing efficient concurrent implementations.

Our current results open up several directions for future work. One important issue is the applicability of relaxed data structures. Demonstrating applicability can either be achieved by exploring applications that tolerate a relaxation, e.g. provide less accurate but nevertheless acceptable results, or showing that end-to-end quality may remain the same despite the actual relaxation of semantics. In the latter case, relaxations do not influence correctness in the sense of [16, 17]. Another evident but difficult goal would be to synthesize well-performing implementations from relaxations. As a first step we believe it is important to study the main principles that lead to good performance. This is another line of future work that we plan to undertake in small steps.

Acknowledgements

This work has been supported by the European Research Council advanced grant QUAREM, the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23), and an Elise Richter Fellowship (Austrian Science Fund V00125). We thank the anonymous referees for their constructive and inspiring comments and suggestions. Ana Sokolova wishes to thank Dexter Kozen and in particular Joel Ouaknine: had they not saved her life, she would have missed a lot of the fun involved in working on this paper and seeing it finished.

References

[1] Y. Afek, G. Korland, M. Natanzon, and N. Shavit. Scalable producer-consumer pools based on elimination-diffraction trees. In *Proc. European Conference on Parallel Processing (Euro-Par)*, pages 151–162. Springer, 2010.

[2] Y. Afek, G. Korland, and E. Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In *Proc. Conference on Principles of Distributed Systems (OPODIS)*, pages 395–410. Springer, 2010.

[3] Y. Afek, M. Hakimi, and A. Morrison. Fast and scalable rendezvousing. In *Proc. International Conference on Distributed Computing (DISC)*, pages 16–31, Berlin, Heidelberg, 2011. Springer-Verlag.

[4] J. Aspnes, M. Herlihy, and N. Shavit. Counting networks. *Journal of the ACM*, 41:1020–1048, 1994.

[5] H. Attiya, R. Guerraoui, D. Hendler, P. Kuznetsov, M. Michael, and M. Vechev. Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In *Proc. of Principles of Programming Languages (POPL)*, pages 487–498. ACM, 2011.

[6] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., 2008.

[7] M. Herlihy and J. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

[8] D. H. I. Incze, N. Shavit, and M. Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 355–364. ACM, 2010.

[9] C. Kirsch and H. Payer. Incorrect systems: It's not the problem, it's the solution. In *Proc. Design Automation Conference (DAC)*. ACM, 2012.

[10] C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Brief announcement: Scalability versus semantics of concurrent FIFO queues. In *Proc. Symposium on Principles of Distributed Computing (PODC)*. ACM, 2011.

[11] C. Kirsch, M. Lippautz, and H. Payer. Fast and scalable k-fifo queues. Technical Report 2012-04, Department of Computer Sciences, University of Salzburg, June 2012.

[12] C. Kirsch, H. Payer, H. Röck, and A. Sokolova. Performance, scalability, and semantics of concurrent FIFO queues. In *Proc. International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, pages 273–287. LNCS 7439, 2012.

[13] V. Luchangco, M. M., and N. Shavit. On the uncontended complexity of consensus. In *Proc. International Symposium on Distributed Computing (DISC)*, pages 45–59. Springer-Verlag, 2003.

[14] M. Michael and M. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proc. Symposium on Principles of Distributed Computing (PODC)*, pages 267–275. ACM, 1996.

[15] M. Michael, M. Vechev, and V. Saraswat. Idempotent work stealing. In *Proc. Principles and Practice of Parallel Programming (PPoPP)*, pages 45–54. ACM, 2009.

[16] S. Misailovic, S. Sidiroglou, H. Hoffmann, and M. C. Rinard. Quality of service profiling. In *Proc. 32nd ACM/IEEE International Conference on Software Engineering (ICSE) - Volume 1*, pages 25–34. ACM, 2010.

[17] A. Sampson, W. Dietl, E. Fortuna, D. Gnanapragasam, L. Ceze, and D. Grossman. Enerj: approximate data types for safe and general low-power computation. In *Proc. 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 164–174. ACM, 2011.

[18] N. Shavit. Data structures in the multicore age. *Communications ACM*, 54:76–84, March 2011.

[19] H. Sundell, A. Gidenstam, M. Papatriantafilou, and P. Tsigas. A lock-free algorithm for concurrent bags. In *Proc. Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 335–344, New York, NY, USA, 2011. ACM.

[20] R. Treiber. Systems programming: Coping with parallelism. Technical Report RJ5118, IBM Almaden Research Center, April 1986.

Int J Softw Tools Technol Transfer (2013) 15:189–209
DOI 10.1007/s10009-012-0246-x

EUROSYS

Temporal isolation in real-time systems: the VBS approach

Silviu S. Craciunas · Christoph M. Kirsch · Hannes Payer · Harald Röck · Ana Sokolova

Published online: 1 July 2012

Abstract Temporal isolation in real-time systems allows the execution of software processes isolated from one another in the temporal domain. Intuitively, the execution of a process is temporally isolated if the real-time behavior of the process is independent of the execution of the other concurrently scheduled processes in the system. The article provides a comprehensive discussion of temporal isolation through variable-bandwidth servers (VBSs). VBS consists of an EDF-based uniprocessor scheduling algorithm and a utilization-based schedulability test. The scheduling algorithm runs in constant time modulo the time complexity of queue management. The schedulability test runs in time linear in the number of processes and enables admission of an individual process in constant time. The test is a sufficient condition for VBS to provide temporal isolation through lower and upper response-time bounds on processes. We present the VBS design, implementation, proofs, and experiments, followed by condensed versions of results on scheduler overhead accounting with VBS and on reducing power consumption in VBS systems.

Keywords Real-time · Scheduling · Resource reservation · Power-aware scheduling · Compositionality

This work has been supported by the EU ArtistDesign Network of Excellence on Embedded Systems Design, the National Research Network RiSE on Rigorous Systems Engineering (Austrian Science Fund S11404-N23), and an Elise Richter Fellowship (Austrian Science Fund V00125). We thank the anonymous reviewers for their comments and suggestions.

S. S. Craciunas (✉) · C. M. Kirsch · H. Payer · H. Röck · A. Sokolova
University of Salzburg, Salzburg, Austria
e-mail: scraciunas@gmail.com

1 Introduction

Temporal isolation can be seen as a quantified version of scheduler fairness that is particularly relevant to real-time systems. Temporal isolation enables real-time programming models that are compositional with respect to real-time behavior. It is thus an important prerequisite of robust and scalable engineering methodologies for real-time systems.

In this article, we extend and complete the discussion of temporal isolation through variable-bandwidth servers (VBSs) from [1]. The process model of VBS is based on the concept of actions as pieces of sequential process code. A VBS process is modeled as a potentially infinite sequence of actions, which allows us to define response times, and thus temporal isolation of processes, at the level of individual actions. The response time of an action is defined as the duration from the time instant when process execution reaches the beginning of the action (arrival) until the time instant when process execution reaches the beginning of the next action (termination). The VBS scheduler is based on the well-known EDF mechanism [2] and executes processes in temporal isolation. More precisely, with VBS the response time as well as the variance of the response time (jitter) of a given action is bounded independently of any other actions that run concurrently and is thus solely determined by the given action itself.

VBS can be seen as a more general form of constant-bandwidth servers (CBSs) [3]. CBS maintains a fixed rate of process execution, i.e., a CBS process is allowed to execute a constant amount of time (limit) in a time period of constant length (period), whereas a VBS process can change both the limit and the period at runtime. The only restriction on a VBS process is that its utilization, i.e., the ratio of its limit over its period, has to be less than or equal to a given utilization cap. This restriction enables a fast, sufficient schedulability test

for VBS, which guarantees temporal isolation as long as the sum of the utilization caps of all processes is less than or equal to 100 %. Dynamically modifying the limit and period (rate) at which VBS processes execute enables different portions of process code to have different throughput (limit) and latency (period) requirements, which may be helpful in applications such as control loops [4].

In this article, we provide a detailed theoretical and practical view on VBS. After reiterating the basic concepts of VBS presented in [1], we extend the discussion twofold with material that has appeared in a technical report version [5] of [1]. In particular, we give detailed proofs of the schedulability test and include experiments analyzing the actual quality of temporal isolation which were not present in [1]. Next, we include implementation details that enable an efficient implementation of VBS in a real system. We present details of four alternative queue management plugins based on lists, arrays, matrices, and trees that trade off time and space complexity of the VBS scheduler. Finally, we present condensed versions of our previous results on scheduler overhead accounting with VBS [6] and on reducing power consumption while maintaining temporal isolation [7].

The structure of the rest of the article is as follows. We start by describing VBS conceptually in Sect. 2 and present the scheduling algorithm in Sect. 3, as introduced in [1], extended by the proofs of the results. In Sect. 4, we present implementation details of the scheduling algorithm including time and space complexity when using the four different queue management plugins. An extended version of the experimental results presented in [1] is discussed in Sect. 5. In Sects. 6 and 7, we round up the description by including the results on scheduler overhead accounting and power-aware scheduling with VBS. We present a detailed account of related work in Sect. 8. Section 9 gathers the conclusions and presents future work.

2 Variable-bandwidth servers

The timeline is represented by the set of natural numbers $\mathbb{N}$, i.e., we consider a discrete timeline. The main ingredients of the scheduling model are VBSs defined through virtual periodic resources and VBS-processes composed of sequential actions.

2.1 VBS and processes

A virtual periodic resource [8,9] (capacity) is a pair $R = (\lambda, \pi)$ where $\lambda \in \mathbb{N}^+$ stands for limit and $\pi \in \mathbb{N}^+$ for period. If no confusion arises, we will say resource for virtual periodic resource. The limit λ specifies the maximum amount of time the resource R can be used (by a server and thus process) within the period π. We assume that in a resource $R = (\lambda, \pi)$, $\lambda \leq \pi$. The ratio $u = \frac{\lambda}{\pi}$ is the utilization of the resource $R = (\lambda, \pi)$. We allow for an arbitrary set of resources denoted by $\mathcal{R}$.

A constant-bandwidth server (CBS) [3] is uniquely determined by a virtual periodic resource $R = (\lambda, \pi)$. A CBS serves CBS-processes at the virtual periodic resource R, i.e., it lets a process execute for λ amount of time, within each period of length π. Hence, the process as a whole receives the constant bandwidth of the server, prescribed by the defining resource.

A variable-bandwidth server (VBS) is uniquely determined by the utilization ratio u of some virtual periodic resource. The utilization ratio prescribes an upper bound bandwidth cap. The server may execute processes that change the resources in time, as long as the resources have utilization less than or equal to the defining utilization. The notion of a process that can be served by a given VBS is, therefore, richer in structure. Note that a VBS can serve processes with any kind of activation. The server itself is periodic (with variable periodicity), but the processes need not be.

A VBS-process $P(u)$ served by a VBS with utilization u, is a finite or infinite sequence of (process) actions, $P(u) = \alpha_0\alpha_1\alpha_2\ldots$ for $\alpha_i \in \text{Act}$, where $\text{Act} = \mathbb{N} \times \mathcal{R}$. An action $\alpha \in \text{Act}$ is a pair $\alpha = (l, R)$ where l standing for load is a natural number, which denotes the exact amount of time the process will perform the action on the resource R, and $R = (\lambda, \pi)$ has utilization less than or equal to the utilization of the VBS, that is $\frac{\lambda}{\pi} \leq u$. If no confusion arises, we call VBS-processes simply processes, and we may also just write P instead of $P(u)$. By $\mathcal{P}$, we denote a finite set of processes under consideration.

Note that any action of a VBS-process is itself a finite CBS-process, hence a VBS-process can be seen as a sequential composition of CBS-processes. Moreover, note that the notion of load simplifies the model definition, although in the implementation it is in general not known a-priori.

Conceptually, the idea of decomposing a process into subtasks that run sequentially (the counterpart to actions in the VBS model) has appeared before, in the context of fixed-priority scheduling [10], and was extended in [4] for solving control-related issues.

As an illustration, let us now consider a simple theoretical example of processes and actions where the limit and period are expressed in seconds (which is the granularity of the time line).

Given a set $\mathcal{R} = \{(1, 2), (1, 4), (1, 3)\}$ of resources, we consider a finite process $P(0.5)$ that first does some computation for 3 s with a virtual periodic resource $(1, 2)$, then it works on allocating/deallocating memory objects of size 200 KB, which takes 2 s with the resource $(1, 4)$, then it produces output of size 100 KB on an I/O device in 1 s with $(1, 3)$, then again it computes, now for 2 s, with $(1, 2)$ again. We can represent P as a finite sequence

$$p(0.5) = \alpha_0\alpha_1\alpha_2\alpha_3 = (3, (1, 2))(2, (1, 4))(1, (1, 3))(2, (1, 2))$$

on the $1s$-timeline. This process corresponds to (can be served by) a VBS with utilization $u = 0.5$ (or more).

2.2 Scheduling

A schedule for a finite set of processes $\mathcal{P}$ is a partial function $\sigma : \mathbb{N} \hookrightarrow \mathcal{P}$ from the time domain to the set of processes that assigns to each moment in time a process that is running in the time interval $[t, t + 1)$. Here, $\sigma(t)$ is undefined if no process runs in $[t, t + 1)$. Due to the sequential nature of the processes, any scheduler σ uniquely determines a function $\sigma_{\mathcal{R}} : \mathbb{N} \hookrightarrow \mathcal{P} \times \mathcal{R}$ which specifies the resource a process uses while being scheduled.

A schedule respects the resource capacity if for any process $P \in \mathcal{P}$ and any resource $R \in \mathcal{R}$, with $R = (\lambda, \pi)$ we have that for any natural number $k \in \mathbb{N}$

$$|\{t \in [k\pi, (k + 1)\pi) \mid \sigma_{\mathcal{R}}(t) = (P, R)\}| \leq \lambda.$$

Hence, if the schedule respects the resource capacity, then the process P uses the resource R at most λ units of time per period of time π, as specified by its capacity.

Given a schedule σ for a set of processes $\mathcal{P}$, for each process $P \in \mathcal{P}$ and each action $\alpha_i = (l_i, R_i)$ that appears in P we distinguish four absolute moments in time:

- Arrival time a_i of the action α_i is the time instant at which the action arrives. We assume that a_i equals the time instant at which the previous action of the same process has finished. The first action of a process has arrival time zero.
- Completion time c_i of the action α_i is the time at which the action completes its execution. It is calculated as

$$c_i = \min\{c \in \mathbb{N} \mid l_i = |\{t \in [a_i, c) \mid \sigma(t) = P\}|\}.$$

- Finishing or termination time f_i of the action α_i is the time at which the action terminates or finishes its execution. We always have $f_i \geq c_i$. The difference between completion and termination is specified by the termination strategy of the scheduler. The process P can only invoke its next action if the previous one has been terminated. In the scheduling algorithm, we adopt the following termination strategy: an action is terminated at the end of the period within which it has completed. Adopting this termination strategy is needed for the correctness of the scheduling algorithm and the validity of the admission (schedulability) test.
- Release time r_i is the earliest time when the action α_i can be scheduled, $r_i \geq a_i$. If not specified otherwise, by the release strategy of the scheduler, we take $r_i = a_i$.

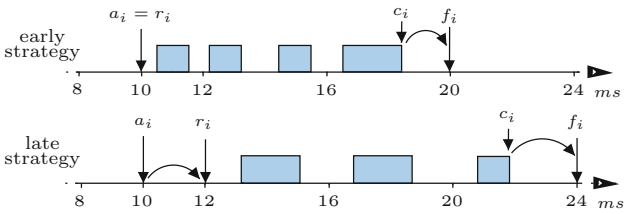


Fig. 1 Scheduling an action $\alpha = (5, (2, 4))$ [1]

In the scheduling algorithm, we will consider two release strategies, which we call early and late strategy.

We now present the two release strategies and elaborate the scheduling method via an example before continuing with the schedulability analysis.

In the late strategy, the release time of an action is delayed until the next period instance (of its resource) after the arrival time of the action. In the early strategy, the release time is equal to the arrival time, however, the limit of the action for the current period is adjusted so that it does not exceed its utilization in the remaining part of the current period. Our late strategy corresponds to the polling server [11] from classical scheduling theory, and the early strategy is similar in goal to the deferrable server [12]: it improves average response times since actions that arrive between period instances are not delayed.

Figure 1 presents the scheduling of an action $\alpha = (5, (2, 4))$ with load of 5 ms, arriving at time 10, in both strategies. The resource used by the action has a period of 4 and a limit of 2 ms. In the late strategy, the action is only released at time 12, which is the next period instance after the actual arrival time. Then, it takes three more periods for the action to finish. In the early strategy, the action is released at once, but in the remaining time of the current period (2 ms) the limit is adjusted to 1, so that the utilization remains 0.5. In this situation, the scheduled response time in the early release strategy is one period shorter than in the late release strategy. In both cases, the action splits into a sequence of three tasks that are released in the consecutive periods. In the early strategy, these tasks are released at time 10, 12, and 16; have deadlines 12, 16, and 20; and durations 1, 2, and 2, respectively. In the late strategy, the tasks are released at times 12, 16, and 20; have deadlines 16, 20, and 24; and durations of 2, 2, and 1, respectively.

Using these notions, we define response time under the scheduler σ of the action α denoted by s_i, as the difference between the finishing time and the arrival time, i.e., $s_i = f_i - a_i$. Note that this definition of response time is logical in the sense that all possible side effects of the action should take effect at termination but not before. In the traditional (non-logical) definition, response time is the time from arrival to completion, decreasing response time (increasing

performance) at the expense of increased jitter (decreased predictability).

Assume that the upper and lower response bounds b_i^u and b_i^l are given for each action α_i of each process P in a set of processes $\mathcal{P}$. The set $\mathcal{P}$ is schedulable with respect to the given bounds if and only if there exists a schedule $\sigma : \mathbb{N} \hookrightarrow \mathcal{P}$ that respects the resource capacity and for which the actual response times do not exceed the upper response bounds, i.e., $s_i \leq b_i^u$, and are greater than the lower response bounds, i.e., $s_i \geq b_i^l$, for all involved actions α_i.

2.3 Schedulability result

Given a finite set $\mathcal{P} = \{P_i(u_i) \mid 1 \leq i \leq n\}$ of processes with corresponding actions $\alpha_{i,j} = (l_{i,j}, R_{i,j})$ for $j \geq 0$, such that $P_i(u_i) = \alpha_{i,0}\alpha_{i,1}\ldots$ corresponds to a VBS with utilization u_i, we define the upper response-time bound as

$$b_{i,j}^u = \pi_{i,j} - 1 + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j}. \tag{1}$$

where $R_{i,j} = (\lambda_{i,j}, \pi_{i,j})$ with $l_{i,j}$, $R_{i,j}$, $\lambda_{i,j}$, and $\pi_{i,j}$ being as before the load, the resource, the limit, and the period for the action $\alpha_{i,j}$, respectively. Since an action $\alpha_{i,j}$ executes at most $\lambda_{i,j}$ of its load $l_{i,j}$ per period of time $\pi_{i,j}$, $\lceil \frac{l_{i,j}}{\lambda_{i,j}} \rceil$ is the number of periods the action needs in order to complete its load. In addition, in the response bound, we account for the time in the period in which the action arrives, which in the worst case is $\pi_{i,j} - 1$ if it arrives right after a period instance.

The lower response-time bound varies depending on the strategy used, namely

$$b_{i,j}^l = \begin{cases} \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j}, & \text{for late release} \\ \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \pi_{i,j}, & \text{for early release.} \end{cases} \tag{2}$$

Note that the lower bound in the early release strategy is achieved only if $\lambda_{i,j}$ divides $l_{i,j}$, in which case $\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \rfloor = \lceil \frac{l_{i,j}}{\lambda_{i,j}} \rceil$. From these bounds, we can derive that the response-time jitter, i.e., the difference between the upper and lower bound on the response time, is at most $\pi_{i,j} - 1$ for the late release strategy and at most $2\pi_{i,j} - 1$ for the early release strategy.

The next schedulability/admission result justifies the definition of the response bounds and shows the correctness of our scheduling algorithm.

Proposition 1 *Given a set of processes $\mathcal{P} = \{P_i(u_i) \mid 1 \leq i \leq n\}$, as above, if*

$$\sum_{i=1}^{n} u_i \leq 1, \tag{3}$$

then the set of processes $\mathcal{P}$ is schedulable with respect to the resource capacity and the response bounds (1) and (2).

The proposition shows that it is enough to test whether the sum of the utilization (bandwidth) caps of all processes is less than one. The test is finite even though the processes may be infinite. In addition, the test is computable even if the actual loads of the actions are unknown, as it is often the case in practice. Hence, the standard utilization-based test for CBS-processes, holds also for VBS-processes. The test runs in constant time, meaning that whenever a new VBS-process enters the system, it is decidable in constant time whether it can be admitted and scheduled.

In order to prove Proposition 1, we first isolate a more essential schedulability property in the following section. The proof of Proposition 1 follows in Sects. 2.5 and 2.6.

2.4 Typed EDF

We describe a schedulability test for a particular dynamic EDF [2] scheduling algorithm, and prove its sufficiency.

Let $\tau = (r, e, d)$ be an aperiodic task with release time r, execution duration e, and deadline d, all natural numbers. We say that τ has type, or specification, (λ, π) where λ and π are natural numbers, $\lambda \leq \pi$, if the following conditions hold:

- $d = (n+1)\pi$ for the (uniquely determined) natural number n such that $r \in [n\pi, (n+1)\pi)$, and
- $e \leq (d - r)\frac{\lambda}{\pi}$.

Hence, a task specification is basically a periodic task which we use to impose a bound on aperiodic tasks. Note that if $r = n\pi$, then the duration e is limited to λ. A task of type (λ, π) need not be released at an instance of the period π, but its utilization factor in the interval of time $[r, d]$ remains at most $\frac{\lambda}{\pi}$.

Let S be a finite set of task types. Let I be a finite index set, and consider a set of tasks

$$\{\tau_{i,j} = (r_{i,j}, e_{i,j}, d_{i,j}) \mid i \in I, j \geq 0\}$$

with the properties:

- Each $\tau_{i,j}$ has a type in S. We will write $(\lambda_{i,j}, \pi_{i,j})$ for the type of $\tau_{i,j}$.
- The tasks with the same first index are released in a sequence, i.e., $r_{i,j+1} \geq d_{i,j}$ and $r_{i,0} = 0$.

The following result provides us with a sufficient schedulability test for such specific set of tasks.

Lemma 1 *Let $\{\tau_{i,j} \mid i \in I, j \geq 0\}$ be a set of tasks as defined above. If*

$$U = \sum_{i \in I} \max_{j \geq 0} \frac{\lambda_{i,j}}{\pi_{i,j}} \leq 1, \tag{4}$$

then this set of tasks is schedulable using the EDF strategy at any point of time, so that each task meets its deadline.

Proof The proof builds upon the standard proof of sufficiency of the utilization test for periodic EDF, see, e.g., [13]. Assume the opposite, i.e., a deadline gets missed at time d by a task $\tau = (r, e, d) \in \{\tau_{i,j} \mid i \in I, j \geq 0\}$. Let t be the earliest moment in time such that in the interval $[t, d]$ there is full utilization and all executed tasks have deadlines that are not larger than d. Note that $t < d$ and t is a release time of some task.

Let $C(x, y)$ denote the computation demand of our set of tasks in an interval of time $[x, y]$. We have that $C(x, y)$ is the sum of the durations of all tasks with release time greater than or equal to x and deadline less than or equal to y.

Since a deadline is missed at d, we have

$$C(t, d) > d - t$$

i.e., the demand is larger than the available time in the interval $[t, d]$. We are going to show that $C(t, d) \leq (d - t)U$, which shows that $U > 1$ and completes the proof.

First we note that

$$C(t, d) = \sum_{i \in I} C_i(t, d)$$

where $C_i(t, d)$ is the computational demand imposed by tasks in $\{\tau_{i,j} \mid j \geq 0\}$ for a fixed $i \in I$.

In the finite interval $[t, d]$ only finitely many tasks in $\{\tau_{i,j} \mid j \geq 0\}$ are executed, say n tasks. Moreover, by the choice of t, none of these tasks is released at time earlier than t. Therefore, we can divide $[t, d]$ to subintervals

$$[t, d] = \bigcup_{k=0}^{n} [t_k, t_{k+1}]$$

where $t = t_0$ and $d = t_{n+1}$, and for all $k \in \{1, \ldots, n\}$, t_k is a release time of a task $\tau_k = (r_k, e_k, d_k)$ in $\{\tau_{i,j} \mid j \geq 0\}$. Since the tasks in $\{\tau_{i,j} \mid j \geq 0\}$ are released and executed in a sequence, we have that $d_k \leq t_{k+1}$. Let (λ_k, π_k) denote the type of τ_k. Moreover, we either have $t_1 = t_0$, or no task at all in $[t_0, t_1]$.

Denote by (λ_i^*, π_i^*) the "most expensive" task type in terms of utilization for $\{\tau_{i,j} \mid j \geq 0\}$, i.e.,

$$\frac{\lambda_i^*}{\pi_i^*} = \max_{j \geq 0} \frac{\lambda_{i,j}}{\pi_{i,j}}.$$

We have $C_i(t_0, t_1) = 0$ and for $k > 0$,

$$\begin{aligned} C_i(t_k, t_{k+1}) &\leq (d_k - r_k)\tfrac{\lambda_k}{\pi_k} \\ &\leq (t_{k+1} - t_k)\tfrac{\lambda_k}{\pi_k} \\ &\leq (t_{k+1} - t_k) \cdot \tfrac{\lambda_i^*}{\pi_i^*}. \end{aligned}$$

Hence for all $k \in \{0, \ldots, n\}$, it holds that

$$C_i(t_k, t_{k+1}) \leq (t_{k+1} - t_k) \cdot \frac{\lambda_i^*}{\pi_i^*}.$$

Therefore,

$$\begin{aligned} C(t, d) &= \textstyle\sum_{i \in I} C_i(t_0, t_n) \\ &= \textstyle\sum_{i \in I} \sum_{k=0}^{n} C_i(t_k, t_{k+1}) \\ &\leq \textstyle\sum_{i \in I} \sum_{k=0}^{n} (t_{k+1} - t_k)\frac{\lambda_i^*}{\pi_i^*} \\ &= \textstyle\sum_{i \in I} (t_{n+1} - t_0)\frac{\lambda_i^*}{\pi_i^*} \\ &= (d - t)U. \end{aligned}$$

which completes the proof. □

The schedulability test (4) computes the maximal utilization from the tasks in $\{\tau_{i,j} \mid j \geq 0\}$, given by the "most expensive" type. Since there are finitely many types, even though the set $\{\tau_{i,j} \mid j \geq 0\}$ may be infinite, the test is computable. Clearly, the test is conservative. For finite or "periodic" sets $\{\tau_{i,j} \mid j \geq 0\}$, one could come up with a more complex sufficient and necessary utilization test based on the overlap of the tasks. We leave such an investigation for future work.

We now present the proof of Proposition 1 first for the upper and then for the lower response-time bounds.[1]

2.5 Upper response-time bound

Each process P_i for $i \in I$ provides a sequence of tasks $\tau_{i,k}$ by refining each action to a corresponding sequence of tasks. Consider the action $\alpha_{i,j} = (l_{i,j}, R_{i,j})$ with capacity of $R_{i,j}$ given by $(\lambda_{i,j}, \pi_{i,j})$. Let n_j be a natural number such that

$$a_{i,j} \in \big((n_j - 1)\pi_{i,j}, n_j \pi_{i,j}\big]$$

if $j > 0$, and let $n_0 = 0$. We distinguish two cases, one for each release strategy.

Case 1 Late release strategy Let

$$k_j = \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil.$$

The action $\alpha_{i,j}$ produces tasks $\tau_{i,k}$ for $k_0 + \cdots + k_{j-1} \leq k \leq k_0 + \cdots + k_{j-1} + k_j - 1$ given by:

$$\tau_{i,k} = \big((n_j + m)\pi_{i,j}, e_{i,k}, (n_j + m + 1)\pi_{i,j}\big)$$

where $m = k - (k_0 + \cdots + k_{j-1})$ and $e_{i,k} = \lambda_{i,j}$ if $k < k_0 + \cdots + k_j - 1$ or if $k = k_0 + \cdots + k_j - 1$ and $\lambda_{i,j}$ divides $l_{i,j}$, otherwise $e_{i,k} = l_{i,j} - \lfloor \frac{l_{i,j}}{\lambda_{i,j}} \rfloor \cdot \lambda_{i,j}$.

Hence, the workload of the action $\alpha_{i,j}$ is split into several tasks that all have type $(\lambda_{i,j}, \pi_{i,j})$. Moreover, the tasks in $\{\tau_{i,k} \mid k \geq 0\}$ are released in a sequence, such that (because of the termination strategy and the resource capacity) the release time of the next task is always equal or grater than the deadline of a given task. Therefore, Lemma 1 is applicable, and from the utilization test we get that the set of tasks

[1] The proof for the lower response-time bound has also appeared in [6].

$\{\tau_{i,k} \mid i \in I, k \geq 0\}$ is schedulable so that all tasks meet their deadlines. Let σ be a schedule for this set of tasks. It corresponds to a schedule $\hat{\sigma}$ for the set of processes $\mathcal{P}$ by: $\hat{\sigma}(t) = P_i$ if and only if $\sigma(t) = \tau_{i,k}$ for some $k \geq 0$.

By construction, $\hat{\sigma}$ respects the resource capacity: Consider $P_i \in \mathcal{P}$ and $R \in \mathcal{R}$ with capacity (λ, π). In any interval of time $[n\pi, (n+1)\pi)$ there is at most one task $\tau_{i,k}$ of type (λ, π) produced by an action with resource R which is available and running, and its duration is limited by λ.

For the bounds, for each action $\alpha_{i,j}$, according to the termination strategy and the late release, we have

$$f_{i,j} = r_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j}$$

where the release times are given by $r_{i,j} = n_j \pi_{i,j}$. The arrival times are $a_{i,j} = f_{i,j-1} \in \big((n_j - 1)\pi_{i,j}, n_j \pi_{i,j}\big]$. Therefore,

$$\begin{aligned} s_{i,j} &= f_{i,j} - a_{i,j} \\ &= \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} + r_{i,j} - a_{i,j} \\ &\leq \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} + \pi_{i,j} - 1 \\ &= b^u_{i,j} \end{aligned}$$

which completes the proof in the case of the late strategy. Hence, if the release time of each action is delayed to the next period instance, then we safely meet the response bounds. However, such a delay is not necessary. We may keep the release time equal to the arrival time and still meet the bounds. On average, we may achieve better response times than indicated by the bounds and also higher utilization.

Case 2 Early release strategy If the action $\alpha_{i,j}$ arrives on a period instance $\alpha_{i,j} = n_j \pi_{i,j}$ then there is nothing we can do better than in the late strategy. If not, then let

$$e_{i,j} = \min \left\{ l_{i,j}, \left\lfloor \left(n_j \pi_{i,j} - a_{i,j}\right) \frac{\lambda_{i,j}}{\pi_{i,j}} \right\rfloor \right\}$$

and

$$k_j = \left\lceil \frac{l_{i,j} - e_{i,j}}{\lambda_{i,j}} \right\rceil + 1.$$

Then $\alpha_{i,j}$ produces k_j tasks $\tau_{i,k}$ for $k_0 + \cdots + k_{j-1} \leq k \leq k_0 + \cdots + k_{j-1} + k_j - 1$ given by: for $k = k_0 + \cdots + k_{j-1}$

$$\tau_{i,k} = \big(a_{i,j}, e_{i,j}, n_j \pi_{i,j}\big),$$

and if $k_j > 1$, then for $k_0 + \cdots + k_{j-1} < k < k_0 + \cdots + k_{j-1} + k_j - 1$

$$\tau_{i,k} = \big((n_j + m)\pi_{i,j}, \lambda_{i,j}, (n_j + m + 1)\pi_{i,j}\big),$$

where $m = k - (k_0 + \cdots + k_{j-1} + 1)$. Now if $\lambda_{i,j}$ divides $l_{i,j} - e_{i,j}$, then also for $k = k_0 + \cdots + k_{j-1} + k_j - 1$ we have

$$\tau_{i,k} = \big((n_j + m)\pi_{i,j}, \lambda_{i,j}, (n_j + m + 1)\pi_{i,j}\big),$$

with $m = k - (k_0 + \cdots + k_{j-1} + 1)$. If, on the other hand, $\lambda_{i,j}$ does not divide $l_{i,j} - e_{i,j}$, then for $k = k_0 + \cdots + k_{j-1} + k_j - 1$ we have

$$\begin{aligned} \tau_{i,k} = \big((n_j + m)\pi_{i,j}, l_{i,j} - e_{i,j} \\ - \left\lfloor \frac{l_{i,j} - e_{i,j}}{\lambda_{i,j}} \right\rfloor \cdot \lambda_{i,j}, (n_j + m + 1)\pi_{i,j}\big), \end{aligned}$$

where again $m = k - (k_0 + \cdots + k_{j-1} + 1)$.

Hence, we let a task of $\alpha_{i,j}$ start as soon as $\alpha_{i,j}$ has arrived, taking care not to exceed the limit in the current period as well as to keep the utilization below the specified bound (via the duration of the task $e_{i,j}$). The rest of the action is divided in tasks as before. Note that all tasks produced by $\alpha_{i,j}$ are still of type $(\lambda_{i,j}, \pi_{i,j})$. Also in this case the termination strategy makes sure that each release time is larger than or equal to the deadline of the previous task. Hence, the set of tasks is schedulable via Lemma 1, and the induced process schedule respects the resource capacity. For the bounds, we now have

$$f_{i,j} \leq r_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} + \pi_{i,j} - 1$$

and $a_{i,j} = r_{i,j}$. Hence,

$$\begin{aligned} s_{i,j} &= f_{i,j} - a_{i,j} \\ &\leq \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} + \pi_{i,j} - 1 \\ &= b^u_{i,j} \end{aligned}$$

which completes the proof.

2.6 Lower response-time bound

For each action $\alpha_{i,j}$, according to the termination strategy and the late release strategy, we have

$$f_{i,j} = r_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} \tag{5}$$

where the release times are given by $r_{i,j} = n_j \pi_{i,j}$ for some natural number n_j such that the arrival times are $a_{i,j} = f_{i,j-1} \in \big((n_j - 1)\pi_{i,j}, n_j \pi_{i,j}\big]$. Therefore, for the late strategy we have

$$\begin{aligned} s_{i,j} = f_{i,j} - a_{i,j} &\overset{(5)}{=} \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} + r_{i,j} - a_{i,j} \\ &\geq \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} = b^l_{i,j}, \text{ for late release.} \end{aligned}$$

For the early release strategy, we distinguish two cases depending on whether the following inequality holds

$$\left\lfloor m \frac{\lambda_{i,j}}{\pi_{i,j}} \right\rfloor \geq l_{i,j} - \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \lambda_{i,j} \tag{6}$$

where $m = n_j \pi_{i,j} - r_{i,j}$ and $r_{i,j} = a_{i,j} = f_{i,j-1} \in \big((n_j - 1)\pi_{i,j}, n_j \pi_{i,j}\big]$. The finishing time for the action $\alpha_{i,j}$ is

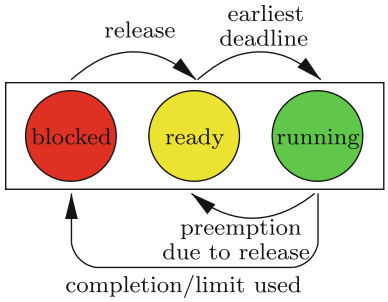


Fig. 2 Process states [1]

$$f_{i,j} = \begin{cases} a_{i,j} + \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \pi_{i,j} + m, & \text{if (6) holds} \\ a_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \pi_{i,j} + m, & \text{otherwise} \end{cases} \tag{7}$$

In both cases, $f_{i,j} \geq \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \pi_{i,j} + a_{i,j}$, so

$$\begin{aligned} s_{i,j} &= f_{i,j} - a_{i,j} \\ &\geq \left\lfloor \frac{l_{i,j}}{\lambda_{i,j}} \right\rfloor \pi_{i,j} = b^l_{i,j}, \quad \text{for early release.} \end{aligned}$$

3 Scheduling algorithm

In this section, we describe the scheduling algorithm which follows the proof of Proposition 1. At any relevant time t, our system state is determined by the state of each process. A process may be blocked, ready, or running as depicted in Fig. 2. By Blocked, Ready, and Running, we denote the current sets of blocked, ready, and running processes, respectively. These sets are ordered: Blocked is ordered by the release times, Ready is ordered by deadlines, and Running is either empty (for an idle system) or contains the currently running process of the system. Thus,

$$\mathcal{P} = \text{Blocked} \cup \text{Ready} \cup \text{Running}$$

and the sets are pairwise disjoint. Additionally, each process is represented by a tuple in which we keep track of the process evolution. For the process P_i, we have a tuple

$$(i, j, d_i, r_i, l^c_i, \lambda^c_i)$$

where i is the process identifier, j stores the identifier of its current action $\alpha_{i,j}$, d_i is the current deadline (which is not the deadline for the entire action, but rather an instance of the action period $\pi_{i,j}$), r_i is the next release time, l^c_i is the current load, and λ^c_i is the current limit. The scheduler also uses a global time value t_s which stores the previous time instant at which the scheduler was invoked.

Given n processes $P_1, \ldots, P_n$, as defined in the previous section, initially we have

$$\text{Blocked} = \{P_1, \ldots, P_n\}, \ \text{Ready} = \text{Running} = \emptyset.$$

At specific moments in time, including the initial time instant, we perform the following steps:

1. Update process state for the process in Running.
2. Move processes from Blocked to Ready.
3. Update the set Running.

We discuss each step in more detail below.

1. If Running $= \emptyset$, i.e., the system was idle, we skip this step. Otherwise, let P_i be the process in Running at time t. We differentiate three reasons for which P_i is preempted at time t: completion, limit, and release.

   Completion P_i completes the entire work related to its current action $\alpha_{i,j} = (l_{i,j}, R_{i,j})$. If we have reached process termination, i.e., there is no next action, we have a zombie process and remove it from the system. Otherwise, $j \leftarrow j + 1$ and the current action becomes $\alpha_{i,j+1} = (l_{i,j+1}, R_{i,j+1})$ with the resource capacity $(\lambda_{i,j+1}, \pi_{i,j+1})$. The current load l^c_i becomes $l_{i,j+1}$.
   If $R_{i,j+1} = R_{i,j}$, i.e., the resource of the next action remains the same, then there is no need to reschedule: P_i is moved to Ready, its deadline d_i, and release time r_i remain unchanged, and we subtract the work done from $\lambda^c_i, \lambda^c_i \leftarrow \lambda^c_i - (t - t_s)$.
   If $R_{i,j+1} \neq R_{i,j}$, we have currently implemented two release strategies as described above. First we take care of the termination strategy. Let $m \in \mathbb{N}$ be a natural number such that $t \in ((m-1)\pi_{i,j}, m\pi_{i,j}]$. According to our termination strategy, the action $\alpha_{i,j}$ is terminated at time $m\pi_{i,j}$ which is the end of the period in which the action has completed. Now let $k \in \mathbb{N}$ be a natural number such that

   $$m\pi_{i,j} \in ((k-1)\pi_{i,j+1}, k\pi_{i,j+1}].$$

   The first strategy, the late release strategy, calculates r_i, the next release time of P_i, as the start of the next period of $R_{i,j+1}$ and its deadline as the start of the second next period,

   $$r_i \leftarrow k\pi_{i,j+1}, \ d_i \leftarrow (k+1)\pi_{i,j+1}.$$

   The new current limit becomes $\lambda_{i,j+1}$ and P_i is moved to Blocked.
   The second strategy, the early release strategy, sets the release time to the termination time and the deadline to the end of the release-time period

   $$r_i \leftarrow m\pi_{i,j}, \ d_i \leftarrow k\pi_{i,j+1}$$

and calculates the new current limit for P_i, as

$$\lambda_i^c \leftarrow \left\lfloor (d_i - r_i)\frac{\lambda_{i,j+1}}{\pi_{i,j+1}} \right\rfloor.$$

The process P_i is moved to Blocked.

Limit P_i uses all of the current limit λ_i^c for the resource $R_{i,j}$. In this case, we update the current load, $l_i^c \leftarrow l_i^c - (t - t_s)$, and

$$\lambda_i^c \leftarrow \lambda_{i,j},\ r_i \leftarrow k\pi_{i,j},\ d_i \leftarrow (k+1)\pi_{i,j},$$

with $k \in \mathbb{N}$ such that $t \in ((k-1)\pi_{i,j}, k\pi_{i,j}]$. With these new values P_i is moved to Blocked.

Release If a process is released at time t, i.e., P_m is a process, $P_m \neq P_i$, with the release time $r_m = t$, then the priorities have to be established anew. We update the current load and limit,

$$l_i^c \leftarrow l_i^c - (t - t_s),\ \lambda_i^c \leftarrow \lambda_i^c - (t - t_s).$$

The deadline for P_i is set to the end of the current period, $d_i \leftarrow k\pi_{i,j}$, with $k \in \mathbb{N}$ such that $t \in ((k-1)\pi_{i,j}, k\pi_{i,j}]$. P_i is then moved to Ready.

2. In the second step, the scheduler chooses the processes from Blocked which are to be released at the current time t, i.e., $\{P_i \mid r_i = t\}$, and moves them to the set Ready.
3. In the third step if the Ready set is empty, the scheduler leaves the Running set empty, thus the system becomes idle. Otherwise, the scheduler chooses a process P_i with the earliest deadline from Ready and moves it to Running.

We calculate:

- t_c: the time at which the new running process P_i would complete its entire work needed for its current action without preemption, i.e., $t_c = t + l_i^c$.
- t_l: the time at which P_i consumes its current limit for the current period of the resource R_i, i.e., $t_l = t + \lambda_i^c$.
- t_r: the next release time of any process in Blocked. If Blocked is empty, $t_r = \infty$.

The scheduler stores the value of the current time in t_s, $t_s \leftarrow t$, and the system lets P_i run until the time $t = \min(t_c, t_l, t_r)$ at which point control is given back to the scheduling algorithm.

As stated, the algorithm uses knowledge of the load of an action. However, in the implementation there is a way around it (by marking a change of action that forces a scheduler invocation) which makes the algorithm applicable to actions with unknown load as well, in which case no explicit response-time guarantees are given. The complexity of the scheduling algorithm amounts to the complexity of the plugins that manage the ordered Blocked and Ready sets, the rest of the algorithm has constant-time complexity.

4 Implementation

The scheduler uses a well-defined interface to manage the processes in the system. This interface is implemented by three alternative plugins, each with different attributes regarding time complexity and space overhead. Currently, our implementation, available via the Tiptoe homepage [14], supports doubly-linked lists (Sect. 4.1), time-slot arrays of FIFO queues (Sect. 4.2), a time-slot matrix of FIFO queues (Sect. 4.3), and a tree-based optimization of the matrix. The implementation details can also be found in a technical report version [5] of [1].

The array and matrix implementation impose a bound on the number of time instants. For this reason, we introduce a finite coarse-grained timeline with t time instants and a distance between any two instants equal to a fixed natural number d. Deadlines and release times are then always in the coarse-grained timeline, which restricts the number of different periods in the system. The scheduler may be invoked at any time instant of the original (fine-grained) timeline. However, the second step of the algorithm (releasing processes) is only executed during scheduler invocations at time instants of the coarse-grained timeline.

The matrix- and tree-based implementations are $O(1)$-schedulers since the period resolution is fixed. However, not surprisingly, temporal performance comes at the expense of space complexity, which grows quadratically in period resolution for both plugins. Space consumption by the tree plugin is significantly smaller than with the matrix plugin, if the period resolution is higher than the number of servers. The array-based implementation runs in linear time in the number of servers and requires linear space in period resolution. The list-based implementation also runs in linear time in the number of servers, but only requires space independent of period resolution (although insertion is more expensive than for the array plugin).

Table 1 shows the system's time and space complexities distinguished by plugin in terms of the number of processes in the system (n), and in the period resolution, that is, the number of time instants the system can distinguish (t). For efficiency, we use a time representation similar to the circular time representation of [15].

Table 2 summarizes the queue operations' time complexity in terms of the number of processes (n) and the number of time instants (t). The first operation is called ordered-insert by which processes are inserted according to a key, and

Table 1 Time and space complexity per plugin

	List	Array	Matrix/tree
Time	$O(n)$	$O(\log(t) + n\log(t))$	$\Theta(t)$
Space	$\Theta(n)$	$\Theta(t+n)$	$O(t^2+n)$

Table 2 Time complexity of the queue operations

	List	Array	Matrix
Ordered-insert	$O(n)$	$\Theta(\log(t))$	$\Theta(\log(t))$
Select-first	$\Theta(1)$	$O(\log(t))$	$O(\log(t))$
Release	$O(n)$	$O(\log(t) + n \cdot \log(t))$	$\Theta(t)$

processes with the same key are kept in FIFO order to maintain fairness. The select-first operation selects the first element in the respective queue. The release operation finds all processes with a certain key, reorders them according to a new key, and merges the result into another given queue. Note that t is actually a constant, so the matrix implementation achieves constant time for all three operations.

We now describe each plugin in more detail.

4.1 Process list

The list plugin uses ordered doubly-linked lists for Ready, which is sorted by deadline, and Blocked, which is sorted first by release time and then by deadline. Therefore, inserting a single element has linear complexity with respect to the number of processes in the queue, while selecting the first element in the queue is done in constant time. Releasing processes in Blocked which contains k processes by moving them to Ready, which contains m processes, takes $k+m$ steps. The upper bound of k and m is n and, therefore, the complexity is $O(n)$. Advantages of this data structure are low memory usage (only two pointers per process) and no limitation on the resolution of the timeline.

4.2 Time-slot array

The array plugin uses an array of pointers to represent the timeline. Each element in the array points to a FIFO queue of processes. A pointer at position t_i in Blocked, for instance, points to a FIFO queue of processes that are to be released at time t_i. In Ready, a pointer at position t_i is a reference to a FIFO queue of processes with a deadline t_i. Note that whenever we speak of time instants in the array or matrix plugins, we mean time instants modulo the size of the array or matrix, respectively.

In a naive implementation, inserting a process would be achieved in constant time using the key (release time or deadline) as index into the array. Finding the first process of this array would then be linear in the number of time instants (t). A more balanced version uses an additional bitmap to represent whether there are any processes at a certain time instant or not. The bitmap is split into words with an additional header bitmap that indicates which word of the bitmap has at least one bit set. Furthermore, if the header bitmap has more than s bits, where s denotes the word size,[2] it recursively maintains a header again. The bitmap implementation, therefore, can be seen as a tree with depth $\log_s(t)$ where the nodes are words and each node has s children. In this way, the select-first operation improves from linear complexity to $\log_s(t)$, but the ordered-insert operation degrades from constant time to $\log_s(t)$ complexity, due to necessary updates in the bitmap.

During the release operation at time instant t_i, all k processes in the FIFO queue at position t_i in the Blocked array are inserted at the correct position in the Ready array. The complexity of this operation is $k \cdot \log_s(t)$. The bit which indicates that there are processes at time instant t_i in Blocked is cleared in $\log_s(t)$ steps. Thus, the time complexity of the release operation is at most $n \cdot \log_s(t) + \log_s(t)$, since n is the upper bound of k.

The disadvantage of this plugin is the static limit on the timeline, i.e., the number of time-slots in an array. This imposes a limitation of how far into the future a process can be released, and on the maximum deadline of a process. Therefore, the possible range of resource periods in the system is bounded with this plugin. Furthermore, the array introduces a constant memory overhead. Each array with t time-slots, contains t pointers and $(s/(s-1)) \cdot (t-1)$ bits for the bitmap. For example, with $t = 1{,}024$ this results in 4 KB for the pointers and 132 bytes for the bitmap.

4.3 Time-slot matrix

In order to achieve constant execution time in the number of processes for all operations on the queues we have designed a matrix of FIFO queues, also referred to as FIFO matrix. The matrix contains all processes in the system, and the position in the matrix indicates the processes deadline (column entry) and the processes release time (row entry). The matrix implicitly contains both Ready and Blocked, which can be computed by providing the current time. In a naive implementation, select-first has complexity $O(t^2)$, whereas insert-ordered and release are constant time. To balance this, additional meta-data are introduced which reduces the complexity of select-first to $O(\log(t))$ and degrades the complexity of the other operations (cf. Table 2).

[2] Our implementation supports 32-bit and 64-bit word size, on corresponding CPU architectures. The measurements and example calculations were done for $s = 32$.

We introduce a two-dimensional matrix of bits, having value 1 at the positions at which there are processes in the original matrix. In addition, we use two more bitmaps, called release bitmap and ready bitmap. The release bitmap indicates at which row in the matrix of bits at least one bit is set. The ready bitmap contains the information in which columns released processes can be found. Note that the release bitmap merely reflects the content of the matrix. The ready bitmap provides additional information, it indicates where the currently released processes are located.

A process is put into the FIFO matrix in constant time. However, the corresponding updates of the bit matrix and the release bitmap take $\log_s(t)$ operations each. Therefore, inserting a process has a time complexity of $2 \cdot \log_s(t)$. Finding and returning the first process in Released or Blocked has also a complexity of $2 \cdot \log_s(t)$. To find the first process in Ready, e.g., we find the first set bit in the ready bitmap in $\log_s(t)$ operations. If the bit is at position i, then the ith column in the bit matrix is examined, in order to find the set bit j corresponding to the first process in Ready, also in at most $\log_s(t)$ operations. The two indexes, i and j, are then used to access the process in the FIFO matrix. As a result, the operation of selecting the first process has a total complexity of $2 \cdot \log_s(t)$.

The release operation does not involve moving processes. Releasing processes is done by updating the ready bitmap. More precisely, the new ready bitmap for time instant t_i is the result of a logical OR between row t_i in the bit matrix and the old ready bitmap. The OR operation is word-wise and, therefore, linear in the size of the bitmap, which is linear in the number of time instants.

In addition to the static limitation for the number of time instants, a disadvantage of the matrix plugin is the high memory usage. To distinguish t time instants the FIFO matrix uses t^2 pointers. Additionally, the meta-data consist of $(s/(s-1)) \cdot t \cdot (t-1)$ bits for the bit matrix and $(s/(s-1)) \cdot (t-1)$ bits for each bitmap. In order to fully exploit the available hardware instruction for searching and modifying bitmaps, the transpose of the bit matrix is also kept in memory, which adds additional $(s/(s-1)) \cdot t \cdot (t-1)$ bits.

As an alternative to the FIFO matrix representation, we also implemented the FIFO matrix as a B+ tree [16]. Using a B+ tree for the FIFO matrix adds $2 \cdot \log_s(t)$ operations to the complexity of the ordered-insert and select-first operations, because the depth of the tree is $2 \cdot \log_s(t)$. The memory usage of the B+ tree might in the worst case exceed the memory usage of the FIFO matrix. A worst-case scenario occurs when each position of the FIFO matrix contains at least one process. However, if the FIFO matrix is sparse, e.g., because the number of processes in the system is much smaller than the number of distinguishable time instants, then the memory overhead reduces drastically. See the next section for details.

5 Experiments and results

We present results of different experiments with the scheduler implementation, running on a 2GHz AMD64 machine with 4GB of memory.

5.1 Scheduler overhead

In order to measure scheduler execution times, we schedule 9 different sets of simulated processes with 10, 25, 50, 75, 100, 150, 250, 500, and 750 processes each, with the number of distinguishable time instants t in the scheduler fixed to $2^{14} = 16{,}384$. Each process has a variable number of actions with random periods and loads. The limit of every action is 1. During these experiments, the execution time of every single scheduler invocation is measured using the software oscilloscope tool TuningFork [17]. From a sample of one million invocations, we calculate the maximum (Fig. 3a), the average (Fig. 3b), and the standard deviation (Fig. 3c) in execution times. The x axis of each of the three figures represents the number of processes in the set and the y axis the execution time in microseconds. The B+ tree plugin performs the same as the matrix plugin up to 140ns and is, therefore, not shown. Note that the experimental results with the list plugin were obtained using an implementation described in [1,5] which had quadratic time complexity.

The execution time measurements conform to the complexity bounds from Sect. 4. For a low number of processes (less than 150), all plugins perform similarly and the scheduler needs at most 20 µs. On average (Fig. 3b), for a low number of processes (up to 100) the list plugin is the fastest. Interestingly, on average the array plugin is always faster than the matrix plugin, even for a high number of processes. The reason is that the constant overhead of the matrix operations is higher, which can be seen in the average but not in the maximal execution times.

The variability (jitter) of the scheduler execution can be expressed in terms of its standard deviation, depicted in Fig. 3c. The variability of the list and array plugins increases similarly to their maximum execution times when more than 150 processes are scheduled. The matrix plugin, however, has a lower standard deviation for a high number of processes and a higher standard deviation for a low number of processes. This is related to the better average execution time (Fig. 3b) for higher number of processes, as a result of cache effects. By instrumenting the scheduler we discovered that bitmap functions, e.g., setting a bit, are on average up to four times faster with 750 processes than with 10 processes, which suggests CPU cache effects.

The memory usage of all plugins, including the tree plugin, for 750 processes with an increasing number of distinguishable time instants is shown in Fig. 3d. The memory usage of just the B+ tree is 370 KB, compared to the 1GB for the

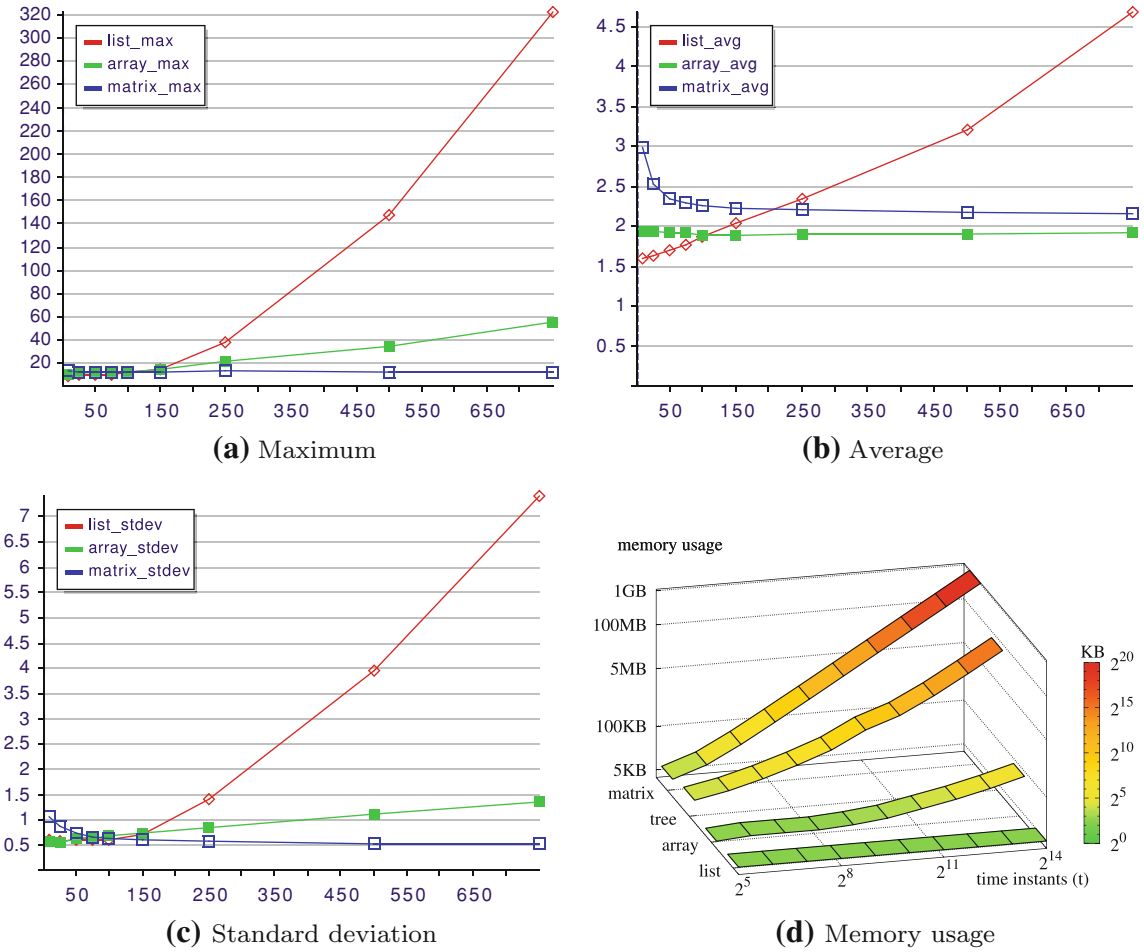


Fig. 3 Scheduler time (**a**–**c**) [x axis: number of processes, y axis: execution time in microseconds] and space overhead (**d**) [1]

matrix plugin. In both cases up to 66 MB additional memory is used for meta-data, which dominates the memory usage of the tree plugin. The graphs in Fig. 3d are calculated from theoretical bounds. However, our experiments confirm the results.

Figure 4a–c shows the different behavior of the presented plugins when scheduling 750 processes. These figures are histograms of the scheduler execution time and are used to highlight the distribution of it. The x axis represents the execution time in microseconds and the y axis (log-scale) represents the number of scheduler calls. For example, in Fig. 4a there are about 50 scheduler calls that executed for 100 µs during the experiment.

The list plugin varies between 0 and 350 µs, the array plugin between 0 and 55 µs, and the matrix plugin does not need more than 20 µs for any scheduler execution. The execution time histograms, especially histogram 4(a), are closely related to the histogram of the number of processes released during the experiment (Fig. 4d). The x axis represents the number of released processes and the y axis (log-scale) represents how many times a certain number of processes is released. The similarity of Fig. 4a and d indicates that the release of processes dominates the execution of the scheduler for the experiment with 750 processes.

5.2 Release strategies

In this section we compare the two implemented release strategies of the scheduler in two experiments and show that the early strategy achieves optimal average response times (always better by one period than the late strategy) for a single process with increasingly non-harmonic periods (Fig. 5a, top), and improves average response times for an increasing number of processes with a random distribution of loads, limits, and periods (Fig. 5b, top). In both experiments, response times are in milliseconds, and limits and periods are chosen such that the theoretically possible CPU utilization (Proposition 1) is close to one. The early strategy achieves at least as high actual CPU utilization as the late strategy, and thus less CPU idle time (bottom part of both figures). For Fig. 5a, the single process alternates between two actions that have their periods (and limits) equal to some natural number n

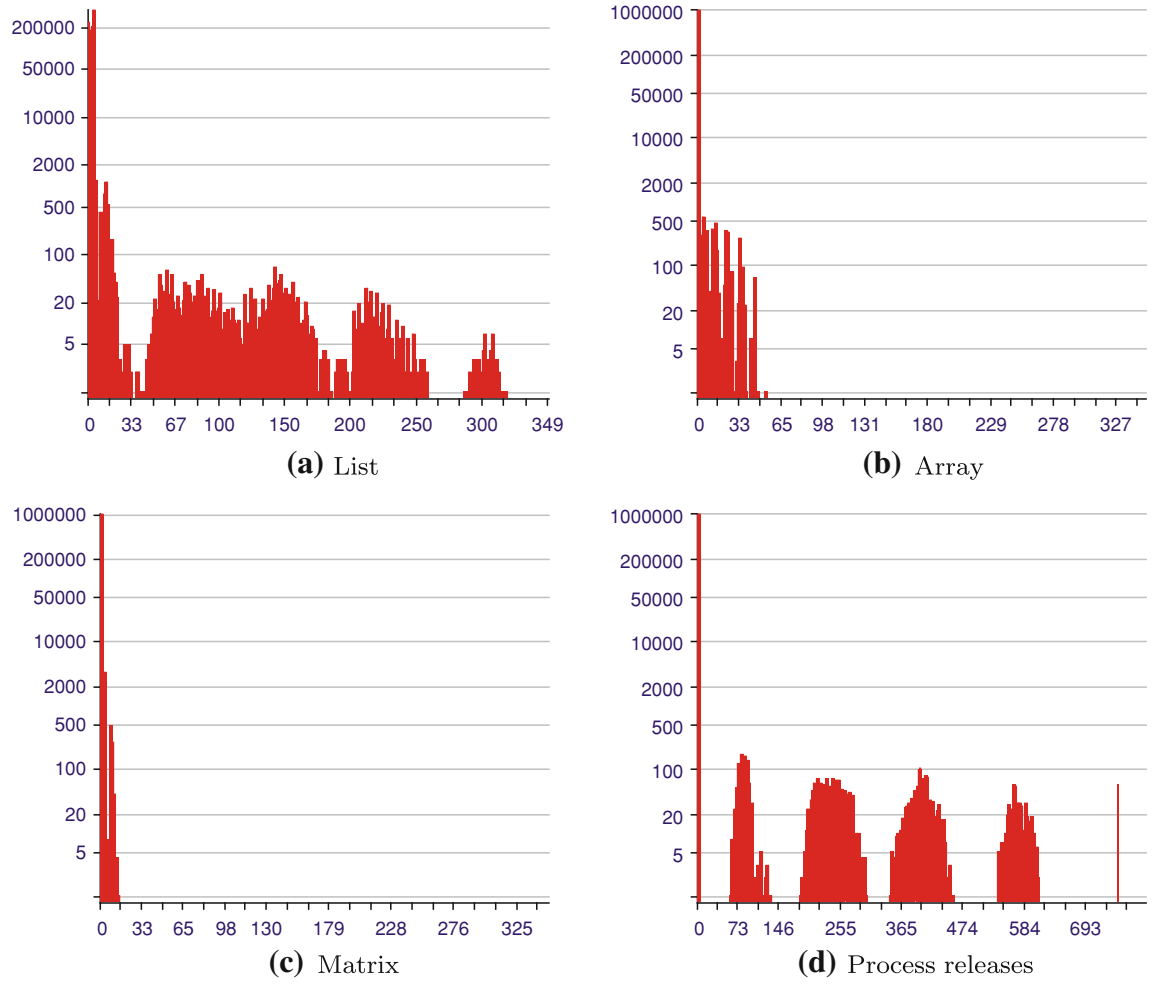


Fig. 4 Execution time histograms (**a–c**) [x axis: execution time in microseconds, y axis: (log-scale) number of scheduler calls] and process release histogram (**d**) [x axis: number of released processes, y axis: (log-scale) number of scheduler calls] [1]

and $n+1$, respectively, shown as pairs $(n, n+1)$ on the x axis. Hence, the actions are increasingly non-harmonic and the corresponding periods are relatively prime. The process always invokes the actions on the lowest possible load to maximize switching between actions resulting in increasingly lower CPU utilization.

6 Scheduler overhead accounting

The response-time bounds (1) and (2) presented in Sect. 2.3 are the result of several assumptions on the process and system model. One important assumption that has been implicitly made in the previous sections and which is prevalent in literature is that the scheduler overhead is zero. However, in a real system the effect of the scheduler overhead on the response-time bounds of processes (or actions) can be substantial. We (a subset of the authors) have extended the VBS schedulability and response-time bound analysis to include scheduler overhead in [6]. Here, we present a condensed version of that analysis for completeness of the VBS result.

The first step towards including the scheduler overhead in the VBS analysis is to determine an upper bound on the number of scheduler invocations that can occur during a time interval. In particular, we want to determine the worst-case number of scheduler invocations that an action of a VBS process experiences during one period.

The duration of a scheduler invocation is typically several orders of magnitude lower than a unit execution of an action. Therefore, we make the assumption that all periods belong to the set of discrete time instants $M = \{c \cdot n \mid n \geq 0\} \subset \mathbb{N}$, for a constant value $c \in \mathbb{N}$, $c > 1$. Hence, for any action $\alpha_{i,j}$ with its associated virtual periodic resource $R_{i,j} = (\lambda_{i,j}, \pi_{i,j})$ we have that $\pi_{i,j} = c \cdot \pi'_{i,j}$ with $\pi'_{i,j} \in \mathbb{N}$. We call c the scale of the system. Intuitively, we can say that there are two different timelines, the "fine-grained timeline" given by the set of natural numbers and the "coarse-grained timeline" given by the set M. Resource periods are defined on the

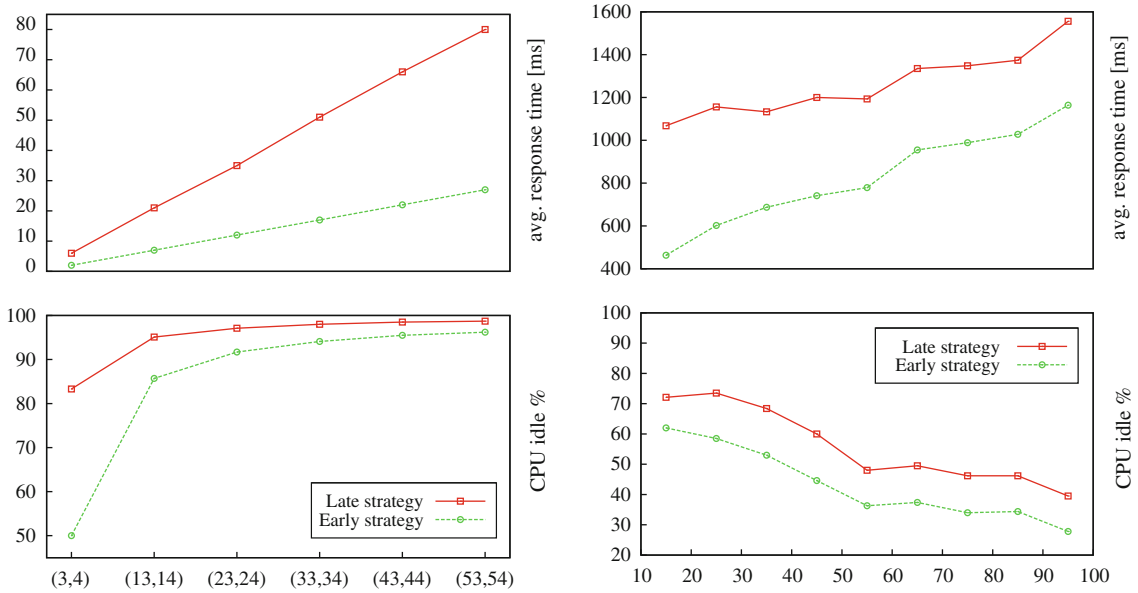


Fig. 5 Release strategies comparisons [5]

"coarse-grained timeline", while the execution time of the scheduler is defined on the "fine-grained timeline".

In VBS scheduling, a process P_i is preempted at a time instant t if and only if one of the following situations occurs:

1. *Completion* P_i has completed the entire work related to its current action $\alpha_{i,j} = (l_{i,j}, R_{i,j})$.
2. *Limit* P_i uses up all resource limit $\lambda_{i,j}$ of the current resource $R_{i,j}$.
3. *Release* A task of an action is released at time t, i.e., an action of another process is activated. Note that all preemptions due to release occur at time instants on the "coarse-grained timeline", the set M.

The following result holds.

Lemma 2 [6] *Let* $\mathcal{P} = \{P_i(u_i) \mid 1 \leq i \leq n\}$ *be a set of VBS processes with actions* $\alpha_{i,j}$ *and corresponding virtual periodic resources* $(\lambda_{i,j}, \pi_{i,j})$. *There are at most* $N_{i,j} = N^R_{i,j} + 1$ *scheduler invocations every period* $\pi_{i,j}$ *for the action* $\alpha_{i,j}$, *where*

$$N^R_{i,j} = \left\lceil \frac{\pi_{i,j}}{gcd(\{\pi_{m,n} \mid m \in I, n \geq 0, m \neq i\})} \right\rceil \tag{8}$$

for $I = \{i \mid 1 \leq i \leq n\}$.

If we denote the duration of one scheduler invocation by ξ, the total scheduler overhead for one period of an action $\alpha_{i,j}$ is $\delta_{i,j} = N_{i,j} \cdot \xi$. The total overhead is, therefore, made up of $N_{i,j}$ pieces of non-preemptable workload ξ. An important consequence of this demarcation is that the scheduler overhead only depends on the finitely many periods in the system and not on the load of the action.

We have determined two complementary methods to account for scheduler overhead, either by decreasing the speed at which processes run to maintain CPU utilization, or by increasing CPU utilization to maintain the speed at which processes run.

We call the first method response accounting and the second utilization accounting. Scheduler overhead accounting can, therefore, be done in two ways. One way is to allow an action to execute for less time than its actual limit within one period and use the remaining time to account for the scheduler overhead. The other way is to increase the limit so that the action can execute both its original limit and the time spent on scheduler invocations within one period.

We write that the overhead is

$$\delta_{i,j} = \delta^b_{i,j} + \delta^u_{i,j},$$

where $\delta^b_{i,j}$ is the overhead that extends the response-time bounds of the respective action and $\delta^u_{i,j}$ increases the utilization. Since ξ is not divisible, both $\delta^b_{i,j}$ and $\delta^u_{i,j}$ are multiples of ξ.

We differentiate three cases summarized in Table 3:

- Response accounting (RA), $\delta_{i,j} = \delta^b_{i,j}$, when the entire overhead is executing within the limit of the action, keeping both the limit and period (and thus the utilization) of the actions constant but increasing the response-time bounds.

Table 3 Scheduler overhead accounting [6]

Case	Overhead distribution	Load	Utilization	Schedulability test
RA	$\delta^b_{i,j} = \delta_{i,j}, \delta^u_{i,j} = 0$	$l^*_{i,j} = l_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j} - \delta_{i,j}} \right\rceil \delta_{i,j}$	$u^*_{i,j} = \frac{\lambda_{i,j}}{\pi_{i,j}}$	$\sum_{i \in I} \max_{j \geq 0} \frac{\lambda_{i,j}}{\pi_{i,j}} \leq 1$
UA	$\delta^b_{i,j} = 0, \delta^u_{i,j} = \delta_{i,j}$	$l^*_{i,j} = l_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \delta_{i,j}$	$u^*_{i,j} = \frac{\lambda_{i,j} + \delta_{i,j}}{\pi_{i,j}}$	$\sum_{i \in I} \max_{j \geq 0} \frac{\lambda_{i,j} + \delta_{i,j}}{\pi_{i,j}} \leq 1$
RUA	$\delta^b_{i,j}, \delta^u_{i,j} > 0$	$l'_{i,j} = l_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j} - \delta^b_{i,j}} \right\rceil \delta^b_{i,j}, l^*_{i,j} = l'_{i,j} + \left\lceil \frac{l'_{i,j}}{\lambda_{i,j}} \right\rceil \delta^u_{i,j}$	$u^*_{i,j} = \frac{\lambda_{i,j} + \delta^u_{i,j}}{\pi_{i,j}}$	$\sum_{i \in I} \max_{j \geq 0} \frac{\lambda_{i,j} + \delta^u_{i,j}}{\pi_{i,j}} \leq 1$

- Utilization accounting (UA), $\delta_{i,j} = \delta^u_{i,j}$, when the entire overhead increases the limit of the action, and thus the utilization, but the response-time bounds remain the same.
- Combined accounting (RUA), with $\delta_{i,j} = \delta^b_{i,j} + \delta^u_{i,j}$, $\delta^b_{i,j} > 0$, and $\delta^u_{i,j} > 0$, which offers the possibility to trade-off utilization for response time, for each action, in the presence of scheduler overhead.

For an action $\alpha_{i,j}$, in the presence of overhead, we denote the new load by $l^*_{i,j}$, the new limit by $\lambda^*_{i,j}$, and the new utilization by $u^*_{i,j}$. Using these new parameters and the old response-time bounds, defined in Eqs. (1) and (2), we determine the new upper and lower response-time bounds which we denote with $b^{u*}_{i,j}$ and $b^{l*}_{i,j}$, respectively. The upper response-time bound $b^{u*}_{i,j}$ for action $\alpha_{i,j}$ is

$$b^{u*}_{i,j} = \left\lceil \frac{l^*_{i,j}}{\lambda^*_{i,j}} \right\rceil \pi_{i,j} + \pi_{i,j} - 1. \tag{9}$$

The lower response-time bound $b^{l*}_{i,j}$ for $\alpha_{i,j}$ using the late release strategy is

$$b^{l*}_{i,j} = \left\lceil \frac{l^*_{i,j}}{\lambda^*_{i,j}} \right\rceil \pi_{i,j}, \tag{10}$$

whereas using the early release strategy is

$$b^{l*}_{i,j} = \left\lfloor \frac{l^*_{i,j}}{\lambda^*_{i,j}} \right\rfloor \pi_{i,j}. \tag{11}$$

We now give a condensed version of the three cases. The proofs of the results and a more detailed study can be found in [6].

6.1 Response accounting

In the response accounting case an action will have the same utilization but its response-time bounds increase. We have that $\delta_{i,j} = \delta^b_{i,j}$.

We compute the new load of the action $\alpha_{i,j}$ as

$$l^*_{i,j} = l_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j} - \delta_{i,j}} \right\rceil \delta_{i,j}.$$

The scheduler overhead that an action experiences during one period has to be smaller than its limit, otherwise the action will not execute any real workload. The new limit and utilization of the action are the same as without overhead, i.e., $\lambda^*_{i,j} = \lambda_{i,j}$ and $u^*_{i,j} = u_{i,j} = \frac{\lambda_{i,j}}{\pi_{i,j}}$. Since $l^*_{i,j} > l_{i,j}$ and $\lambda^*_{i,j} = \lambda_{i,j}$, we get that both the upper and the lower response-time bounds increase in case of response accounting.

Proposition 2 [6] *Let $\mathcal{P} = \{P_i(u_i) \mid 1 \leq i \leq n\}$ be a set of VBS processes each with bandwidth cap u_i. If $\sum_{i=1}^{n} u_i \leq 1$ and $\delta_{i,j} < \lambda_{i,j}$, with $\delta_{i,j}, \lambda_{i,j}$ as defined ABOVE, then the set of processes are schedulable with respect to the new response-time bounds $b^{u*}_{i,j}$ and $b^{l*}_{i,j}$, in the presence of worst-case scheduler overhead.*

The jitter for any action $\alpha_{i,j}$ in the response accounting case is at most $b^{u*}_{i,j} - b^{l}_{i,j}$.

For further reference, we write the new load in the response accounting case as a function

$$\mathrm{RA}(l, \lambda, \delta) = l + \left\lceil \frac{l}{\lambda - \delta} \right\rceil \delta.$$

6.2 Utilization accounting

In the utilization accounting case, an action is allowed to execute for more time than its original limit within a period and hence its utilization will increase (since the period duration remains the same). We have that $\delta_{i,j} = \delta^u_{i,j}$. The new load of action $\alpha_{i,j}$ becomes

$$l^*_{i,j} = l_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j}} \right\rceil \delta_{i,j}.$$

The new limit is $\lambda^*_{i,j} = \lambda_{i,j} + \delta_{i,j}$, and the new utilization is

$$u^*_{i,j} = \frac{\lambda_{i,j} + \delta_{i,j}}{\pi_{i,j}}.$$

Proposition 3 [6] *Given a set of processes $\mathcal{P} = \{P_i(u_i) \mid 1 \leq i \leq n\}$, let*

$$u^*_i = \max_{j \geq 0} \frac{\lambda_{i,j} + \delta_{i,j}}{\pi_{i,j}}.$$

If $\sum_{i=1}^{n} u_i^* \leq 1$*, then the set of processes* $\mathcal{P}$ *is schedulable with respect to the original response-time bounds* $b_{i,j}^u$ *and* $b_{i,j}^l$ *defined in Sect. 2.3, in the presence of worst-case scheduler overhead.*

Since the response-time bounds do not change in the utilization accounting case, the jitter for any action is the same as in the analysis without overhead.

We write the new load again as a function

$$\mathrm{UA}(l, \lambda, \delta) = l + \left\lceil \frac{l}{\lambda} \right\rceil \delta.$$

6.3 Combined accounting

The combined accounting case allows the total scheduler overhead to be accounted for both in the response-time bounds and in the utilization of an action. This allows a system designer, e.g., to increase the utilization of the actions up to available CPU bandwidth and account the rest of the scheduler overhead in the response-time bounds, thus only delaying the finishing of action by the smallest possible amount.

We have that $\delta_{i,j} = \delta_{i,j}^b + \delta_{i,j}^u$, $\delta_{i,j}^b > 0$, and $\delta_{i,j}^u > 0$. Given an action $\alpha_{i,j}$ with its associated virtual periodic resource $R_{i,j} = (\lambda_{i,j}, \pi_{i,j})$, and load $l_{i,j}$, the new load $l_{i,j}^*$ is computed in two steps. First, we account for the overhead that increases the response time

$$l'_{i,j} = l_{i,j} + \left\lceil \frac{l_{i,j}}{\lambda_{i,j} - \delta_{i,j}^b} \right\rceil \delta_{i,j}^b$$

and then we add the overhead that increases the utilization

$$l_{i,j}^* = l'_{i,j} + \left\lceil \frac{l'_{i,j}}{\lambda_{i,j}} \right\rceil \delta_{i,j}^u.$$

The load function for the combined case is, therefore,

$$\mathrm{RUA}(l, \lambda, \delta^b, \delta^u) = \mathrm{UA}(\mathrm{RA}(l, \lambda, \delta^b), \lambda, \delta^u).$$

The new limit for action $\alpha_{i,j}$ is $\lambda_{i,j}^* = \lambda_{i,j} + \delta_{i,j}^u$, and the utilization becomes

$$u_{i,j}^* = \frac{\lambda_{i,j} + \delta_{i,j}^u}{\pi_{i,j}}.$$

The upper response-time bound $b_{i,j}^{u*}$ for action $\alpha_{i,j}$ is now

$$b_{i,j}^{u*} = \left\lceil \frac{\mathrm{RUA}(l_{i,j}, \lambda_{i,j}, \delta_{i,j}^b, \delta_{i,j}^u)}{\lambda_{i,j} + \delta_{i,j}^u} \right\rceil \pi_{i,j} + \pi_{i,j} - 1.$$

The lower response-time bound $b_{i,j}^{l*}$ for the same action using the late release strategy is

$$b_{i,j}^{l*} = \left\lceil \frac{\mathrm{RUA}(l_{i,j}, \lambda_{i,j}, \delta_{i,j}^b, \delta_{i,j}^u)}{\lambda_{i,j} + \delta_{i,j}^u} \right\rceil \pi_{i,j},$$

and using the early release strategy is

$$b_{i,j}^{l*} = \left\lfloor \frac{\mathrm{RUA}(l_{i,j}, \lambda_{i,j}, \delta_{i,j}^b, \delta_{i,j}^u)}{\lambda_{i,j} + \delta_{i,j}^u} \right\rfloor \pi_{i,j}.$$

Proposition 4 [6] *Given a set of processes* $\mathcal{P} = \{P_i(u_i) \mid 1 \leq i \leq n\}$*, let*

$$u_i^* = \max_{j \geq 0} \frac{\lambda_{i,j} + \delta_{i,j}^u}{\pi_{i,j}}.$$

If $\sum_{i=1}^{n} u_i^* \leq 1$*, then the set of processes* $\mathcal{P}$ *is schedulable with respect to the response-time bounds* $b_{i,j}^{u*}$ *and* $b_{i,j}^{l*}$*, in the presence of worst-case scheduler overhead.*

6.4 Optimizations

In [6], we also discuss optimizations to the combined and utilization accounting cases presented before. The optimization possibility comes from the over-approximation of the estimate of the number of scheduler invocations during an action period. During one period of an action, any other process can have a release triggering a scheduler invocation. In the above cases, these invocations are accounted for in the response-time bounds or utilization of that action. However, the invocations due to the same release are incorporated as overhead for more than one action, meaning that scheduler overhead is sometimes accounted for more than necessary.

In order to separate the reasons for scheduler invocations, we observe a natural division of the VBS scheduler overhead into overhead due to releasing and due to suspending processes. Scheduler invocations that occur due to releasing of processes can be improved by accounting for them in a separate, virtual VBS process instead of the given VBS processes. We call this process the scheduler process. The scheduler process is then accounted for in increased overall CPU utilization. The remaining overhead due to suspending processes may then be accounted for using the methods described above.

A sufficient condition for improvement of the estimate is that there exists a process P_m that accounts, in the original estimate, for as many scheduler invocations as the scheduler process, i.e., $gcd(\{\pi_{i,j} \mid i \in I, j \geq 0\})$ equals $gcd(\{\pi_{i,j} \mid i \in I, j \geq 0, i \neq m\})$.

We denote the scheduler process by P_S with all actions equal $\alpha_{S,j} = (\xi, R_S)$ where $R_S = (\lambda_S, \pi_S) = (\xi, gcd(\{\pi_{i,j} \mid i \in I, j \geq 0\})$. Thus, the utilization of the scheduler process is

$$u_S = \frac{\xi}{gcd(\{\pi_{i,j} \mid i \in I, j \geq 0\})}.$$

In the combined accounting case an optimization is possible only in the case $\delta_{i,j}^u = N_{i,j}^R \cdot \xi$ and $\delta_{i,j}^b = \xi$. In the utilization accounting case the limit of the action $\alpha_{i,j}$

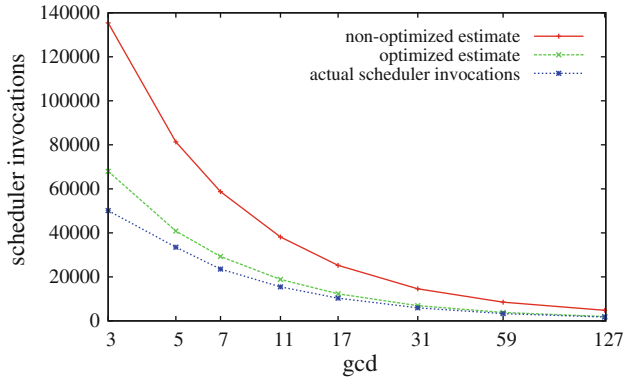


Fig. 6 Accuracy of the estimate on scheduler invocations [6]

becomes $\lambda^*_{i,j} = \lambda_{i,j} + \xi$ and, therefore, the utilization is

$$u^*_{i,j} = \frac{\lambda_{i,j} + \xi}{\pi_{i,j}}.$$

An important property of the scheduler process is that the period of its action is smaller than or equal to any other period in the system meaning that it can always execute at the appropriate time and, therefore, does not change the schedulability result.

In order to see the result of the optimization on the scheduler invocation estimates we conducted a simulation experiment (Fig. 6) in [6] showing a comparison of the global optimized and non-optimized estimates, and the total number of actual scheduler invocations measured for three concurrently running actions over 100000 time units. The periods of the actions are chosen so that they result in an increasing *gcd* (logarithmic x axis). The periods are actually multiples of the *gcd* by 2, 3, and 5. For small values of the *gcd* the accuracy of the optimized estimate is considerably better than the accuracy of the non-optimized estimate. For large values of the *gcd*, the estimates converge.

7 Power-aware VBS

An important non-functional aspect of real-time systems is power consumption. Many modern processors contain mechanisms that enable dynamic voltage and frequency scaling (DVS) [18,19]. In the area of real-time systems, power-aware scheduling mechanisms make use of DVS in order to allow processes to maintain their real-time properties while reducing the overall CPU power consumption. Power-aware real-time scheduling, e.g., for EDF and rate-monotonic, has been extensively studied [19–22]. Similarly, there has also been research concerning power-aware server mechanisms [23,24]. We have introduced and discussed a power-aware version of VBS in [7]. Here, we briefly present some of our findings in order to give a complete picture of VBS scheduling.

In a simplified CPU power model [19], the consumed CPU power P is proportional to the operating frequency f and the square of the voltage level V, i.e., $P \propto f \cdot V^2$ [18]. Since a reduced voltage imposes a maximal available frequency level and reducing the operating frequency extends the execution time of a workload [19], the scheduling mechanism has to find the minimal possible frequency at which the processes still meet their deadlines.

A reduction in the operating frequency is only possible when there is so-called slack in the system, i.e., when the scheduled processes do not use 100 % of the CPU bandwidth.

Since the VBS process model is different from the EDF or CBS process model, in [7], we distinguish two types of slack specific to VBS systems, namely static and dynamic VBS slack. The static slack results from the predefined bandwidth caps of the VBS processes whereas the dynamic slack results from the individual actions of each VBS process. Furthermore, we divide the dynamic slack in action and termination slack. We next briefly describe the implications on power consumption of each type of slack.

Static slack In the VBS process model, each process P_i has a bandwidth cap u_i which represents the maximum utilization that any of its actions may have. It has been shown in [7,19] that if the total utilization of a set of EDF processes is $U \in [0, 1]$, the frequency f can be computed as $f = U \cdot f_{\max}$, where $f_{\max}$ is the maximal available frequency, such that no process will violate its deadline. As a consequence, if the sum of all VBS bandwidth caps is less than 1, we can safely scale down the processor frequency to f without any consequence on the response-time bounds of actions. The computed frequency f is set once, before the system runs, and is not modified during runtime.

Dynamic slack Due to the action model of VBS, sometimes slack arises during runtime such that at certain points in time the frequency can be scaled lower than the aforementioned computed value. The important aspect of these dynamic frequency scalings is that we have to make sure that we adapt the frequency to the currently available slack so that no action will miss deadlines and the response-time bounds of actions remain unchanged.

We distinguish two types of dynamic slack.

- *Termination slack*, resulting from the VBS termination strategy.
- *Action slack*, generated by an action having utilization less than the bandwidth cap of the process.

The termination strategy, explained in Sect. 2.2 postpones the logical termination of an action to the end of the last period of that action. We can generate slack at runtime by computing at every arrival of a new action the lowest

possible limit such that the action still finishes its load within the original number of periods, i.e., such that the original response-time bound is maintained.

For example, an action $\alpha = (55, (30, 100))$ could run for 28 time units every period and still meet its response-time bound 200. Therefore, the virtual periodic resource for this action can be changed from $(30, 100)$ to $(28, 100)$ and the resulting slack of 2 time units per period can be used to scale down the processor.

More formally, at every arrival time t of an action, a new limit is computed as follows:

$$\lambda^*_{i,j} = \left\lceil \frac{l_{i,j}}{n_{i,j}} \right\rceil,$$

where $n_{i,j} = \lceil \frac{l_{i,j}}{\lambda_{i,j}} \rceil$ is the number of periods needed for the action $\alpha_{i,j}$ to finish its load. Note that the new limit never exceeds the old limit ($\lambda^*_{i,j} \leq \lambda_{i,j}$), so this change does not influence schedulability. The action $\alpha_{i,j}$ will thus be transformed into an action $\alpha^*_{i,j} = (l_{i,j}, (\lambda^*_{i,j}, \pi_{i,j}))$.

The second type of dynamic slack is the action slack which is a result of newly arriving actions for a VBS process. Dynamic slack is generated when the utilization of the new action is lower than the bandwidth cap for the process. In [7], we show that it is possible to scale the frequency at runtime by a new scaling factor computed as the sum of remaining utilizations of the active actions (the current actions of each VBS process). The computation and scaling of the frequency happen at every time instant when an action has a release.

Proposition 5 [7] *Let $\mathcal{P} = \{P_i(u_i) \mid 1 \leq i \leq n\}$ be a schedulable set of VBS processes, with a total utilization cap $U = \sum_{i=1}^{n} u_i \leq 1$ and corresponding process actions $\alpha_{i,j}$, with virtual periodic resources $(\lambda_{i,j}, \pi_{i,j})$, for $j \geq 0$, of process P_i. This set of processes is schedulable within the response-time bounds if in between two action releases the processor frequency is at least $f_{new} = U_c \cdot f_{\max}$ where $U_c = \sum_{i=1}^{n} \frac{\lambda_{i,j_i}}{\pi_{i,j_i}}$ is the total utilization of all released actions α_{i,j_i} in the considered interval of time between two action releases.*

Since the two types of dynamic slack address separate aspects of VBS process execution, they can be exploited separately or together. Using only the termination slack may reduce the actual response-time jitter of the action, while using only the action slack does not modify the original limit of the action. If both types of dynamic slack are exploited, the minimum possible operating frequency is achieved and CPU utilization is maximized.

We present an experiment (Fig. 7) where we show the power savings using action slack alone, and action slack combined with termination slack for 10 sets of randomly-generate simulated VBS-processes. The left y axis shows the normalized power consumption while the right y axis shows the CPU idle percentage. As expected, we see that higher CPU idle time results in lower power consumption and that the combined method results in the lowest power consumption as opposed to the unscaled system and the system scaled using the action slack only.

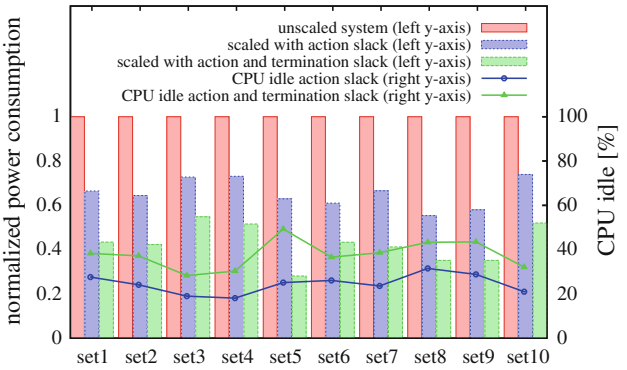


Fig. 7 Normalized power consumption for 10 sets of random processes [7]

The presented methods for reducing power consumption based on static and dynamic slack adhere strictly to the VBS process model described in Sect. 2. Diverging from the strict model and allowing the scheduler to redistribute computation time of process actions among the server periods during which the actions execute, results in even lower power consumption without affecting the actions' original response-time bounds. In order to be able to redistribute computation time between periods the power-aware VBS scheduler must be aware of the future, i.e., it must know the sequence of action changes of every VBS process in advance.

Reducing power consumption using future knowledge depends on the specific power model of the CPU (represented through a power consumption function) as well as the frequency switching overhead. In [7], we (a subset of the authors) present an optimal offline algorithm that computes the best possible configuration of server bandwidths for every period of an action during the whole lifetime of a system thus minimizing the given power-consumption function. We show that it is also possible to incorporate frequency switching overhead into the computation. Since the offline method may not be feasible for a real system, we also give an approximate to the optimal offline method using a feasible online algorithm. Given a simplified power consumption model, we show (cf. [7]) that it is possible to approximate the optimal offline results by decreasing CPU utilization jitter, i.e., the actual CPU utilization is steered towards a computed average.

8 Related work

There is a significant body of research concerning temporal isolation. We first present several mechanisms that enable

some form of temporal isolation for resources like CPU, disk, and I/O.

One of the the first models proposed is the fair queueing model in the context of communication networks [25,26]. In a simplified version, each communication source is serviced in a dedicated queue of infinite storage size and the queues are handled in round-robin fashion. Isolation is provided in the sense that a source will not affect other sources if it receives more requests than specified.

The generalized processor sharing (GPS) approach [27] also allows communication sources to be isolated in their temporal behavior under the assumption that the network traffic is infinitely divisible. Both [27] and [26] describe approximations of the GPS algorithm that are viable for real systems. The methods described above are similar to proportional share allocation (PSA) introduced in [28,29] which allows weighted sharing of computing resources in isolation. Each shared resource is discretized in quanta of size q and each process is assigned a fraction of the resources through a predefined weight. The deviation of this model from the ideal one is bounded by q [29].

In the context of CPU scheduling for real-time systems, the concept of CPU capacity reserves was introduced in [30,31] for both real-time and non-real-time processes. Virtual periodic resources [8] are similar to CPU capacity reserves but are used to describe periodic resource allocation used in compositional timing analysis. We describe a more general form of resource reservations that are not restricted to the CPU resource. The resource reservation concept is extended in [32] to include other system resources with a focus on QoS guarantees. The model in [32] is similar to ours, but uses resource fractions instead of limits and periods. As a consequence, there are no process actions and hence no deadlines that would enable EDF [2] scheduling. A similar model is found in the Rialto [33,34] system but lacks the concept of sequential process actions from the VBS model. SMART [35] is another related scheduling model intended for adaptive multimedia and soft real-time processes. Similar to our approach, SMART allows processes to vary their alloted reservation, but is not designed to support hard deadlines.

Server mechanisms [3] are another form of resource reservations. A server is usually defined by a server capacity or limit (C_S) and a server period (T_S) [13]. A process that is encapsulated within a server will execute at most C_S time units in a time window of T_S time units. A large variety of server mechanisms have been introduced, e.g., [3,11,12,36–38].

The constant-utilization server (CUS) [37,39] and the total-bandwidth server (TBS) [38] are very similar to VBS but do not have the ability to change the server limit and period at runtime. There is no notion of sequentiality within a process, i.e., there is no counterpart of our action model.

The work on CBSs [3] is highly related to ours, as already elaborated in the previous sections. Similar to CBS, VBS also uses an EDF-based algorithm for scheduling. The drawback of a CBS, like with CUS and TBS, is that its resource's limit and period cannot be changed. As elaborated before, a process may sometimes need to execute a small portion of its code with lower latency than the rest of its code and, therefore, temporarily require a shorter period [4,40]. We next present several related mechanisms that allow such a change in the rate of execution within the process.

RBED [41,42] is a rate-based scheduler that involves a modified EDF process model which allows the bandwidth (limit) and rate (period) of processes to be changed at runtime. RBED is most closely related to VBS as it also incorporates dynamic adjustments in process parameters. RBED and VBS differ on the level of abstraction: in VBS, processes are modeled as sequences of actions to quantify the response times of portions of process code where each transition from one action to the next offers the possibility of parameter adjustment. RBED provides ranges of feasible reconfiguration but does not specify any higher-level mechanism for changing the period or limit of processes. Moreover, the reconfiguration is done within the limits of the currently available system utilization making an offline analysis difficult.

Elastic scheduling [43,44] introduces a new process model in conjunction with EDF, which views process utilization as a spring with a certain elasticity and maximum length configuration. Given the configuration, processes can change both limit and period within the resulting constraints. In [45], CBS are dynamically reconfigured using a benefit function through genetic algorithms. The goal of this and similar approaches, such as [46], is mainly to handle reconfiguration of processes and the potentially ensuing overloads in a more robust manner by performing adaptations based on the current system utilization. The VBS model offers flexibility by defining processes whose throughput and latency change at runtime through individual actions and, therefore, differs in implementation and goal from the mentioned approaches.

In [47], the authors discuss dynamic reconfiguration of servers based on TDMA that enables a change in both limit and period of the TDMA partitioning allocation. The authors present schedulability analyses and algorithms for all possible reconfiguration cases.

In the context of virtualization research, XEN [48] employs three schedulers which were compared and discussed in [49]: borrowed virtual time [50], a weighted proportional-share scheduler; SEDF [51], a modified version of EDF; and a credit scheduler that allows automatic load balancing. Another scheduling method for XEN, which gives more attention to I/O-intensive domains, was presented in [52].

Other areas of related work that enable either temporal isolation or dynamic reconfiguration of systems are strongly partitioned systems (e.g., [39,53–55]) and mode switches in real-time systems (cf. [56]). Our system can be seen as scheduling partitions, namely processes correspond to partitions and inside a partition actions are sequentially released processes. Similarly, on an abstract level, actions of VBS processes can be interpreted as different modes of the same process and the VBS mechanism as providing safe mode changes. The scheduling goal, process model, and methods, however, are in both cases different from the mentioned areas of research.

Finally, we compare the complexity of our queue management plugins and scheduler to other work. By n we denote the number of processes. The SMART [35] scheduler's time complexity is determined by the complexity of special list operations and the complexity of schedule operations. The list operations complexity is $O(\log(n))$ if tree data structures are used and $O(n)$ otherwise. The schedule complexity is $O(n_R^2)$, where n_R is the number of active real-time processes with higher priority than non-real-time processes. The authors in [35] point out that, in special cases, the complexity can be reduced to $O(n)$ and even $O(1)$. The Move-to-Rear List scheduling algorithm [32] has $O(log(n))$ complexity. A recent study [57] on EDF-scheduled systems describes an implementation mechanism for queue operations using deadline wheels where the resulting time and space complexities are similar to those achieved by our four queue mechanisms.

9 Conclusions

We have presented a comprehensive study of theoretical and practical aspects of VBSs [1]. In particular, we focused on the VBS process model and schedulability results with detailed proofs. Furthermore, we have presented implementation details and experimental results involving four queue management plugins that allow trading-off time and space complexity of the VBS scheduler. We have also presented a survey of results on scheduler overhead accounting [6] and power-aware scheduling with VBS [7].

We now briefly discuss possible future work. On the practical side, it is certainly interesting to see VBS in action, i.e., scheduling realistic case studies (beyond synthetic benchmarks). Another practical issue would be to exploit even further the possible ways to efficiently implement the queues, e.g., using (binary) heap(s) [58].

On the conceptual side, temporal isolation is just one type of process isolation. Another well-known type of process isolation that is relevant in virtually all computer systems including non-real-time systems is spatial isolation. Processes that are spatially isolated do not tamper with the memory regions of one another, except when explicitly allowed. A third, more recently studied type of process isolation is power isolation where the amount of power consumed by a process may be approximated independently of the power consumed by other processes [59], or individually controlled and capped by a power manager [60,61]. We envision a system that provides full temporal, spatial, and power isolation of software processes simultaneously [62]. We have taken the first step towards this goal by studying process isolation with respect to each property individually, through the VBS approach for temporal isolation, the compact-fit memory management system [63] for spatial isolation in real time, and the work on power isolation in EDF-scheduled systems [59]. The key insight is that there appears to be a fundamental trade-off between quality and cost of time, space, and power isolation. Designing an integrated system that supports temporal, spatial, and power isolation remains the key challenge in this context.

References

1. Craciunas, S.S., Kirsch, C.M., Payer, H., Röck, H., Sokolova, A.: Programmable temporal isolation through variable-bandwidth servers. In: Proceedings of SIES. IEEE, New York (2009)
2. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard real-time environment. J. ACM **20**(1), 46–61 (1973)
3. Abeni, L., Buttazzo, G.: Resource reservation in dynamic real-time systems. Real-Time Syst. **27**(2), 123–167 (2004)
4. Cervin, A.: Improved scheduling of control tasks. In: Proceedings of ECRTS. IEEE, New York (1999)
5. Craciunas, S.S., Kirsch, C.M., Röck, H., Sokolova, A.: Real-time scheduling for workload-oriented programming. Department of Computer Sciences, University of Salzburg, Technical Report 2008-02, September (2008)
6. Craciunas, S.S., Kirsch, C.M., Sokolova, A.: Response time versus utilization in scheduler overhead accounting. In: Proceedings of RTAS. IEEE, New York (2010)
7. Craciunas, S.S., Kirsch, C.M., Sokolova, A.: Power-aware temporal isolation with variable-bandwidth servers. In: Proceedings of EMSOFT. ACM, New York (2010)
8. Shin, I., Lee, I.: Periodic resource model for compositional real-time guarantees. In: Proceedings of RTSS. IEEE, New York (2003)
9. Shin, I., Lee, I.: Compositional real-time scheduling framework. In: Proceedings of RTSS. IEEE, New York (2004)
10. Harbour, M.G., Klein, M.H., Lehoczky, J.P.: Timing analysis for fixed-priority scheduling of hard real-time systems. IEEE Trans. Softw. Eng. **20**(1), 13–28 (1994)
11. Sprunt, B., Sha, L., Lehoczky, J.P.: Aperiodic task scheduling for hard-real-time systems. Real-Time Syst. **1**, 27–60 (1989)
12. Strosnider, J.K., Lehoczky, J.P., Sha, L.: The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments. IEEE Trans. Comput. **44**, 73–91 (1995)
13. Buttazzo, G.: Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications. Kluwer, Norwell (1997)
14. Craciunas, S.S., Kirsch, C.M., Payer, H., Röck, H., Sokolova, A., Stadler, H., Staudinger, R.: The Tiptoe system (2007). http://tiptoe.cs.uni-salzburg.at
15. Buttazzo, G., Gai, P.: Efficient implementation of an EDF scheduler for small embedded systems. In: Proceedings of OSPERT (2006)
16. Bayer, R., McCreight, E.M.: Organization and maintenance of large ordered indices. Acta Informatica **1**, 173–189 (1972)

17. Bacon, D.F., Cheng, P., Grove, D.: Tuningfork: a platform for visualization and analysis of complex real-time systems. In: Proceedings of Companion on OOPSLA. ACM, New York (2007)
18. Burd, T.D., Brodersen, R.W.: Energy efficient CMOS microprocessor design. In: Proceedings of HICSS. IEEE, New York (1995)
19. Pillai, P., Shin, K.G.: Real-time dynamic voltage scaling for low-power embedded operating systems. In: Proceedings of SOSP. ACM, New York (2001)
20. Aydin, H., Mejía-Alvarez, P., Mossé, D., Melhem, R.: Dynamic and aggressive scheduling techniques for power-aware real-time systems. In: Proceedings of RTSS. IEEE, New York (2001)
21. Qadi, A., Goddard, S., Farritor, S.: A dynamic voltage scaling algorithm for sporadic tasks. In: Proceedings of RTSS. IEEE, New York (2003)
22. Shin, D., Kim, J.: Dynamic voltage scaling of periodic and aperiodic tasks in priority-driven systems. In: Proceedings of ASP-DAC. IEEE Press, New York (2004)
23. Lawitzky, M.P., Snowdon, D.C., Petters, S.M.: Integrating real time and power management in a real system. In: Proceedings of OSPERT (2008)
24. Scordino, C., Lipari, G.: Using resource reservation techniques for power-aware scheduling. In: Proceedings of EMSOFT. ACM, New York (2004)
25. Nagle, J.: On packet switches with infinite storage. IEEE Trans. Commun. **35**(4), 435–438 (1987)
26. Demers, A., Keshav, S., Shenker, S.: Analysis and simulation of a fair queueing algorithm. SIGCOMM Comput. Commun. Rev. **19**, 1–12 (1989)
27. Parekh A.K., Gallager R.G.: A generalized processor sharing approach to flow control in integrated services networks: the single-node case. IEEE/ACM Trans. Netw. **1**, 344–357 (1993)
28. Jeffay, K., Smith, F.D., Moorthy, A., Anderson, J.: Proportional share scheduling of operating system services for real-time applications. In: Proceedings of RTSS. IEEE Computer Society, New York (1998)
29. Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S.K., Gehrke, J.E., Plaxton, C.G.: A proportional share resource allocation algorithm for real-ti time-shared systems. In: Proceedings of RTSS. IEEE Computer Society, New York (1996)
30. Mercer, C.W., Savage, S., Tokuda, H.: Processor capacity reserves for multimedia operating systems. Carnegie Mellon University, Technical Report (1993)
31. Mercer, C.W., Savage, S., Tokuda, H.: Processor capacity reserves: Operating system support for multimedia applications. In: Proceedings of ICMCS (1994)
32. Bruno, J., Gabber, E., Özden, B., Silberschatz, A.: Move-to-rear list scheduling: a new scheduling algorithm for providing QoS guarantees. In Proceedings of MULTIMEDIA. ACM, New York (1997)
33. Jones, M., Leach, P., Draves, R., Barrera, J.: Modular real-time resource management in the Rialto operating system. In: Proceedings of HOTOS. IEEE, New York (1995)
34. Jones, M.B., Roşu, D., Roşu, C.: CPU reservations and time constraints: efficient, predictable scheduling of independent activities. In: Proceedings of SOSP. ACM, New York (1997)
35. Nieh, J., Lam, M.S.: The design, implementation and evaluation of SMART: a scheduler for multimedia applications. In: Proceedings of SOSP. ACM, New York (1997)
36. Lehoczky, J.P., Sha, L., Strosnider, J.K.: Enhanced aperiodic responsiveness in hard real-time environments. In: Proceedings of RTSS. IEEE, New York (1987)
37. Deng, Z., Liu, J.W.-S., Sun, S.: Dynamic scheduling of hard real-time applications in open system environment. University of Illinois at Urbana-Champaign, Technical Report (1996)
38. Spuri, M., Buttazzo, G.C.: Scheduling aperiodic tasks in dynamic priority systems. J. Real-Time Syst. **10**(2), 179–210 (1996)
39. Deng, Z., Liu, J.W.-S., Zhang, L., Mouna, S., Frei, A.: An open environment for real-time applications. Real-Time Syst. **16**(2–3), 155–185 (1999)
40. Cervin, A., Eker, J.: The Control Server: a computational model for real-time control tasks. In: Proceedings of ECRTS. IEEE, New York (2003)
41. Brandt, S.A., Banachowski, S., Lin, C., Bisson, T.: Dynamic integrated scheduling of hard real-time, soft real-time and non-real-time processes. In: Proceedings of RTSS. IEEE, New York (2003)
42. Goddard, S., Liu, X.: Scheduling aperiodic requests under the rate-based execution model. In: Proceedings of RTSS. IEEE, New York (2002)
43. Buttazzo, G., Abeni, L.: Adaptive workload management through elastic scheduling. Real-Time Syst. **23**(1–2), 7–24 (2002)
44. Buttazzo, G.C., Lipari, G., Abeni, L.: Elastic task model for adaptive rate control. In: Proceedings of RTSS. IEEE, New York (1998)
45. Simoes, M.A.C., Lima, G., Camponogara, E.: A GA-based approach to dynamic reconfiguration of real-time systems. In: Proceedings of APRES (2008)
46. Beccari, G., Reggiani, M., Zanichelli, F.: Rate modulation of soft real-time tasks in autonomous robot control systems. In: Proceedings of ECRTS (1999)
47. Stoimenov, N., Thiele, L., Santinelli, L., Buttazzo, G.: Resource adaptations with servers for hard real-time systems. In: Proceedings of EMSOFT. ACM, New York (2010)
48. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the art of virtualization. In: Proceedings of SOSP. ACM, New York (2003)
49. Cherkasova, L., Gupta, D., Vahdat, A.: Comparison of the three CPU schedulers in Xen. SIGMETRICS Perform. Eval. Rev. **35**(2), 42–51 (2007)
50. Duda, K.J., Cheriton, D.R.: Borrowed-virtual-time (BVT) scheduling: supporting latency-sensitive threads in a general-purpose scheduler. SIGOPS Oper. Syst. Rev. **33**(5), 261–276 (1999)
51. Leslie, I.M., Mcauley, D., Black, R., Roscoe, T., Barham, P.T., Evers, D., Fairbairns, R., Hyden, E.: The design and implementation of an operating system to support distributed multimedia applications. IEEE JSAC **14**(7), 1280–1297 (1996)
52. Govindan, S., Nath, A.R., Das, A., Urgaonkar, B., Sivasubramaniam, A.: Xen and co.: Communication-aware cpu scheduling for consolidated Xen-based hosting platforms. In: Proceedings of VEE. ACM, New York (2007)
53. Kim, D., Lee, Y.-H., Younis, M.: SPIRIT-μKernel for strongly partitioned real-time systems. In: Proceedings of RTCSA. IEEE, New York (2000)
54. Kim, D., Lee, Y.-H.: Periodic and aperiodic task scheduling in strongly partitioned integrated real-time systems. Comput. J. **45**(4), 395–409 (2002)
55. Lipari, G., Bini, E.: A methodology for designing hierarchical scheduling systems. J. Embedded Comput. **1**(2), 257–269 (2005)
56. Real, J., Crespo, A.: Mode change protocols for real-time systems: a survey and a new proposal. Real-Time Syst. **26**, 161–197 (2004)
57. Short, M.: Improved task management techniques for enforcing EDF scheduling on recurring tasks. In: Proceedings of RTAS. IEEE, New York (2010)
58. Stein, C., Cormen, T., Rivest, R., Leiserson, C.: Introduction To Algorithms. MIT Press, Cambridge (2001)
59. Craciunas, S.S., Kirsch, C.M., Sokolova, A.: The power of isolation. Department of Computer Sciences, University of Salzburg, Technical Report 2011-02 July (2011)
60. Liu, X., Shenoy, P., Corner, M.: Chameleon: application level power management with performance isolation. In: Proceedings of MULTIMEDIA. ACM, New York (2005)
61. Cao, Q., Fesehaye, D., Pham, N., Sarwar, Y., Abdelzaher, T.: Virtual battery: an energy reserve abstraction for embedded sensor

networks. In: Proceedings of RTSS, pp. 123–133. IEEE Computer Society, New York (2008)
62. Craciunas, S.S., Haas, A., Kirsch, C.M., Payer, H., Röck, H., Rottmann, A., Sokolova, A., Trummer, R., Love, J., Sengupta, R.: Information-acquisition-as-a-service for cyber-physical cloud computing. In: Proceedings of HotCloud. USENIX, Boston (2010)
63. Craciunas, S.S., Kirsch, C.M., Payer, H., Sokolova, A., Stadler, H., Staudinger, R.: A compacting real-time memory management system. In: Proceedings of USENIX ATC. USENIX, Boston (2008)

Journal of Computer and System Sciences 81 (2015) 859–879

Contents lists available at ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Trace semantics via determinization ☆

Bart Jacobs [a], Alexandra Silva [a,*,1], Ana Sokolova [b]

[a] Institute for Computing and Information Sciences, Radboud University Nijmegen, Netherlands
[b] Department of Computer Sciences, University of Salzburg, Austria

ARTICLE INFO

Article history:
Received 23 January 2013
Accepted 15 May 2014
Available online 5 December 2014

Keywords:
Coalgebra
Kleisli category
Eilenberg–Moore category
Trace semantics

ABSTRACT

This paper takes a fresh look at the topic of trace semantics in the theory of coalgebras. In the last few years, two approaches, somewhat incomparable at first sight, captured successfully in a coalgebraic setting trace semantics for various types of transition systems. The first development of coalgebraic trace semantics used final coalgebras in Kleisli categories and required some non-trivial assumptions, which do not always hold, even in cases where one can reasonably speak of traces (like for weighted automata). The second development stemmed from the observation that trace semantics can also arise by performing a determinization construction and used final coalgebras in Eilenberg–Moore categories. In this paper, we develop a systematic study in which the two approaches can be studied and compared. Notably, we show that the two different views on trace semantics are equivalent, in the examples where both approaches are applicable.

1. Introduction

Studying the semantics of state-based systems has been a long quest in Theoretical Computer Science. Central to this study is the search for the correct notion of equivalence and proof methods thereof. Coalgebras provide an abstract framework for state-based computation, where a canonical notion of equivalence can be uniformly derived from the type (formally, a functor) of the system under study. The canonical notion of equivalence is adequate in many situations, but turns out to be too fine when the type of the system includes certain computational effects that are intended to be hidden from the observer. To overcome this shortcoming of the general theory of coalgebras, notions of equivalence in which the type of the system can be split into relevant type information and computational effects to be hidden were devised. The coarser equivalences were coined under the generic name of *coalgebraic (decorated) trace semantics* [18,36,37,6].

In this paper, we take a fresh look at the topic of coalgebraic trace semantics and provide a framework where the existing two main approaches, which we explain next, can be studied and compared. Throughout the rest of the paper we assume a certain familiarity with basic notions of category theory, such as functors, natural transformations and distributive laws, as well as with the definitions of Kleisli and Eilenberg–Moore categories. Readers unfamiliar with these notions can find basic material on these in standard category theory books, see e.g. [2].

☆ Extended version of [23].

* Corresponding author.
E-mail addresses: bart@cs.ru.nl (B. Jacobs), alexandra@cs.ru.nl (A. Silva), anas@cs.uni-salzburg.at (A. Sokolova).

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

http://dx.doi.org/10.1016/j.jcss.2014.12.005
0022-0000/

A coalgebra is a map of the form $X \to H(X)$, where X is a state space and H is a functor that captures the kind of computation involved. Often, this H is, or contains, a monad T, providing certain computational effects, such as when T is lift $1 + (-)$, powerset $\mathcal{P}$ or distribution $\mathcal{D}$, giving partial, non-deterministic or probabilistic computation.

In the case such an H contains a monad T, it turns out that there are two archetypical forms in which the monad T plays a role, namely in:

$$X \longrightarrow G(TX) \qquad \text{or} \qquad X \longrightarrow T(FX) \tag{1}$$

In the first case, the monad T occurs inside a functor G that typically handles input and output. In the second case the monad T is on the outside, encapsulating a form of computation over a functor F, that typically describes the transitions involved. In some cases these two forms are equivalent, for instance for non-deterministic automata with labels—given by a set A—and termination. They involve the powerset monad $\mathcal{P}$ and can be described equivalently as:

$$X \longrightarrow 2 \times (\mathcal{P}X)^A \qquad \text{or} \qquad X \longrightarrow \mathcal{P}(1 + A \times X)$$

where on the left-hand-side we have the functor $G = 2 \times (-)^A$ and on the right-hand-side $F = 1 + A \times (-)$. These descriptions are equivalent because the powerset monad $\mathcal{P}$ is special (it is "additive", see [12], mapping coproducts to products). In fact, there are isomorphisms:

$$\mathcal{P}(1 + A \times X) \cong \mathcal{P}(1) \times \mathcal{P}(A \times X) \cong 2 \times \big(\mathcal{P}(X)\big)^A. \tag{2}$$

Classically, to study language or trace equivalence for non-deterministic automata there is a standard construction called determinization [20]. It involves changing the state space X into a state space $\mathcal{P}(X)$. In doing so, the transition structure becomes much simpler (in particular, deterministic).

In this paper, we are interested in a similar determinization construction, but on a more abstract level. It involves changing the state space from X into $T(X)$, for a monad T. It turns out that this can be done relatively easily for coalgebras of the form $X \to G(TX)$, on the left in (1), as illustrated in [36]: it involves a distributive law between G and T, and such law corresponds to a lifting $\widehat{G}$ of the functor G to the category $\mathcal{EM}(T)$ of Eilenberg–Moore algebras of T. Moreover, the final G-coalgebra lifts to a final $\widehat{G}$-coalgebra in $\mathcal{EM}(T)$. In this way, one obtains final coalgebra semantics in a category of algebras. It yields a first form of "$\mathcal{EM}$" extension semantics, see Definition 1. Categorically, this extension $X \mapsto T(X)$ is given by the free algebra functor $\mathbf{C} \to \mathcal{EM}(T)$.

Extension for coalgebras of the form $X \to T(FX)$ on the right in (1) is more complicated; it involves the comparison functor $\mathcal{K}\ell(T) \to \mathcal{EM}(T)$. We proceed by first translating these coalgebras to coalgebras of the lifted functor $\widehat{G}$ via a suitable law $\mathfrak{e}\colon TF \Rightarrow GT$, see Section 6 (and [3]). It will be shown that the resulting trace semantics, in categories of algebras, as sketched above,

- not only includes the trace semantics in Kleisli categories developed in [18];
- but also covers more examples; in particular, it covers trace semantics for weighted automata $X \to \mathcal{M}(1 + A \times X)$, involving the multiset monad $\mathcal{M}$. This monad does not fit in the trace framework of [18] because its Kleisli category is not dcpo-enriched.

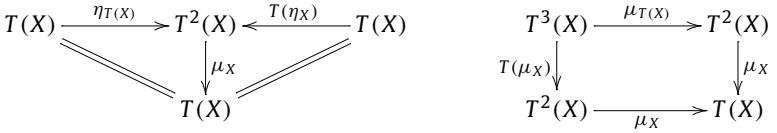
The technical (categorical) core of the paper concentrates on lifting the comparison functor $\mathcal{K}\ell(T) \to \mathcal{EM}(T)$ to categories of coalgebras $\mathbf{CoAlg}(\widehat{F}) \to \mathbf{CoAlg}(\widehat{G})$, of lifted functors $\widehat{F}$, $\widehat{G}$. We specialize this framework by taking G to be the functor $B \times (-)^A$ for deterministic automata. Its final coalgebra $B^{A^\star}$ gives trace semantics. Our framework is general enough to allow a different semantics, like "tree" semantics, by using a different functor G, as we illustrate in other examples.

The paper is organized as follows. The first section below recalls the basics about monads and the associated Kleisli category and category of (Eilenberg–Moore) algebras. Section 3 gives a systematic account of liftings of functors to such (Kleisli and algebra) categories, and of distributive laws; it includes a lifting result for final coalgebras to categories of Eilenberg–Moore algebras. Subsequently, Section 4 briefly recalls the coalgebraic description of deterministic automata, their final coalgebras, and the lifting of these final coalgebras to categories of Eilenberg–Moore algebras; it also includes a description of such final coalgebras obtained via a lifting of an adjunction $\mathbf{Sets}^{\mathrm{op}} \leftrightarrows \mathcal{EM}(T)$, in the style of [34]. Section 5 contains examples of the application of $\mathcal{EM}$-extension semantics: for classical non-deterministic automata and a new example mixing probability and non-determinism, for simple Segala systems, applicable also to alternating automata and non-deterministic weighted automata, as well as to pushdown automata. In Section 6, we develop an extension semantics for coalgebras of type TF (Definition 2, via Theorem 2) and show that the Kleisli trace semantics from [18] fits in the current setting (Proposition 5). We also summarize and relate all relevant categories and functors (Theorem 3). Section 7 presents examples of the $\mathcal{K}\ell$-extension semantics, including examples already treated in [18] and, more notably, examples that cannot be studied in the framework of [18]. Section 8 contains concluding remarks and pointers for future work.

This paper is an extended version of [23]. We include a new section on duality and extra examples: alternating, non-deterministic weighted, and pushdown automata.

2. Monads and their Kleisli and Eilenberg–Moore categories

This section recalls the basics of the theory of monads, as needed here. For more information, see e.g. [29,4,28,7]. A monad is a functor $T: \mathbf{C} \to \mathbf{C}$ together with two natural transformations: a unit $\eta: \mathrm{id}_{\mathbf{C}} \Rightarrow T$ and multiplication $\mu: T^2 \Rightarrow T$. These are required to make the following diagrams commute, for $X \in \mathbf{C}$.



We briefly describe the examples of monads on **Sets** that we use in this paper.

- The powerset monad $\mathcal{P}$ maps a set X to the set $\mathcal{P}X$ of subsets of X, and a function $f: X \to Y$ to $\mathcal{P}(f): \mathcal{P}X \to \mathcal{P}Y$ given by direct image. Its unit is given by singleton $\eta(x) = \{x\}$ and multiplication by union $\mu(\{X_i \in \mathcal{P}X \mid i \in I\}) = \bigcup_{i \in I} X_i$.
- The subprobability distribution monad $\mathcal{D}$ is defined, for a set X and a function $f: X \to Y$, as

$$\mathcal{D}X = \left\{ \varphi: X \to [0,1] \;\middle|\; \sum_{x \in X} \varphi(x) \leq 1 \right\} \qquad \mathcal{D}f(\varphi)(y) = \sum_{x \in f^{-1}(y)} \varphi(x).$$

  The support set of a distribution $\varphi \in \mathcal{D}X$ is defined as

$$\mathrm{supp}(\varphi) = \big\{ x \in X \;\big|\; \varphi(x) \neq 0 \big\}.$$

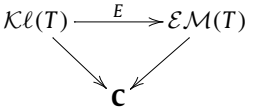
  Note that we do not restrict distributions to finitely supported ones in the definition of $\mathcal{D}$. The unit of $\mathcal{D}$ is given by a Dirac distribution $\eta(x) = \delta_x = (x \mapsto 1)$ for $x \in X$ and the multiplication by $\mu(\Phi)(x) = \sum_{\varphi \in \mathrm{supp}(\Phi)} \Phi(\varphi) \cdot \varphi(x)$ for $\Phi \in \mathcal{D}\mathcal{D}X$.
- For a semiring S, the multiset monad $\mathcal{M}_S$ is defined on a set X as:

$$\mathcal{M}_S X = \big\{ \varphi: X \to S \;\big|\; \mathrm{supp}(\varphi) \text{ is finite} \big\}.$$

  This monad captures multisets $\varphi \in \mathcal{M}_S X$, where the value $\varphi(x) \in S$ gives the multiplicity of the element $x \in X$. When $S = \mathbb{N}$, this is sometimes called the bag monad.
  On functions, $\mathcal{M}_S$ is defined like the subdistribution monad $\mathcal{D}$. Again, the support set of a multiset $\varphi \in \mathcal{M}_S X$ is defined as $\mathrm{supp}(\varphi) = \{x \in X \mid \varphi(x) \neq 0\}$. The finite support requirement is needed for $\mathcal{M}$ to be a monad. The unit η and multiplication μ of $\mathcal{M}_S$ are defined in the same way as for $\mathcal{D}$.

We also use the non-deterministic side-effect monad whose definition we postpone to Section 5.5.

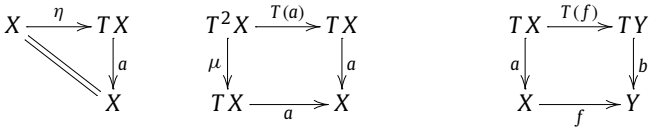
With a monad T on a category $\mathbf{C}$ one associates two categories and a comparison functor E between them, as below.



The "Kleisli" category $\mathcal{K}\ell(T)$ is used to capture computations of type T, in the paradigm that uses monads for effects in a functional world [31]. The objects of the "Eilenberg–Moore" category $\mathcal{EM}(T)$ are algebraic structures, in abstract form. Objects of $\mathcal{EM}(T)$ are called 'algebras', or sometimes 'Eilenberg–Moore algebras' to avoid possible confusion with algebras of a functor F. The latter also form a category written as $\mathbf{Alg}(F)$. The comparison functor $E: \mathcal{K}\ell(T) \to \mathcal{EM}(T)$ plays here the role of pure determinization operation. The categories $\mathcal{K}\ell(T)$ and $\mathcal{EM}(T)$ are initial and final in a suitable sense, see [29] for details. This comparison functor E will be used as the determinization functor. We now describe the above points in more detail.

The Kleisli category $\mathcal{K}\ell(T)$ has the same objects as the underlying category $\mathbf{C}$, but morphisms $X \to Y$ in $\mathcal{K}\ell(T)$ are maps $X \to T(Y)$ in $\mathbf{C}$. The identity map $X \to X$ in $\mathcal{K}\ell(T)$ is T's unit $\eta_X: X \to T(X)$; and composition $g \circ f$ in $\mathcal{K}\ell(T)$ uses T's multiplication in: $g \circ f = \mu \circ T(g) \circ f$. There is a forgetful functor $\mathcal{U}: \mathcal{K}\ell(T) \to \mathbf{C}$, sending X to $T(X)$ and f to $\mu \circ T(f)$. This functor has a left adjoint $\mathcal{F}$, given by $\mathcal{F}(X) = X$ and $\mathcal{F}(f) = \eta \circ f$. Such a Kleisli category $\mathcal{K}\ell(T)$ inherits colimits from the underlying category $\mathbf{C}$.

The category $\mathcal{EM}(T)$ of Eilenberg–Moore algebras has as objects maps of the form $a: T(X) \to X$, making the first two diagrams below commute.



A homomorphism of algebras $(TX \xrightarrow{a} X) \to (TY \xrightarrow{b} Y)$ is a map $f: X \to Y$ in $\mathbf{C}$ between the underlying objects making the diagram above on the right commute. The diagram in the middle thus says that the map a is a homomorphism $\mu \to a$. The forgetful functor $\mathcal{U}: \mathcal{EM}(T) \to \mathbf{C}$ has a left adjoint $\mathcal{F}$, mapping an object $X \in \mathbf{X}$ to the (free) algebra $\mu_X: T^2(X) \to T(X)$ with carrier $T(X)$.

Each category $\mathcal{EM}(T)$ inherits limits from the category $\mathbf{C}$. In the special case where $\mathbf{C} = \mathbf{Sets}$, the category of sets and functions (our standard universe), the category $\mathcal{EM}(T)$ is not only complete but also cocomplete (see [4, § 9.3, Prop. 4]).

The extension functor $E: \mathcal{K}\ell(T) \to \mathcal{EM}(T)$ sends an object $X \in \mathcal{K}\ell(T)$ to the free algebra $E(X) = (\mu: T^2(X) \to T(X))$. For a morphism $f: X \to Y$ in $\mathcal{K}\ell(T)$, that is, $f: X \to T(Y)$ in $\mathbf{C}$, we have $E(f) = \mu \circ T(f): T(X) \to T(Y)$. It forms a map of algebras. Sometimes this $E(f)$ is called the "Kleisli extension" of f.

3. Liftings to Kleisli and Eilenberg–Moore categories

In this section we consider the situation where we have a monad $T: \mathbf{C} \to \mathbf{C}$, with unit η and multiplication μ, and two endofunctors $F, G: \mathbf{C} \to \mathbf{C}$ on the same category $\mathbf{C}$. We will be interested in distributive laws between T, and F or G. These can be of two forms, namely:

$$\begin{array}{lcll} FT & \Longrightarrow & TF & \text{"}F \text{ distributes over } T\text{"} \quad \text{"}\mathcal{K}\ell\text{-law"} \\ TG & \Longrightarrow & GT & \text{"}T \text{ distributes over } G\text{"} \quad \text{"}\mathcal{EM}\text{-law"} \end{array}$$

It is rather difficult to remember whether the monad distributes over a functor or vice versa and so we prefer to use the terminology "$\mathcal{K}\ell$-law" and "$\mathcal{EM}$-law". This is justified by Proposition 1 below.

But first we have to be more precise about what a distributive law is. A $\mathcal{K}\ell$-law $\lambda: FT \Rightarrow TF$ is a natural transformation that commutes appropriately with the unit and multiplication of the monad, i.e., for all X in $\mathbf{C}$ the following diagrams commute:

$$\begin{array}{ccc} FX & = & FX \\ {\scriptstyle F(\eta_X)}\downarrow & & \downarrow{\scriptstyle \eta_{FX}} \\ FTX & \xrightarrow[\lambda_X]{} & TFX \end{array} \qquad \begin{array}{ccccc} FT^2X & \xrightarrow{\lambda_{TX}} & TFTX & \xrightarrow{T(\lambda_X)} & T^2FX \\ {\scriptstyle F(\mu_X)}\downarrow & & & & \downarrow{\scriptstyle \mu_{FX}} \\ FTX & & \xrightarrow[\lambda_X]{} & & TFX \end{array} \tag{3}$$

An $\mathcal{EM}$-law $\rho: TG \Rightarrow GT$ is a natural transformation for which the following diagrams commute for all X in $\mathbf{C}$.

$$\begin{array}{ccc} GX & = & GX \\ {\scriptstyle \eta_{GX}}\downarrow & & \downarrow{\scriptstyle G(\eta_X)} \\ TGX & \xrightarrow[\rho_X]{} & GTX \end{array} \qquad \begin{array}{ccccc} T^2GX & \xrightarrow{T(\rho_X)} & TGTX & \xrightarrow{\rho_{TX}} & GT^2X \\ {\scriptstyle \mu_{GX}}\downarrow & & & & \downarrow{\scriptstyle G(\mu_X)} \\ TGX & & \xrightarrow[\rho_X]{} & & GTX \end{array} \tag{4}$$

The following "folklore" result gives an alternative description of distributive laws in terms of liftings to Kleisli and Eilenberg–Moore categories, see also [25,32] or [3].

Proposition 1 *("laws and liftings"). Assume a monad T and endofunctors F, G on the same category $\mathbf{C}$, as above. There are bijective correspondences between $\mathcal{K}\ell$ ($\mathcal{EM}$)-laws and liftings of F (G) to $\mathcal{K}\ell$ ($\mathcal{EM}$)-categories, in:*

$$\begin{array}{ccc} \mathcal{K}\ell\text{-law } \lambda: FT \Rightarrow TF & & \mathcal{EM}\text{-law } \rho: TG \Rightarrow GT \\ \mathcal{K}\ell(T) \xrightarrow{L} \mathcal{K}\ell(T) & & \mathcal{EM}(T) \xrightarrow{R} \mathcal{EM}(T) \\ \downarrow \quad\quad \downarrow & & \downarrow \quad\quad \downarrow \\ \mathbf{C} \xrightarrow{F} \mathbf{C} & & \mathbf{C} \xrightarrow{G} \mathbf{C} \end{array}$$

Proof. Assuming a $\mathcal{K}\ell$-law $\lambda: FT \Rightarrow TF$ we can define $L: \mathcal{K}\ell(T) \to \mathcal{K}\ell(T)$ as:

$$L(X) = F(X) \qquad L(X \xrightarrow{f} Y) = \big(F(X) \xrightarrow{F(f)} F(TY) \xrightarrow{\lambda_X} T(FY)\big).$$

The above two requirements (3) for λ precisely say that L is a functor.

Conversely, assume there is a functor $L: \mathcal{K}\ell(T) \to \mathcal{K}\ell(T)$ in a commuting square as described in the proposition. Then, on objects, $L(X) = F(X)$. Further, for a map $f: X \to TY$ in $\mathbf{C}$ we get $L(f): FX \to TFY$ in $\mathbf{C}$. This suggests how to define a distributive law: the identity map $\mathrm{id}_{TX}: TX \to TX$ in $\mathbf{C}$ forms a map $TX \to X$ in $\mathcal{K}\ell(T)$, so that we can define $\lambda_X = L(\mathrm{id}_{TX}): FTX \to TFX$ in $\mathbf{C}$. It satisfies (3).

For the second correspondence assume we have an $\mathcal{EM}$-law $\rho: TG \Rightarrow GT$. It gives rise to a functor $R: \mathcal{EM}(T) \to \mathcal{EM}(T)$ by:

$$\begin{pmatrix} TX \\ \downarrow a \\ X \end{pmatrix} \longmapsto \begin{pmatrix} TGX \\ \downarrow G(a)\circ\rho \\ GX \end{pmatrix} \quad \text{and} \quad f \longmapsto G(f).$$

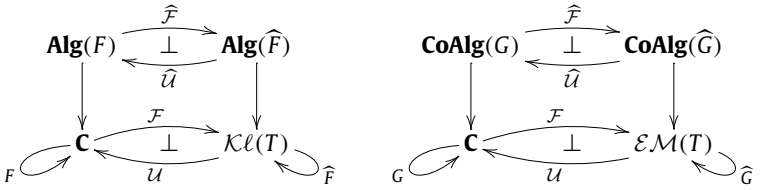
Eqs. (4) guarantee that this yields a new T-algebra.

In the reverse direction, assume a lifting $R: \mathcal{EM}(T) \to \mathcal{EM}(T)$. Applying it to the multiplication μ_X yields an algebra $R(\mu_X): T(GTX) \to GTX$. We then define $\rho_X = R(\mu_X) \circ TG(\eta_X): TGX \to GTX$. Remaining details are left to the reader. □

In what follows we shall simply write $\widehat{F}$, $\widehat{G}$ for the lifting of F, G, both when it comes from a $\mathcal{K}\ell$-law λ or from an $\mathcal{EM}$-law ρ. Usually these laws are fixed, so confusion is unlikely, and a light, overloaded notation is preferred.

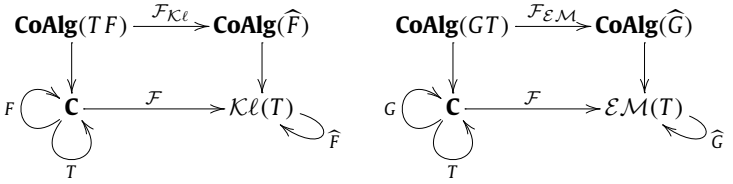
The next result (see also [3]) is not really used in this paper, but it is a natural sequel to the previous proposition since it relates the liftings $\widehat{F}$, $\widehat{G}$ to the standard adjunctions. Recall that we write $\mathbf{Alg}(-)$ and $\mathbf{CoAlg}(-)$ for categories of algebras and coalgebras of a *functor*, not of a (co)monad.

Proposition 2. *In presence of a $\mathcal{K}\ell$-law and an $\mathcal{EM}$-law, the adjunctions $\mathbf{C} \rightleftarrows \mathcal{K}\ell(T)$ and $\mathbf{C} \rightleftarrows \mathcal{EM}(T)$ lift to adjunctions between categories of, respectively, algebras and coalgebras, as described below.*



There is another lifting result, for free functors only, that is relevant in this setting.

Lemma 1. *In presence of a $\mathcal{K}\ell$-law the free functor $\mathcal{F}: \mathbf{C} \to \mathcal{K}\ell(T)$ can be lifted, and similarly, given an $\mathcal{EM}$-law the free algebra functor $\mathcal{F}: \mathbf{C} \to \mathcal{EM}(T)$ can be lifted:*



(5)

The functor $\mathbf{CoAlg}(GT) \to \mathbf{CoAlg}(\widehat{G})$ on the right gives an abstract description of what is called the generalized powerset construction in [36].

Proof. The first part is easy, since the functor $\mathcal{F}_{\mathcal{K}\ell}: \mathbf{CoAlg}(TF) \to \mathbf{CoAlg}(\widehat{F})$ is the identity on objects; it sends a map f of TF-coalgebras to $\mathcal{F}(f) = \eta \circ f$.

Next, assuming an $\mathcal{EM}$-law $\rho: TG \Rightarrow GT$ one defines $\mathcal{F}_{\mathcal{EM}}: \mathbf{CoAlg}(GT) \to \mathbf{CoAlg}(\widehat{G})$ by

$$\mathcal{F}_{\mathcal{EM}}\big(X \xrightarrow{c} GTX\big) = \big(TX \xrightarrow{T(c)} TGTX \xrightarrow{\rho_{TX}} GT^2X \xrightarrow{G(\mu_X)} GTX\big). \tag{6}$$

It is not hard to see that $\mathcal{F}_{\mathcal{EM}}(c)$ is a coalgebra $\mu_X \to \widehat{G}(\mu_X)$ on the free algebra μ_X. On morphisms one simply has $\mathcal{F}_{\mathcal{EM}}(f) = T(f)$. □

$\mathcal{K}\ell$-laws are used to obtain final coalgebras in Kleisli categories ([18]) but this requires non-trivial side-conditions, like enrichment in dcpo's. For $\mathcal{EM}$-laws the situation is much easier, see below; instances of this result have been studied in [36], see also [3].

Proposition 3. *Assume a monad T and endofunctor G on a category $\mathbf{C}$, with an $\mathcal{EM}$-law $\rho: TG \Rightarrow GT$ between them. If G has a final coalgebra $\zeta: Z \xrightarrow{\cong} GZ$ in $\mathbf{C}$, then Z carries a T-algebra structure obtained by finality, as on the left below. The map ζ then forms a map of algebras as on the right, which is the final coalgebra for the lifted functor $\widehat{G}: \mathcal{EM}(T) \to \mathcal{EM}(T)$.*

$$\begin{array}{ccc} GTZ & \xrightarrow{G(\alpha)} & GZ \\ {\scriptstyle \rho \circ T(\zeta)}\uparrow & & \uparrow{\scriptstyle \cong\, \zeta} \\ TZ & \xrightarrow{\alpha} & Z \end{array} \qquad \begin{pmatrix} TZ \\ \downarrow \alpha \\ Z \end{pmatrix} \xrightarrow[\cong]{\zeta} \widehat{G} \begin{pmatrix} TZ \\ \downarrow \alpha \\ Z \end{pmatrix} = \begin{pmatrix} T(GZ) \\ \downarrow G(\alpha) \circ \rho \\ GZ \end{pmatrix}$$

Proof. We leave it to the reader to verify that α is a T-algebra. By construction of α, the map ζ is an algebra homomorphism $\alpha \to \widehat{G}(\alpha)$. Suppose for an arbitrary algebra $b: TY \to Y$ we have a $\widehat{G}$-coalgebra $c: b \to \widehat{G}(b)$. Then $c: Y \to GY$ satisfies

$G(b) \circ \rho \circ T(c) = c \circ b$. By finality in $\mathbf{C}$ there is a unique map $f: Y \to Z$ with $\zeta \circ f = G(f) \circ c$. This f is then the unique map $b \to \alpha$ in $\mathcal{EM}(T)$. $\square$

At this stage we can describe the first form of extension semantics, which we will call $\mathcal{EM}$-extension semantics, since it depends on an $\mathcal{EM}$-law.

Definition 1 *($\mathcal{EM}$-extension).* Let $\rho: TG \Rightarrow GT$ be an $\mathcal{EM}$-law, for $G, T: \mathbf{C} \to \mathbf{C}$, such that the final G-coalgebra $Z \xrightarrow{\cong} GZ$ exists. "Extension" semantics $X \to Z$ in $\mathbf{C}$, for a coalgebra $c: X \to GTX$, is obtained via the following three steps:

1. Transform c into a $\widehat{G}$-coalgebra $\mathcal{F}_{\mathcal{EM}}(c)$.
2. Get the resulting $\widehat{G}$-coalgebra map $TX \to Z$ in $\mathcal{EM}(T)$ by finality.
3. Precompose this map with the unit, yielding $X \to TX \to Z$ in $\mathbf{C}$.

Note that Lemma 1 enables Step 1. Also, recall that the final G-coalgebra $Z \xrightarrow{\cong} GZ$ lifts to a final $\widehat{G}$-coalgebra by Proposition 3, which enables Step 2. Intuitively, Step 1 in Definition 1 corresponds to constructing the determinization of the original coalgebra; Step 2 provides semantics (for the determinization) via finality; Step 3 extracts the semantics for individual states. The reader will find elaborated examples of extension semantics in Section 5.

The next two sections will introduce examples of $\mathcal{EM}$-laws. Here, we briefly look at $\mathcal{K}\ell$-laws. The following result, from [30], shows that these $\mathcal{K}\ell$-laws are quite common, namely for *commutative* monads and *analytic functors*.

Lemma 2. *Let $T: \mathbf{Sets} \to \mathbf{Sets}$ be a commutative monad, and $F: \mathbf{Sets} \to \mathbf{Sets}$ an analytic functor. Then there is a (canonical) $\mathcal{K}\ell$-law $\lambda: FT \Rightarrow TF$.* $\square$

4. Deterministic automata

This section briefly describes deterministic automata as coalgebras, recalls the final coalgebra, and introduces an associated $\mathcal{EM}$-law.

For arbitrary sets A, B there is an endofunctor $B \times (-)^A: \mathbf{Sets} \to \mathbf{Sets}$. Its coalgebras $\phi = \langle \phi_o, \phi_i \rangle: X \to B \times X^A$ are deterministic (Moore) automata. The map $\phi_o: X \to B$ describes the immediate output. The map $\phi_i: X \to X^A$ is the transition function, mapping a state $x \in X$ and an input $a \in A$ to a successor state $\phi_i(x)(a) \in X$. For the special case $B = 2 = \{0, 1\}$, the map ϕ_o tells of a state whether it is final or not. The following result is standard, so we omit the proof.

Lemma 3. *The final coalgebra of the functor $B \times (-)^A$ on $\mathbf{Sets}$ is given by the set of functions $B^{A^\star}$, with structure:*

$$B^{A^\star} \xrightarrow{\zeta = \langle \zeta_o, \zeta_i \rangle} B \times \left(B^{A^\star}\right)^A$$

defined via the empty sequence $\langle\rangle \in A^\star$ and via prefixing $a \cdot \sigma$ of $a \in A$ to $\sigma \in A^\star$: For $t \in B^{A^\star}$

$$\zeta_o(t) = t(\langle\rangle) \quad \text{and} \quad \zeta_i(t)(a) = \lambda \sigma \in A^\star.\, t(a \cdot \sigma). \qquad \square$$

This result captures the paradigm of trace semantics: a state $x \in X$ of an arbitrary coalgebra $X \to B \times X^A$ has a "behaviour" in the carrier of the final coalgebra $B^{A^\star}$ that maps a trace-as-word of inputs in $A^\star$ to an output in B. In the sequel we consider the special case where the output set is the free algebra $T(B)$, for a monad T. In the next result we show that we then get a distributive law. We apply this result only for the category $\mathbf{Sets}$. But it will be formulated more generally, using a strong monad [27] on a Cartesian closed category. This strength is automatic for any monad on $\mathbf{Sets}$ [27].

The following property follows directly from the properties of an alternative formulation of strength $\mathrm{st}: T(U^V) \to T(U)^V$, given by $\mathrm{st}(h)(x) = T(\lambda p.\ p(x))(h)$ for $h \in T(U^V)$ and $x \in V$. Such a formulation of strength [22, Exercise 4.8.13] arises from the monad strength st_T by $\mathrm{st} = T(\mathrm{ev})^A \circ \Lambda(\mathrm{st}_T)$ where $\mathrm{ev}: A \times B^A \to B$ is the evaluation map and $\Lambda(f): A \to B^C$, for an arrow $f: A \times C \to B$, is the abstraction map.

Lemma 4. *Let T be a strong monad on a Cartesian closed category $\mathbf{C}$ and let $A, B \in \mathbf{C}$ be arbitrary objects. Consider the associated "machine" endofunctor $M = T(B) \times (-)^A$ on $\mathbf{C}$. Then there is an $\mathcal{EM}$-law $\rho: TM \Rightarrow MT$ given by:*

$$\rho_X = \left(T\left(T(B) \times X^A\right) \xrightarrow{\langle T(\pi_1), T(\pi_2) \rangle} T^2(B) \times T(X^A) \xrightarrow{\mu \times \mathrm{st}} T(B) \times T(X)^A\right). \qquad \square$$

This $\mathcal{EM}$-law can be defined slightly more generally, not with a free algebra $T(B)$ as output, but with an arbitrary algebra. But in fact, most of our examples involve free algebras.

The resulting lifting $\widehat{M}: \mathcal{EM}(T) \to \mathcal{EM}(T)$ sends an algebra $a: TX \to X$ to the algebra $TMX \to MX$ given by:

$$T\left(T(B) \times X^A\right) \xrightarrow{\langle T(\pi_1), T(\pi_2) \rangle} T^2(B) \times T(X^A) \xrightarrow{\mu \times (a^A \circ \mathrm{st})} T(B) \times X^A$$

Proposition 3, in combination with Lemma 3, says that the final M-coalgebra $T(B)^{A^\star}$ carries a T-algebra structure that forms the final $\widehat{M}$-coalgebra in $\mathcal{EM}(T)$. With a bit of effort one shows that this algebra on $T(B)^{A^\star}$ is given by:

$$\alpha = \big(T\big(T(B)^{A^\star}\big) \xrightarrow{\text{st}} \big(T^2(B)\big)^{A^\star} \xrightarrow{\mu^{A^\star}} T(B)^{A^\star}\big)$$

Then, by Proposition 3, the map $\zeta : \alpha \xrightarrow{\cong} \widehat{M}(\alpha)$ from Lemma 3 forms the final $\widehat{M}$-coalgebra.

4.1. Finality via duality

We have just seen how to obtain a final coalgebra in categories of Eilenberg–Moore algebras, for the lifted automaton functor $B \times (-)^A$. It turns out that in the particular cases that we are interested in there is an alternative way to obtain such finality, namely via duality. This will be described in the current subsection; it involves a very limited class of functors, namely of the form $F = B + A \times (-)$ where B, A are sets. The results below are not needed for what follows. This duality-based approach makes a connection to the work of [34,26].

We start from a basic result, given as exercise in [22].

Lemma 5. *Let $T : \mathbf{Sets} \to \mathbf{Sets}$ be a monad, and $d : T(D) \to D$ an arbitrary (but fixed) Eilenberg–Moore algebra. Then there is an adjunction:*

$$\mathbf{Sets}^{\text{op}} \underset{Hom(-,d)}{\overset{D^{(-)}}{\underset{\longleftarrow}{\overset{\longrightarrow}{\top}}}} \mathcal{EM}(T) \tag{7}$$

Proof. Since algebras are closed under products, for each set X there is an algebra structure on D^X given by:

$$T\big(D^X\big) \xrightarrow{\text{st}} T(D)^X \xrightarrow{d^X} D^X,$$

where st is the strength map (as in Lemma 4). It is easy to see that for functions $f : X \to Y$ and $\varphi : Y \to D$ the map $D^f(\varphi) = \varphi \circ f : D^Y \to D^X$ is a homomorphism of algebras. Thus we have a functor $\mathbf{Sets}^{\text{op}} \to \mathcal{EM}(T)$.

In the other direction we map an algebra $\beta : T(B) \to B$ to the set

$$Hom(\beta, d) = \big\{ g : B \to D \;\big|\; g \circ d = \beta \circ T(g) \big\}.$$

This obviously yields a functor $\mathcal{EM}(T) \to \mathbf{Sets}^{\text{op}}$. Moreover, there is a bijective correspondence:

$$\frac{\begin{pmatrix} T(B) \\ {\scriptstyle\beta}\downarrow \\ B \end{pmatrix} \xrightarrow{\quad f \quad} \begin{pmatrix} T(D^X) \\ {\scriptstyle d^X \circ \text{st}}\downarrow \\ D^X \end{pmatrix}}{X \xrightarrow{\quad g \quad} Hom(\beta, d)}$$

It is obtained by swapping arguments.

- Given an algebra map $f : B \to D^X$ define $\overline{f} : X \to D^B$ by $\overline{f}(x)(b) = f(b)(x)$. We have to show that $\overline{f}(x) : B \to D$ is an algebra map. Well, for $u \in T(B)$:

$$\begin{aligned}
\big(\overline{f}(x) \circ \beta\big)(u) = \overline{f}\big(x\big)(\beta(u)) = f\big(\beta(u)\big)(x) &= \big(d^X \circ \text{st} \circ T(f)\big)(u)(x) \\
&= d\big(\text{st}\big(T(f)(u)\big)(x)\big) \\
&= d\big(T\big((\lambda p.\ p(x)) \circ f\big)(u)\big) \\
&= d\big(T\big(\lambda b.\ f(b)(x)\big)(u)\big) \\
&= d\big(T\big(\lambda b.\ \overline{f}(x)(b)\big)(u)\big) \\
&= \big(d \circ T(\overline{f}(x))\big)(u).
\end{aligned}$$

- Similarly, for a function $g : X \to Hom(\beta, \alpha)$, the map $\overline{g} : B \to D^X$ defined by $\overline{g}(b)(x) = g(x)(b)$ forms a map of algebras.

It is easy to see that these constructions are inverse to each other. □

We need another basic result about lifting adjunctions in situations with a "dual" adjunction $\mathbf{A}^{\text{op}} \leftrightarrows \mathbf{B}$, like in [24] and many other places.

Lemma 6. *Consider the situation:*

$$\begin{array}{ccccc} H \circlearrowleft & \mathbf{C}^{\mathrm{op}} & \underset{F}{\overset{P}{\rightleftarrows}} & \mathbf{A} & \circlearrowright K \end{array} \qquad \text{with } F \dashv P,\ \text{unit } \eta,\ \text{and counit } \varepsilon \tag{8}$$

We assume a natural transformation $\sigma\colon PH \Rightarrow KP$.

1. *This* σ *gives rise to a functor* $\overline{P}\colon \mathbf{Alg}(H)^{\mathrm{op}} \to \mathbf{CoAlg}(K)$, *given by:*

$$\overline{P}\left(H(B) \xrightarrow{\beta} B\right) = \left(P(B) \xrightarrow{P(\beta)} PH(B) \xrightarrow{\sigma_B} KP(B)\right).$$

2. *If this* $\sigma\colon PH \Rightarrow KP$ *is an isomorphism, then there is a similar lifting of the functor* $F\colon \mathbf{A} \to \mathbf{C}^{\mathrm{op}}$ *to* $\overline{F}\colon \mathbf{CoAlg}(K) \to \mathbf{Alg}(H)^{\mathrm{op}}$, *by:*

$$\overline{F}\left(X \xrightarrow{c} K(X)\right) = \left(HF(X) \xrightarrow[\cong]{\overline{\sigma}_X} FK(X) \xrightarrow{F(c)} F(X)\right).$$

   where $\overline{\sigma}\colon HF \Rightarrow FK$ *is obtained as:*

$$\overline{\sigma} = \left(HF \xRightarrow{\varepsilon HF} FPHF \xRightarrow{F\sigma^{-1}F} FKPF \xRightarrow{FK\eta} FK\right).$$

3. *This yields an adjunction* $\overline{F} \dashv \overline{P}$ *over* $F \dashv P$ *in:*

$$\begin{array}{ccccc} \mathbf{Alg}(H)^{\mathrm{op}} & \underset{\overline{F}}{\overset{\overline{P}}{\rightleftarrows}} & \mathbf{CoAlg}(K) & & \\ \downarrow & & \downarrow & & \\ \mathbf{C}^{\mathrm{op}} & \underset{F}{\overset{P}{\rightleftarrows}} & \mathbf{A} & \circlearrowright K & \\ H \circlearrowleft & & & & \end{array}$$

Proof. By naturality of σ and $\overline{\sigma}$ these $\overline{F}$ and $\overline{P}$ are functors. We need to show that there is a bijective correspondence between homomorphisms of (co)algebras f and g in:

$$\begin{array}{ll} \overline{F}\big(X \xrightarrow{c} K(X)\big) \xrightarrow{f} \big(H(B) \xrightarrow{\beta} B\big) & \text{in } \mathbf{Alg}(H)^{\mathrm{op}} \\ \hline \big(X \xrightarrow{c} K(X)\big) \xrightarrow{g} \overline{P}\big(H(B) \xrightarrow{\beta} B\big) & \text{in } \mathbf{CoAlg}(K) \end{array}$$

This correspondence is essentially the one of the underlying adjunction $F \dashv P$. □

Lemma 7. *Let* T *be a monad with algebra* $d\colon T(D) \to D$ *giving rise to an adjunction* (7). *For arbitrary sets* A, B *consider the two functors* $F, G\colon \mathbf{Sets} \to \mathbf{Sets}$ *given by:*

$$F(X) = B + (A \times X) \quad \text{and} \quad G(X) = D^B \times X^A.$$

1. *There is an* $\mathcal{EM}$*-law* $\rho\colon TG \Rightarrow GT$, *with components:*

$$\rho_X = \Big(T(D^B \times X^A) \xrightarrow{\langle T(\pi_1), T(\pi_2)\rangle} T(D^B) \times T(X^A) \xrightarrow{\mathrm{st}\times\mathrm{st}} T(D)^B \times T(X)^A \xrightarrow{d^B \times \mathrm{id}} D^B \times T(X)^A\Big).$$

2. *This* ρ *induces a lifting* $\widehat{G}\colon \mathcal{EM}(T) \to \mathcal{EM}(T)$ *for which there is an isomorphism* $\sigma\colon D^{(-)}F \Rightarrow \widehat{G}D^{(-)}$.
3. *Lemma 6 now makes it possible to lift the adjunction* (7) *in:*

$$\begin{array}{ccccc} \mathbf{Alg}(F)^{\mathrm{op}} & \underset{}{\overset{}{\rightleftarrows}} & \mathbf{CoAlg}(\widehat{G}) & & \\ \downarrow & & \downarrow & & \\ \mathbf{Sets}^{\mathrm{op}} & \underset{Hom(-,d)}{\overset{D^{(-)}}{\rightleftarrows}} & \mathcal{EM}(T) & \circlearrowright \widehat{G} & \\ F \circlearrowleft & & & & \end{array} \tag{9}$$

Proof. We leave it to the reader to verify that ρ as defined above is indeed a distributive law. The isomorphism $\sigma_X : D^{F(X)} \xrightarrow{\cong} G(D^X)$ in the second point is simply:

$$D^{F(X)} = D^{B+(A\times X)} \cong D^B \times D^{A\times X} \cong D^B \times \left(D^X\right)^A = G(D^X).$$

Some elementary calculations show that it is a map of algebras. The lifting of adjunctions follows directly from Lemma 6. □

Many of the trace semantics examples are instances of the following result.

Theorem 1. *Let $T : \mathbf{Sets} \to \mathbf{Sets}$ be a monad with dual adjunction $\mathbf{Sets}^{\mathrm{op}} \leftrightarrows \mathcal{EM}(T)$ as in (7), resulting from an algebra $d : T(D) \to D$. For each functor $F = B + (A \times (-))$ we consider the associated functor $G = D^B \times (-)^A$ with its lifting $\widehat{G} : \mathcal{EM}(T) \to \mathcal{EM}(T)$ as in Lemma 6. The initial F-algebra $B \times A^\star$ in $\mathbf{Sets}$ then yields the final $\widehat{G}$-coalgebra $D^{B\times A^\star}$ in $\mathcal{EM}(T)$.*

Proof. The initial object in $\mathbf{Alg}(F)$ is final in the dual category $\mathbf{Alg}(F)^{\mathrm{op}}$. Hence it is preserved by the right adjoint $\mathbf{Alg}(F)^{\mathrm{op}} \to \mathbf{CoAlg}(\widehat{G})$ in (9). This right adjoint sends the carrier $B \times A^\star$ of the initial F-algebra to the carrier $D^{B\times A^\star}$ of the final $\widehat{G}$-coalgebra. □

5. Examples of $\mathcal{EM}$-extension semantics

This section describes three examples of $\mathcal{EM}$-laws, one familiar one in the context of non-deterministic automata, a new one for simple Segala systems. The latter involves a more general law (Lemma 8) that can also be used for alternating automata and non-deterministic multiset automata. Finally, we also include pushdown automata.

5.1. Non-deterministic automata, in $\mathcal{EM}$-style

We briefly restate the non-deterministic automaton example from [36], but this time using the general constructions developed so far. A non-deterministic automaton is understood here as a coalgebra $c : X \to 2 \times (\mathcal{P}X)^A$, which is of the form $X \to G(TX)$, where G is the functor $2 \times (-)^A$ and T is the powerset monad $\mathcal{P}$ on $\mathbf{Sets}$.

Since $2 = \{0, 1\}$ is the (carrier of the) free algebra $\mathcal{P}(1)$ on the singleton set $1 = \{*\}$, Lemma 4 applies. It yields an $\mathcal{EM}$-law with components $\rho = \langle \rho_1, \rho_2 \rangle : \mathcal{P}(2 \times X^A) \to 2 \times (\mathcal{P}X)^A$, given by:

$$\begin{cases} \rho_1(U) = 1 & \iff \exists \langle b, h\rangle \in U.\ b = 1 \\ x \in \rho_2(U)(a) & \iff \exists \langle b, h\rangle \in U.\ h(a) = x. \end{cases}$$

Lemma 1 yields a coalgebra $\mathcal{F}_{\mathcal{EM}}(c) = \langle \phi_o, \phi_i \rangle : \mathcal{P}(X) \to 2 \times (\mathcal{P}X)^A$ in the category $\mathcal{EM}(\mathcal{P})$, where:

$$\begin{cases} \phi_o(U) = 1 & \iff \exists x \in U.\ \pi_1 c(x) = 1 \\ y \in \phi_i(U)(a) & \iff \exists x \in U.\ y \in \pi_2 c(x)(a). \end{cases}$$

By Proposition 3 the final G-coalgebra $2^{A^\star} = \mathcal{P}(A^\star)$ of languages is also final for the functor $\widehat{G}$ on $\mathcal{EM}(\mathcal{P})$. Hence, we get a map $[\![-]\!] : \mathcal{P}(X) \to \mathcal{P}(A^\star)$ by finality. Applying this map to the singleton set $\{x\} \in \mathcal{P}(X)$ yields the set of words that are accepted in the state $x \in X$. Thus, the $\mathcal{EM}$-semantics from Definition 1 yields the trace semantics for a non-deterministic automaton $c : X \to 2 \times (\mathcal{P}X)^A$, via the language accepted in a state.

To illustrate that different $\mathcal{EM}$-laws generate different semantics, consider the following alternative law $\rho' = \langle \rho'_1, \rho_2 \rangle$, which is a slight adaptation of the law ρ above, namely by changing the existential quantification in ρ_1 to a universal one.

$$\rho_1{}'(U) = 1 \iff \forall \langle b, h\rangle \in U.\ b = 1$$

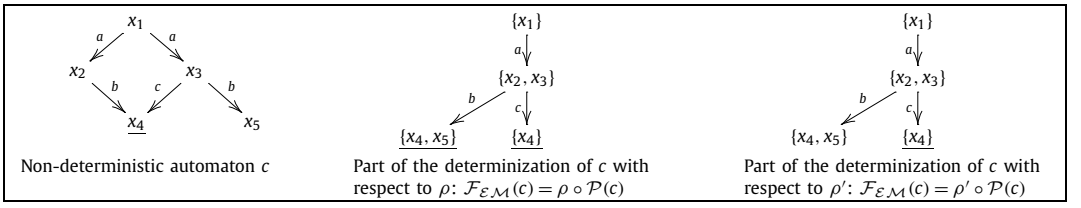
The determinization of $c : X \to 2 \times (\mathcal{P}X)^A$ is now given by $\mathcal{F}_{\mathcal{EM}}(c) = \langle \phi'_o, \phi_i \rangle$, where

$$\phi'_o(U) = 1 \iff \forall x \in U.\ \pi_1 c(x) = 1$$

and ϕ_i is defined as above. The map $[\![-]\!] : \mathcal{P}(X) \to \mathcal{P}(A^\star)$ into the final coalgebra yields, when applied to the singleton set $\{x\} \in \mathcal{P}(X)$, the set of words which, starting from x in the automaton $c : X \to 2 \times (\mathcal{P}X)^A$, always lead to a final state. That is,

$$w \in [\![\{x\}]\!] \iff \text{all paths starting from } x \text{ and labelled by } w \text{ are accepting in the automaton } c$$

Consider the following non-deterministic automaton $c : X \to 2 \times (\mathcal{P}X)^A$, with $X = \{x_1, x_2, x_3, x_4, x_5\}$ and only one final state x_4, which we underline. Starting from state $\{x_1\}$, the reachable parts of the two determinizations, arising from ρ and ρ' are given as follows.



Non-deterministic automaton c | Part of the determinization of c with respect to ρ: $\mathcal{F}_{\mathcal{EM}}(c) = \rho \circ \mathcal{P}(c)$ | Part of the determinization of c with respect to ρ': $\mathcal{F}_{\mathcal{EM}}(c) = \rho' \circ \mathcal{P}(c)$

Note the subtle difference: in the automaton displayed in the middle the state $\{x_1\}$ accepts the language $\{ab, ac\}$, whereas in the automaton depicted on the right the word ab is not accepted, since it also labels a non-accepting path in the automaton c: $x_1 \xrightarrow{a} x_3 \xrightarrow{b} x_5$.

5.2. Simple Segala systems, in $\mathcal{EM}$-style

Next, we consider simple Segala systems as a non-trivial example of $\mathcal{EM}$-extension semantics, which was not considered in [36]. These systems are also called simple probabilistic automata [35], Markov decision processes, or probabilistic labelled transition systems (LTSs). They are coalgebras of the form $c\colon X \to \mathcal{P}(A \times \mathcal{D}X)$, mixing probability and non-determinism. In a recent paper [13], with ideas appearing already in [33,15,10,19], it has been recognized that it might be useful for verification purposes to transform them into so-called distribution LTSs, i.e. into LTSs with state space $\mathcal{D}X$ of so-called uncertain or belief states.

Given a simple Segala system $c\colon X \to \mathcal{P}(A \times \mathcal{D}X)$, we denote by $c^\sharp\colon \mathcal{D}X \to \mathcal{P}(A \times \mathcal{D}X)$ its distribution LTS from [13]. It is defined by

$$\begin{aligned} c^\sharp(\varphi) &= \big\{\langle a, \psi\rangle \;\big|\; \exists x \in \mathrm{supp}(\varphi).\ \langle a, \psi\rangle \in c(x)\big\} \\ &= \big(\mu^{\mathcal{P}} \circ \mathcal{P}(c) \circ \mathrm{supp}\big)(\varphi), \end{aligned} \tag{10}$$

where $\mathrm{supp}(\varphi) = \{x \in X \mid \varphi(x) \neq 0\}$. In the usual notation for transitions, this means that for $\varphi, \psi \in \mathcal{D}X$,

$$\varphi \xrightarrow[c^\sharp]{a} \psi \quad \text{if and only if} \quad \exists x \in \mathrm{supp}(\varphi).\ x \xrightarrow[c]{a} \psi.$$

Here, we capture this situation via a distributive law $\rho\colon \mathcal{D}\mathcal{P}(A \times (-)) \Longrightarrow \mathcal{P}(A \times \mathcal{D})$. It is an $\mathcal{EM}$-law $\rho\colon TG \Rightarrow GT$ for $T = \mathcal{D}$ and $G = \mathcal{P}(A \times -)$ with the property that the $\mathcal{EM}$-extension from Lemma 1 turns a simple Segala system into its distribution LTS, see Proposition 4 below. Explicitly, for a distribution $\Phi \in \mathcal{D}\mathcal{P}(A \times X)$, we define

$$\begin{aligned} \rho(\Phi) &= \big\{\langle a, \delta_x\rangle \;\big|\; \exists U \in \mathrm{supp}(\Phi).\ \langle a, x\rangle \in U\big\} \\ &= \bigcup_{U \in \mathrm{supp}(\Phi)} \big\{\langle a, \delta_x\rangle \;\big|\; \langle a, x\rangle \in U\big\} \\ &= \big(\mu^{\mathcal{P}} \circ \mathcal{P}^2\big(\mathrm{id} \times \eta^{\mathcal{D}}\big) \circ \mathrm{supp}\big)(\Phi), \end{aligned} \tag{11}$$

where δ_x is the Dirac distribution assigning probability 1 to x, i.e. $\delta_x = \eta(x)$. The fact that ρ is an $\mathcal{EM}$-law is an instance of the following result—using that the support forms a map of monads $\mathrm{supp}\colon \mathcal{D} \Rightarrow \mathcal{P}$.

Lemma 8. *Each map of monads $\sigma\colon T \Rightarrow S$ induces an $\mathcal{EM}$-law*

$$TS(A \times -) \Longrightarrow S(A \times T(-))$$

of the monad T *over the* functor $S(A \times -)$. *The components of ρ are given by:*

$$\rho_X = \big(TS(A \times X) \xrightarrow{\sigma_{S(A\times -)X}} S^2(A \times X) \xrightarrow{S^2(\mathrm{id} \times \eta^T_X)} S^2(A \times TX) \xrightarrow{\mu^S_{(A\times TX)}} S(A \times TX)\big). \qquad \square$$

Remark 1. As emphasized, ρ in (11) is a distributive law between the monad $\mathcal{D}$ and the *functor* $\mathcal{P}(A \times -)$. In particular for $A = 1$ it is a distributive law between the monad $\mathcal{D}$ and the *functor* $\mathcal{P}$. An important but subtle point is that ρ is not a distributive law between the monad $\mathcal{D}$ and the *monad* $\mathcal{P}$. There is no such distributive law as shown in [39]. The unit law for the powerset monad, required for a monad distributive law, does not hold for ρ with $A = 1$: $\rho \circ \mathcal{D}(\eta^{\mathcal{P}}) \neq \eta^{\mathcal{P}}\mathcal{D}$. Nevertheless, one can distribute probability over non-determinism, via ρ. The construction provides a non-standard LTS semantics to simple Segala systems, by first lifting these systems to distributions.

The distribution LTS in (10) from [13] is an instance of our general framework.

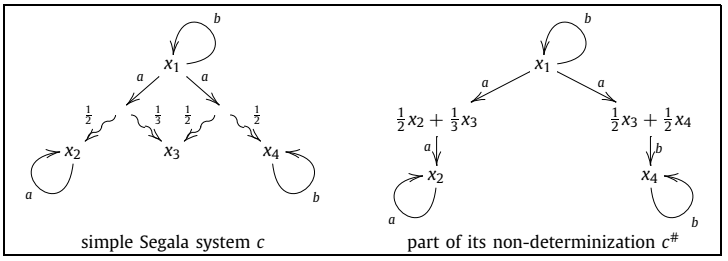
Proposition 4. *Given a simple Segala system $c\colon X \to \mathcal{P}(A \times \mathcal{D}X)$, its $\mathcal{EM}$-extension $\mathcal{F}_{\mathcal{EM}}(c)$ from Lemma 1, obtained via the $\mathcal{EM}$-law ρ from (11), is the same as the coalgebra $c^\#\colon \mathcal{D}X \to \mathcal{P}(A \times \mathcal{D}X)$ described in Eq. (10).*

Proof. By a straightforward calculation:

$$\begin{aligned} \mathcal{F}_{\mathcal{EM}}(c) &\overset{(6)}{=} \mathcal{P}\big(\mathrm{id} \times \mu^{\mathcal{D}}\big) \circ \rho \circ \mathcal{D}(c) \\ &\overset{(11)}{=} \mathcal{P}\big(\mathrm{id} \times \mu^{\mathcal{D}}\big) \circ \mu^{\mathcal{P}} \circ \mathcal{P}^2\big(\mathrm{id} \times \eta^{\mathcal{D}}\big) \circ \mathrm{supp} \circ \mathcal{D}(c) \\ &= \mathcal{P}\big(\mathrm{id} \times \mu^{\mathcal{D}}\big) \circ \mathcal{P}\big(\mathrm{id} \times \eta^{\mathcal{D}}\big) \circ \mu^{\mathcal{P}} \circ \mathrm{supp} \circ \mathcal{D}(c) \end{aligned}$$

$$= \mu^{\mathcal{P}} \circ \mathcal{P}(c) \circ \mathrm{supp}$$
$$\overset{(10)}{=} c^{\sharp}. \qquad \square$$

The following is a simple example of a non-determinization.



In the representation of the non-determinization, on the right, state x_1 is formally $1x_1 = \eta(x_1)$, an element of $\mathcal{D}(X)$. Moreover, on the left, an arrow $x \xrightarrow{a} y$ means $x \xrightarrow{a} \rightsquigarrow y$.

Remark 2. Definition 1 describes how $\mathcal{EM}$-extension semantics arises in presence of a final coalgebra. This does not directly apply in this situation because the (ordinary) powerset functor does not have a final coalgebra, for cardinality reasons. But if we restrict ourselves to finite subsets and distributions with finite support, there is still a map of monads $\mathcal{D}_{\mathrm{fin}} \Rightarrow \mathcal{P}_{\mathrm{fin}}$, so that we get an $\mathcal{EM}$-law $\mathcal{D}_{\mathrm{fin}}\mathcal{P}_{\mathrm{fin}}(A \times -) \Rightarrow \mathcal{P}_{\mathrm{fin}}(A \times \mathcal{D}_{\mathrm{fin}})$ by Lemma 8. For a "finitely branching" Segala system $c\colon X \to \mathcal{P}_{\mathrm{fin}}(A \times \mathcal{D}_{\mathrm{fin}}X)$ one obtains semantics $X \to Z$, where $Z \xrightarrow{\cong} \mathcal{P}_{\mathrm{fin}}(A \times Z)$ is the final coalgebra.

5.3. Alternating automata, in $\mathcal{EM}$-style

Alternating automata with only existential states and transitions [9,11] are coalgebras of the form $c\colon X \to \mathcal{P}(A \times \mathcal{P}X)$ hence of type GT for $G = \mathcal{P}(A \times (-))$ and $T = \mathcal{P}$. As there is a trivial map of monads $\mathrm{id}\colon \mathcal{P} \Rightarrow \mathcal{P}$, Lemma 8 provides us with a distributive law

$$\rho\colon \mathcal{P}\mathcal{P}(A \times (-)) \quad \Rightarrow \quad \mathcal{P}(A \times \mathcal{P}(-))$$

with components, given concretely, by

$$\rho_X = \big(\mathcal{P}\mathcal{P}(A \times X) \xrightarrow{\mathcal{P}\mathcal{P}(A \times \eta_X)} \mathcal{P}\mathcal{P}(A \times \mathcal{P}X) \xrightarrow{\mu_{(A \times \mathcal{P})X}} \mathcal{P}(A \times \mathcal{P}X)\big) \tag{12}$$

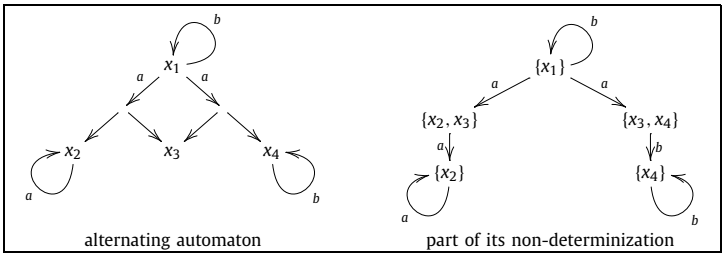
Given an alternating automaton $c\colon X \to \mathcal{P}(A \times \mathcal{P}X)$, the resulting $\mathcal{EM}$-extension $\mathcal{F}_{\mathcal{EM}}(c)\colon \mathcal{P}X \to \mathcal{P}(A \times \mathcal{P}X)$ from Lemma 1, obtained via the $\mathcal{EM}$-law ρ from (12), amounts to

$$\mathcal{F}_{\mathcal{EM}}(c) = G\mu \circ \rho \circ \mathcal{P}(c) = \mu \circ \mathcal{P}(c)$$

which in concrete terms gives

$$\mathcal{F}_{\mathcal{EM}}(c)(U) = \bigcup_{x \in U} c(x), \quad \text{for } U \subseteq X.$$

Clearly, the $\mathcal{EM}$-extension (the non-determinization) of an alternating automaton is once again an LTS, this time over subsets of the original state space. This non-determinization looks pretty much the same as the one for simple Segala systems if one disregards the probabilities. Here is an example. In an alternating automaton, we use unlabelled arrows to denote non-deterministic branching (from the powerset monad) after an a-labelled transition.



Remark 2 applies here as well, $\mathrm{id} : \mathcal{P}_{\mathrm{fin}} \Rightarrow \mathcal{P}_{\mathrm{fin}}$ is of course also a map of monads, leading to semantics $X \to Z$, where $Z \xrightarrow{\cong} \mathcal{P}_{\mathrm{fin}}(A \times Z)$ is the final coalgebra, for a "finitely branching" alternating automaton $c\colon X \to \mathcal{P}_{\mathrm{fin}}(A \times \mathcal{P}_{\mathrm{fin}}X)$.

5.4. Non-deterministic weighted automata, in $\mathcal{EM}$-style

Coalgebras of the form $c: X \to \mathcal{P}(A \times \mathcal{M}_S X)$ are known as non-deterministic weighted automata or non-deterministic multiset automata [14,17]. They allow non-deterministic choice over labelled transitions into a multiset with weights from a semiring S, commonly the semiring of natural numbers. These coalgebras are again of type GT for the functor $G = \mathcal{P}(A \times (-))$ and the monad $T = \mathcal{M}_S$.

Similar to the case of simple Segala systems, the support forms a map of monads $\mathrm{supp}: \mathcal{M}_S \Rightarrow \mathcal{P}$ and Lemma 8 provides us with a distributive law

$$\mathcal{M}_S\mathcal{P}\big(A \times (-)\big) \xRightarrow{\rho} \mathcal{P}\big(A \times \mathcal{M}_S(-)\big)$$

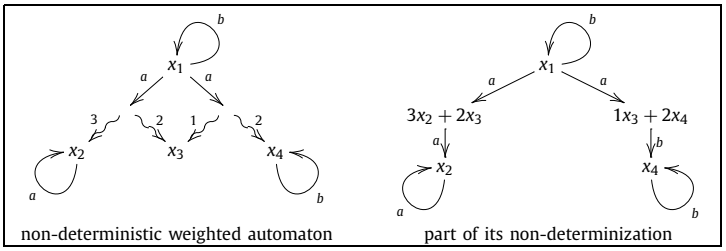
with components:

$$\rho_X = \mu^{\mathcal{P}} \circ \mathcal{P}\mathcal{P}\big(\mathrm{id} \times \eta^{\mathcal{M}_S}\big) \circ \mathrm{supp}_X .$$

Just like for simple Segala systems, this amounts to an $\mathcal{EM}$-extension $\mathcal{F}_{\mathcal{EM}}(c): \mathcal{M}_S X \to \mathcal{P}(A \times \mathcal{M}_S X)$ (a non-determinization) of a non-deterministic weighted automaton $c: X \to \mathcal{P}(A \times \mathcal{M}_S X)$, given by

$$\mathcal{F}_{\mathcal{EM}}(c) = \mu^{\mathcal{P}} \circ \mathcal{P}(c) \circ \mathrm{supp}_X .$$

The only difference with the non-determinization of simple Segala systems is that probabilities are replaced by weights in a semiring. For completeness, we give an example with $S = \mathbb{N}$.



Again, similar to the example of Segala systems, we denote, on the right, by x the formal sum $1x = \eta(x) \in \mathcal{M}_S(X)$. Also here Remark 2 applies as $\mathrm{supp}: \mathcal{M}_S \Rightarrow \mathcal{P}_{\mathrm{fin}}$ is a map of monads, leading to LTS semantics for "finitely branching" non-deterministic weighted automata. Note that the multiset monad always has finite branching (finite support), so the finite branching requirement applies to the non-deterministic branching only: a finitely branching non-deterministic weighted automaton is one of the form $c: X \to \mathcal{P}_{\mathrm{fin}}(A \times \mathcal{M}_S X)$.

5.5. Pushdown automata, in $\mathcal{EM}$-style

In this section we give another example of the framework by modeling pushdown automata coalgebraically, following [37], using the general constructions developed so far.

For this purpose, we will make use of $T(X) = \mathcal{P}(S \times X)^S$, the non-deterministic side-effect monad [5] with side-effects in a set S. The unit $\eta_X: X \to T(X)$ is given by $\eta(x)(s) = \{\langle s, x\rangle\}$, and the multiplication $\mu_X: TT(X) \to T(X)$ is given, for $f \in \mathcal{P}(S \times (\mathcal{P}(S \times X)^S))^S$, by

$$\mu_X(f)(s) = \bigcup_{\langle s', g\rangle \in f(s)} g(s')$$

A (realtime) pushdown automaton (PDA) [1] is a tuple $(Q, \Sigma, \Gamma, \delta, \langle q_0, \alpha_0\rangle, C_F)$, where Q is the set of states, Σ is the set of input symbols, Γ is the set of stack symbols, $\delta: Q \to \mathcal{P}(Q \times \Gamma^\star)^{\Sigma \times \Gamma}$ is the transition function, $\langle q_0, \alpha_0\rangle \in Q \times \Gamma^\star$ is the initial configuration and $C_F \subseteq Q \times \Gamma^\star$ is the set of final configurations. Here, $Q \times \Gamma^\star$ is the set of all configurations.

This definition is slightly non-standard in the sense that no transition with empty word as input or empty stack is allowed, which is what "realtime" refers to. However, the definition is equivalent to the standard one, i.e., realtime PDA with the usual sets of accepting configurations accept the class of context-free languages, and it allows for a smooth coalgebraic treatment.

For convenience, we introduce the notion of transition relation for configurations. A configuration $\langle q, b\beta\rangle$ evolves to a configuration $\langle q', \alpha\beta\rangle$ by reading input $a \in \Sigma$, written $\langle q, b\beta\rangle \xrightarrow{a} \langle q', \alpha\beta\rangle$, if and only if $\langle q', \alpha\rangle \in \delta(q)(a, b)$. The transition relation extends to words $w \in \Sigma^\star$ in the usual way.

A word $w \in \Sigma^\star$ is accepted by a PDA if $\langle q_0, \alpha_0\rangle \xrightarrow{w} \langle q, \alpha\rangle$ and $\langle q, \alpha\rangle \in C_F$.

This definition captures several (equivalent) versions of acceptance for PDA. For instance, by taking $C_F = F \times \{\langle\rangle\}$, for $F \subseteq Q$, we obtain the so-called acceptance by final states and empty stack.

Every pushdown automaton induces a function $\langle o, t\rangle \colon Q \to GTQ$ where G is the functor $2^{\Gamma^\star} \times (-)^\Sigma$ and T is the non-deterministic side-effect monad defined above specialized for $S = \Gamma^\star$ (intuitively, side effects in a pushdown machine are changes in the stack). The functions $o \colon Q \to 2^{\Gamma^\star}$ and $t \colon Q \to (\mathcal{P}(Q \times \Gamma^*)^{\Gamma^\star})^\Sigma$ are defined as

$$\begin{aligned} o(q)(\beta) = 1 \quad &\text{if and only if} \quad \langle q, \beta\rangle \in C_F \\ t(q)(a)\big(\langle\rangle\big) &= \emptyset \\ t(q)(a)(b\beta) &= \big\{\langle q', \alpha\beta\rangle \bigm| \langle q', \alpha\rangle \in \delta(q)(a, b)\big\} \end{aligned}$$

Hence, the output function o keeps track of final configuration and the transition function t describes the steps between PDA configurations as specified by the transition function δ. In particular $t(q)(a)(b) = \delta(q)(a, b)$.

Note that every pushdown automaton gives rise to a GT-coalgebra, but not every function $\langle o, t\rangle \colon Q \to GTQ$ defines a (realtime) pushdown automaton, since, for instance, $t(q)$ may depend on the content of the whole stack and not just on the top element.

We write $q \xrightarrow{a,b|\alpha} q'$ for $\langle q', \alpha\rangle \in t(q)(a)(b)$ indicating that the automaton is in the state q, reads an input symbol a, pops b from the stack, moves to state q' and pushes the string $\alpha \in \Gamma^\star$ on top of the stack.

There is an $\mathcal{EM}$-law $\rho \colon TG \Rightarrow GT$ with components:

$$\mathcal{P}\big(\Gamma^\star \times \big(2^{\Gamma^\star} \times Q^\Sigma\big)\big)^{\Gamma^\star} \xrightarrow{\rho_Q = \langle \rho_1, \rho_2\rangle} 2^{\Gamma^\star} \times \big(\mathcal{P}\big(\Gamma^\star \times Q\big)^{\Gamma^\star}\big)^\Sigma$$

given by:

$$\begin{cases} \rho_1(f)(\alpha) = 1 & \iff \exists \langle \beta, \langle g, t\rangle\rangle \in f(\alpha).\ g(\beta) = 1 \\ \langle \beta, q\rangle \in \rho_2(f)(a)(\alpha) & \iff \exists \langle \gamma, \langle g, t\rangle\rangle \in f(\alpha).\ \gamma = \beta \text{ and } q = t(a). \end{cases}$$

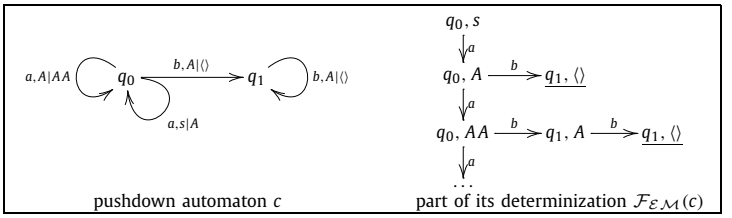
This law is an instance of the general law defined in Lemma 7, for $B = 1$ and $D = 2^{\Gamma^\star}$. Given a coalgebra $c \colon Q \to GTQ$, its $\mathcal{EM}$-extension $\mathcal{F}_{\mathcal{EM}}(c) \colon TQ \to GTQ$ is

$$\mathcal{F}_{\mathcal{EM}}(c) = (id \times \mu^\Sigma) \circ \rho \circ \mathcal{P}(id \times c)^{\Gamma^\star}.$$

More concretely, given such a coalgebra $c = \langle o, t\rangle$, we get $\mathcal{F}_{\mathcal{EM}}(c) = \langle o^\sharp, t^\sharp\rangle \colon \mathcal{P}(\Gamma^\star \times Q)^{\Gamma^\star} \to 2^{\Gamma^\star} \times (\mathcal{P}(\Gamma^\star \times Q)^{\Gamma^\star})^\Sigma$ as follows. For $f \in \mathcal{P}(\Gamma^\star \times Q)^{\Gamma^\star}$,

$$\begin{cases} o^\sharp(f)(\alpha) = 1 & \iff \exists \langle \beta, q\rangle \in f(\alpha).\ o(q)(\beta) = 1 \\ \langle \beta, q\rangle \in t^\sharp(f)(a)(\alpha) & \iff \exists \langle \gamma, r\rangle \in f(\alpha).\ \langle \beta, q\rangle \in t(r)(a)(\gamma). \end{cases}$$

The extension (determinization) can be thought of, in this case, as the infinite deterministic automaton that recognizes the same language. As an example consider the PDA below, on the left, with starting configuration $\langle q_0, s\rangle$ and set of final configurations $\{\langle q_1, \langle\rangle\rangle\}$. The language accepted by this PDA is $\{a^i b^i \mid i \geq 1\}$.



In the representation of the determinization above, on the right, we represent the state (function) $f \in TQ = \mathcal{P}(\Gamma^\star \times Q)^{\Gamma^\star}$ by its value when applied to the current stack, starting from $\eta(q_0)$. Hence, the initial state is represented by $\eta(q_0)(s) = \{\langle s, q_0\rangle\}$. We refrain from writing unnecessary brackets and swap the order of the PDA state and the stack content for better readability. This representation allows us to put in evidence that the determinization indeed is a representation of the infinite deterministic automaton that would recognize the language. Final states are, as before, underlined. Note that to enable this representation we actually use the bijective correspondence:

$$\begin{array}{c} \mathcal{P}(\Gamma^\star \times Q)^{\Gamma^\star} \longrightarrow 2^{\Gamma^\star} \times (\mathcal{P}(\Gamma^\star \times Q)^{\Gamma^\star})^\Sigma \\ \hline\hline \mathcal{P}(\Gamma^\star \times Q)^{\Gamma^\star} \longrightarrow 2^{\Gamma^\star} \times (\mathcal{P}(\Gamma^\star \times Q)^\Sigma)^{\Gamma^\star} \\ \hline\hline \mathcal{P}(\Gamma^\star \times Q)^{\Gamma^\star \times \Gamma^\star} \longrightarrow 2 \times \mathcal{P}(\Gamma^\star \times Q)^\Sigma \end{array}$$

Clearly, $G = 2^{\Gamma^\star} \times (-)^\Sigma$ is a particular deterministic automata functor and its final coalgebra (see Lemma 3) is carried by $(2^{\Gamma^*})^{\Sigma^*}$. It lifts to a final $\widehat{G}$-coalgebra in $\mathcal{EM}(T)$ by Proposition 3 and we get extension semantics by

$$Q \xrightarrow{\eta} TQ \xrightarrow{[\![-]\!]} \big(2^{\Gamma^*}\big)^{\Sigma^*}$$

where $[\![-]\!]$ is the unique map into the final.

The extension semantics allows us to recover the definition of acceptance for PDA. A word $w \in A^\star$ is accepted by a PDA-coalgebra with initial configuration $\langle q_0, \alpha_0 \rangle$ if and only if $[\![\eta(q_0)]\!](w)(\alpha_0) = 1$. If one takes $C_F = F \times \{\langle\rangle\}$ and $\alpha_0 = \langle\rangle$, this definition coincides with the usual acceptance definition (via final states and empty stack): a word $w \in \Sigma^\star$ is accepted by a PDA-coalgebra if, starting from the initial state and empty stack, by reading w the PDA can reach a final state and an empty stack.

6. $\mathcal{K}\ell$-Extension semantics

In a next step we wish to develop extension semantics not only for coalgebras of the form $X \to G(TX)$, on the left in (1), but also for coalgebras $X \to T(FX)$, on the right in (1). This turns out to be more complicated. First of all, the lifting $\mathcal{F}_{\mathcal{K}\ell}\colon \mathbf{CoAlg}(TF) \to \mathbf{CoAlg}(\widehat{F})$ in Lemma 1 is not interesting in the current setting because it does not involve a state space extension $X \mapsto T(X)$.

The next thought might be to translate coalgebras $X \to T(FX)$ into coalgebras $X \to G(TX)$, via a distributive law $TF \Rightarrow GT$. This results in functors

$$\mathbf{CoAlg}(TF) \longrightarrow \mathbf{CoAlg}(GT) \xrightarrow{\text{Lemma 1}} \mathbf{CoAlg}(\widehat{G})$$

where the first functor is obtained by post-composing with the distributive law. We will get back to this point later. However, coalgebras $X \to T(FX)$ are usually considered as coalgebras $X \to \widehat{F}X$ of the lifted functor $\widehat{F}$ on the Kleisli category $\mathcal{K}\ell(T)$. Therefore, our aim is to obtain a functor $\mathbf{CoAlg}(\widehat{F}) \to \mathbf{CoAlg}(\widehat{G})$. This will be described below.

Recall from Section 2 that there is a comparison functor $E\colon \mathcal{K}\ell(T) \to \mathcal{EM}(T)$. In this section we show how it can be lifted to coalgebras. We consider the following situation.

1. A monad $T\colon \mathbf{C} \to \mathbf{C}$ on a category $\mathbf{C}$.
2. An endofunctor $F\colon \mathbf{C} \to \mathbf{C}$ with a $\mathcal{K}\ell$-law $\lambda\colon FT \Rightarrow TF$, leading to a lifting $\widehat{F}\colon \mathcal{K}\ell(T) \to \mathcal{K}\ell(T)$ to T's Kleisli category, via Proposition 1.
3. Another endofunctor $G\colon \mathbf{C} \to \mathbf{C}$, but this time with an $\mathcal{EM}$-law $\rho\colon TG \Rightarrow GT$, yielding a lifting $\widehat{G}\colon \mathcal{EM}(T) \to \mathcal{EM}(T)$ to the category of T-algebras.
4. An "extension" natural transformation $\mathfrak{e}\colon TF \Rightarrow GT$ that connects the $\mathcal{K}\ell$- and $\mathcal{EM}$-laws via the following two commuting diagrams.

$$\begin{array}{ccccc} TFT & \xrightarrow{T(\lambda)} & T^2F & \xrightarrow{\mu F} & TF \\ {\scriptstyle \mathfrak{e}T}\downarrow & & & & \downarrow{\scriptstyle \mathfrak{e}} \\ GT^2 & & \xrightarrow{G\mu} & & GT \end{array} \qquad \begin{array}{ccccc} T^2F & & \xrightarrow{\mu F} & & TF \\ {\scriptstyle T\mathfrak{e}}\downarrow & & & & \downarrow{\scriptstyle \mathfrak{e}} \\ TGT & \xrightarrow{\rho T} & GT^2 & \xrightarrow{G\mu} & GT \end{array} \tag{13}$$

When such $\mathfrak{e}$ is an isomorphism, it forms a "commuting pair of endofunctors" from [3].

Theorem 2. *Assuming the above points 1–4, there is a lifting $\widehat{E}$ of the extension functor E in:*

$$\begin{array}{ccc} \mathbf{CoAlg}(\widehat{F}) & \xrightarrow{\widehat{E}} & \mathbf{CoAlg}(\widehat{G}) \\ \downarrow & & \downarrow \\ \mathcal{K}\ell(T) & \xrightarrow{E} & \mathcal{EM}(T) \\ {\scriptstyle \widehat{F}}\circlearrowleft & & \circlearrowright{\scriptstyle \widehat{G}} \end{array}$$

This functor $\widehat{E}$ is automatically faithful; and it is also full if the extension natural transformation $\mathfrak{e}\colon TF \Rightarrow GT$ consists of monomorphisms.

Proof. We define the functor $\widehat{E}\colon \mathbf{CoAlg}(\widehat{F}) \to \mathbf{CoAlg}(\widehat{G})$ by:

$$\begin{aligned} \big(X \xrightarrow{c} \widehat{F}X\big) &\longmapsto \big(TX \xrightarrow{T(c)} T^2FX \xrightarrow{\mu} TFX \xrightarrow{\mathfrak{e}} GTX\big) \\ f &\longmapsto E(f) = \mu \circ T(f). \end{aligned}$$

We need to show that $\widehat{E}(c)$ is a map of algebras $E(X) = \mu_X \to \widehat{G}(\mu_X) = \widehat{G}(EX)$ in:

$$\begin{pmatrix} T^2X \\ \downarrow{\scriptstyle \mu_X} \\ TX \end{pmatrix} \xrightarrow{\widehat{E}(c)} \begin{pmatrix} TGTX \\ \downarrow{\scriptstyle G(\mu_X)\circ\rho} \\ GTX \end{pmatrix} = \widehat{G}\begin{pmatrix} T^2X \\ \downarrow{\scriptstyle \mu_X} \\ TX \end{pmatrix}$$

But this is simply the above requirement 4.

Assume f is a map of $\widehat{F}$-coalgebras, from $c: X \to \widehat{F}X$ to $d: Y \to \widehat{F}Y$. That is, $f: X \to TY$, $c: X \to TFX$ and $d: Y \to TFY$ satisfy:

$$\mu \circ T(d) \circ f = \mu \circ T\big(\lambda \circ F(f)\big) \circ c. \tag{14}$$

Then $E(f) = \mu \circ T(f)$ is a map of coalgebras $\widehat{E}(c) \to \widehat{E}(d)$, again by requirement 4:

$$\begin{aligned}
\widehat{E}(d) \circ E(f) &= \mathfrak{e}_Y \circ \mu \circ T(d) \circ \mu \circ T(f) \\
&= \mathfrak{e}_Y \circ \mu \circ \mu \circ T\big(T(d) \circ f\big) \\
&= \mathfrak{e}_Y \circ \mu \circ T(\mu \circ T(d) \circ f) \\
&\overset{(14)}{=} \mathfrak{e}_Y \circ \mu \circ T\big(\mu \circ T\big(\lambda \circ F(f)\big) \circ c\big) \\
&= \mathfrak{e}_Y \circ \mu \circ \mu \circ T^2\big(\lambda \circ F(f)\big) \circ T(c) \\
&= \mathfrak{e}_Y \circ \mu \circ T\big(\lambda \circ F(f)\big) \circ \mu \circ T(c) \\
&\overset{(13)}{=} G(\mu) \circ \mathfrak{e}_{TY} \circ TF(f) \circ \mu \circ T(c) \\
&= G\big(\mu \circ T(f)\big) \circ \mathfrak{e}_X \circ \mu \circ T(c) \\
&= \widehat{G}(Ef) \circ \widehat{E}(c).
\end{aligned}$$

Clearly, the functor $\widehat{E}$ is faithful: if $f, g: X \to TY$ satisfy $\widehat{E}(f) = E(f) = E(g) = \widehat{E}(g)$, then $f = E(f) \circ \eta = E(g) \circ \eta = g$.

Now assume the components $\mathfrak{e}_X: TFX \to GTX$ are monomorphisms in $\mathbf{C}$. Towards fulness of $\widehat{E}$, let $h: \widehat{E}(c) \to \widehat{E}(d)$ be a morphism in $\mathbf{CoAlg}(\widehat{G})$. It is a map $h: TX \to TY$ that is both a map of algebras and of coalgebras, so:

$$\begin{array}{ccc}
T^2X & \xrightarrow{T(h)} & T^2Y \\
{\scriptstyle\mu}\downarrow & & \downarrow{\scriptstyle\mu} \\
TX & \xrightarrow[h]{} & TY
\end{array}
\qquad\qquad
\begin{array}{ccc}
GTX & \xrightarrow{G(h)} & GTY \\
{\scriptstyle \mathfrak{e}_X \circ \mu \circ T(c)}\uparrow & & \uparrow{\scriptstyle \mathfrak{e}_Y \circ \mu \circ T(d)} \\
TX & \xrightarrow[h]{} & TY
\end{array} \tag{15}$$

We now take $f = h \circ \eta: X \to T(Y)$ and need to show that $\widehat{E}(f) = E(f) = h$ and that it is a map of $\widehat{F}$-coalgebras $c \to d$. The first is easy since:

$$E(f) = \mu \circ T(h \circ \eta) = h \circ \mu \circ T(\eta) = h.$$

In order to show that f is a map of coalgebras we use that $\mathfrak{e}$ consists of monos, in:

$$\begin{aligned}
\mathfrak{e} \circ \big(d \odot f\big) &= \mathfrak{e} \circ \mu \circ T(d) \circ f \\
&= \mathfrak{e} \circ \mu \circ T(d) \circ h \circ \eta \\
&\overset{(15)}{=} G(h) \circ \mathfrak{e} \circ \mu \circ T(c) \circ \eta \\
&= G(h) \circ \mathfrak{e} \circ \mu \circ \eta \circ c \\
&= G(h) \circ \mathfrak{e} \circ c \\
&= G(h) \circ G\big(\mu \circ T(\eta)\big) \circ \mathfrak{e} \circ c \\
&\overset{(15)}{=} G(\mu) \circ GT(h \circ \eta) \circ \mathfrak{e} \circ c \\
&= G(\mu) \circ \mathfrak{e} \circ TF(f) \circ c \\
&\overset{(13)}{=} \mathfrak{e} \circ \mu \circ T\big(\lambda \circ F(f)\big) \circ c \\
&= \mathfrak{e} \circ \big(\widehat{F}(f) \odot c\big). \qquad \square
\end{aligned}$$

On a more abstract level, what the previous result does is lift $\mathfrak{e}: TF \Rightarrow GT$ to a natural transformation $\widehat{\mathfrak{e}}: E\widehat{F} \Rightarrow \widehat{G}E$. In this way we can also define the functor $\widehat{E}: \mathbf{CoAlg}(\widehat{F}) \to \mathbf{CoAlg}(\widehat{G})$ by:

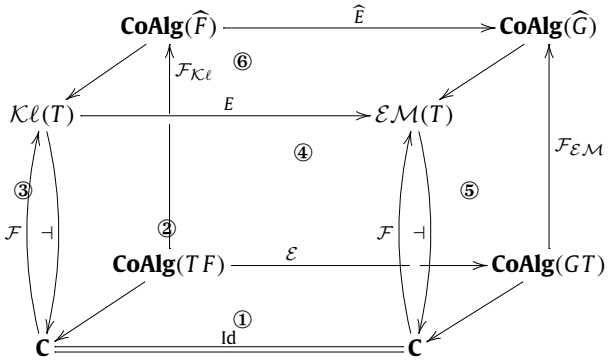
$$\big(X \xrightarrow{c} \widehat{F}X\big) \longmapsto \big(EX \xrightarrow{\widehat{\mathfrak{e}} \circ E(c)} \widehat{G}(EX)\big) \quad \text{and} \quad f \longmapsto E(f) = \mu \circ T(f).$$

Getting back to the original intention, let $\mathcal{E}: \mathbf{CoAlg}(TF) \to \mathbf{CoAlg}(GT)$ be the functor obtained by post-composing with the extension natural transformation $\mathfrak{e}$, that is, given $c: X \to TF(X)$ we have

$$\mathcal{E}(c) = \mathfrak{e}_X \circ c \quad \text{and} \quad \mathcal{E}(f) = f$$

for a coalgebra homomorphism f. We can now summarize all results in one cube-shaped diagram.

Theorem 3. *Under Assumptions 1.–4. from the beginning of this section, we have the following commuting cube of (lifted) functors:*



Proof. The bottom and frontal faces – ① and ② – commute trivially as $\mathcal{E}$ and E are functors. Lemma 1 provides the commutativity of the left and right faces – ③ and ⑤ – under the existence of a $\mathcal{K}\ell$-law and $\mathcal{EM}$-law, respectively. The back face ④ commutes by the definitions: we have, on objects,

$$\begin{aligned}(\mathcal{F}_{\mathcal{EM}} \circ \mathcal{E})(c) &= \mathcal{F}_{\mathcal{EM}}(\mathfrak{e} \circ c) \\ &= G\mu \circ \rho \circ T\mathfrak{e} \circ Tc \\ &\overset{(13)}{=} \mathfrak{e} \circ \mu \circ Tc \\ &= \widehat{E}(c) \\ &= (\widehat{E} \circ \mathcal{F}_{\mathcal{K}\ell})(c)\end{aligned}$$

and on morphisms,

$$\mathcal{F}_{\mathcal{EM}} \circ \mathcal{E}(f) = \mathcal{F}_{\mathcal{EM}}(f) = Tf = \mu \circ T\eta \circ Tf = \mu \circ T(\eta \circ f) = \widehat{E} \circ \mathcal{F}_{\mathcal{K}\ell}(f).$$

Commutativity of the top face ⑥ is given by Theorem 2. The extension law $\mathfrak{e}$ is needed for ④ and ⑥. □

We can now define extension semantics for coalgebras $X \to T(FX)$, analogously to Definition 1. The reader can find examples of $\mathcal{K}\ell$-extension semantics in Section 7.

Definition 2 *($\mathcal{K}\ell$-extension).* In addition to the points 1–4 from the beginning of this section, let the functor G in point 3. have a final coalgebra $Z \xrightarrow{\cong} G(Z)$. For a coalgebra $c\colon X \to T(FX)$ one obtains its $\mathcal{K}\ell$-extension semantics in three steps, like in Definition 1:

1. Transform c into a $\widehat{G}$-coalgebra $\widehat{E}(c)$.
2. Obtain a map $TX \to Z$ in $\mathcal{EM}(T)$ by finality.
3. Get $X \to Z$ by precomposition with the unit $X \to TX$.

Step 1 is justified by Theorem 2. Note that Theorem 3 states that the same transformation is obtained via $\mathcal{F}_{\mathcal{EM}} \circ \mathcal{E}(c)$ since $\widehat{E}(c) = \widehat{E} \circ \mathcal{F}_{\mathcal{K}\ell}(c) = \mathcal{F}_{\mathcal{EM}} \circ \mathcal{E}(c)$. Step 2 is justified by Proposition 3, as the final G-coalgebra $Z \xrightarrow{\cong} G(Z)$ lifts to a final $\widehat{G}$-coalgebra. The intuition behind each of the steps is as follows: Step 1 produces a "determinization" after transforming the original TF-coalgebra to a GT-coalgebra; Step 2 provides semantics via finality, for the determinization; finally Step 3 gives the semantics of individual states.

We conclude this section by showing how the "Kleisli" trace semantics from [18] fits in the current setting. Thus, we assume in the situation of Definition 2 that the functor F has an initial algebra $\iota\colon F(W) \xrightarrow{\cong} W$.

Proposition 5. *Assume the map $\mathcal{F}(\iota^{-1})\colon W \to \widehat{F}(W)$ is final $\widehat{F}$-coalgebra. Each coalgebra $c\colon X \to TF(X)$ then gives rise to a "Kleisli" trace map in the Kleisli category (as in [18]), namely:*

$$\begin{array}{ccc} \widehat{F}X & \dashrightarrow & \widehat{F}(W) \\ {\scriptstyle c}\uparrow & & \cong\uparrow{\scriptstyle \mathcal{F}(\iota^{-1})} \\ X & \xrightarrow{\mathrm{tr}_{\mathcal{K}\ell}(c)} & W \end{array}$$

When we apply the functor $\widehat{E}$ from Theorem 2 to this diagram we get the rectangle on the left in:

$$\begin{array}{ccccc} & & \widehat{G}(TX) & \longrightarrow & \widehat{G}(TW) - - - - - \succ \widehat{G}(Z) \\ & & \widehat{E}(c)\uparrow & \cong & \uparrow\widehat{E}(\mathcal{F}(\iota^{-1})) \qquad \cong\uparrow\zeta \\ X & \xrightarrow{\eta} & TX & \xrightarrow{\widehat{E}(\mathrm{tr}_{\mathcal{K}\ell}(c))} & TW - - - - - - \succ Z \\ & \searrow^{\mathrm{tr}_{\mathcal{K}\ell}(c)} & & \nearrow & \end{array} \tag{16}$$

The resulting $\mathcal{K}\ell$-extension semantics map $X \to Z$ is then the $\mathcal{K}\ell$-extension semantics of the final $\widehat{F}$-coalgebra $\mathcal{F}(\iota^{-1}) \colon W \to \widehat{F}(W)$, precomposed with the Kleisli trace semantics $\mathrm{tr}_{\mathcal{K}\ell}(c)$. □

7. Determinization and trace semantics

In this section we specialize the $\mathcal{K}\ell$-extension framework developed so far to deterministic automata, leading to a novel definition of trace semantics, namely via $\mathcal{K}\ell$-extension semantics. Several illustrations will be given, including the standards ones from the trace semantics of [18].

Definition 3. Let T be a monad and F an endofunctor on the category **Sets**, with a $\mathcal{K}\ell$-law $\lambda \colon FT \Rightarrow TF$ between them. Let A, B be sets, leading to endofunctor $M = T(B) \times (-)^A$, which comes with an $\mathcal{EM}$-law $\rho \colon TM \Rightarrow MT$ like in Lemma 4. Finally, we assume an extension law $\mathfrak{e} \colon TF \Rightarrow MT = T(B) \times T(-)^A$ like in the previous section. In this situation,

- the determinization of a coalgebra $c \colon X \to TFX$ is the $\widehat{M}$-coalgebra $\widehat{E}(c)$ in the category of algebras $\mathcal{EM}(T)$ given by:

$$T(X) \xrightarrow{T(c)} T^2FX \xrightarrow{\mu} TFX \xrightarrow{\mathfrak{e}} T(B) \times T(X)^A$$

  Thus, determinization turns the coalgebra c on X into a deterministic automaton on TX.
- the trace map $\mathrm{tr}(c) \colon X \to T(B)^{A^\star}$ of such a coalgebra c is obtained via the unique coalgebra map $T(X) \to T(B)^{A^\star}$ that arises by finality in $\mathcal{EM}(T)$ in:

$$\begin{array}{ccccc} T(B) \times T(X)^A = \widehat{M}(TX) & - - - - - \succ & \widehat{M}\big(T(B)^{A^\star}\big) = T(B) \times \big(T(B)^{A^\star}\big)^A \\ \widehat{E}(c)\uparrow & & \cong\uparrow\zeta \\ X \xrightarrow{\eta} TX & - - - - - \succ & T(B)^{A^\star} \\ \searrow^{\mathrm{tr}(c)} & \nearrow & \end{array}$$

7.1. Non-deterministic automata, in $\mathcal{K}\ell$-style

In Subsection 5.1 we have seen how to obtain traces for non-deterministic automata via determinization (like in [36]). Now we re-describe the same example in $\mathcal{K}\ell$-style, via the isomorphisms (2). In essence we translate the "Kleisli" trace semantics approach of [18] into the current setting, like in Proposition 5. Thus we start with a non-deterministic automaton as a coalgebra of the form $c \colon X \to \mathcal{P}(1 + A \times X)$, for a fixed set of labels A and $1 = \{*\}$ modeling termination. A state $x \in X$ of such a coalgebra is final if and only if $* \in c(x)$. In this case the monad is the powerset monad $\mathcal{P}$ and the functor is $F = 1 + A \times (-)$ with finite lists $A^\star$ as initial algebra. We recall that the Kleisli category $\mathcal{K}\ell(\mathcal{P})$ is the category **Rel** of sets and relations between them, and the category $\mathcal{EM}(\mathcal{P})$ is the category **CL** of complete lattices with join-preserving maps.

The functor F lifts to $\widehat{F} \colon \mathbf{Rel} \to \mathbf{Rel}$ by Lemma 2. We instantiate M for the set $B = 1$ and the powerset monad, and get $M = 2 \times (-)^A$ since $\mathcal{P}(1) \cong 2$. Then there is an extension law $\mathfrak{e} \colon \mathcal{P}(1 + A \times (-)) \Rightarrow 2 \times \mathcal{P}^A$ with

$$\mathfrak{e}(U) = \big\langle o(U), \lambda a.\, \{x \mid \langle a, x\rangle \in U\}\big\rangle \quad \text{where } o(U) = \begin{cases} 1 & \text{if } * \in U \\ 0 & \text{if } * \notin U. \end{cases}$$

One can easily check that $\mathfrak{e}$ is injective (it is actually an isomorphism) and that it satisfies the requirements (13) for an extension law.

Given a coalgebra $c \colon X \to \mathcal{P}(1 + A \times X)$ its determinization is the deterministic automaton $\widehat{E}(c) \colon \mathcal{P}(X) \to 2 \times \mathcal{P}(X)^A$ with subsets $V \subseteq X$ as states, given by $\widehat{E}(c) = \mathfrak{e} \circ \mu \circ \mathcal{P}(c)$ or, more concretely,

$$\widehat{E}(c)(V) = \left\langle o\left(\bigcup_{x \in V} c(x)\right),\ \lambda a.\ \bigcup_{x \in V} \{y \mid \langle a, y\rangle \in c(x)\} \right\rangle.$$

This determinization coincides with the well-known subset construction [20]. The trace map $\mathrm{tr}(c)$ associates with each state of the original non-deterministic automaton the language it recognizes.

The dashed map $TW \dashrightarrow Z$ in (16) in Proposition 5 is the obvious isomorphism $\mathcal{P}(A^\star) \xrightarrow{\cong} 2^{A^\star}$ for non-deterministic automata. Therefore, Proposition 5 yields that "Kleisli" trace and trace via $\mathcal{K}\ell$-extension semantics coincide, i.e. $\mathrm{tr}(c) = \mathrm{tr}_{\mathcal{K}\ell}(c)$ for any non-deterministic automaton $c \colon X \to \mathcal{P}(1 + A \times X)$.

7.2. Generative probabilistic systems

Next, we consider generative probabilistic systems with explicit termination. They also fit in the "Kleisli" trace approach of [18]. Their determinization was introduced by the last two authors of the present paper in [38] and motivated us to look at the framework of the present paper.

Generative systems are coalgebras for the functor $\mathcal{D}(1 + A \times (-))$ where $\mathcal{D}$ is the subprobability distribution monad. The functor $F = 1 + A \times (-)$ lifts to $\widehat{F}: \mathcal{K}\ell(\mathcal{D}) \to \mathcal{K}\ell(\mathcal{D})$ by Lemma 2. The category $\mathcal{EM}(\mathcal{D})$ is the category **PCA** of positive convex algebras and convex maps [16]. We instantiate the functor M to $B = 1$ and the subprobability distribution monad $\mathcal{D}$, and get $M = [0, 1] \times (-)^A$ since $\mathcal{D}(1) \cong [0, 1]$. Define $\mathfrak{e}: \mathcal{D}(1 + A \times (-)) \Rightarrow [0, 1] \times \mathcal{D}^A$ by

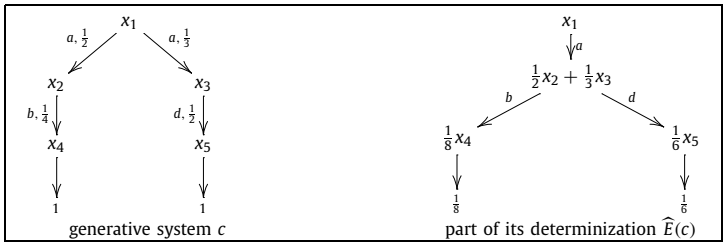
$$\mathfrak{e}(\xi) = \big\langle \xi(*),\ \lambda a.\ \lambda y.\ \xi(a, y) \big\rangle.$$

It is not difficult to check that $\mathfrak{e}$ is an extension law and that it is injective.

Given a coalgebra $c: X \to \mathcal{D}(1 + A \times X)$ its determinization is the deterministic automaton $\widehat{E}(c): \mathcal{D}(X) \to [0, 1] \times \mathcal{D}(X)^A$ with states $\mathcal{D}(X)$, given by $\widehat{E}(c) = \mathfrak{e} \circ \mu \circ \mathcal{D}(c)$ or, more concretely,

$$\widehat{E}(c)(\xi) = \left\langle \sum_{x \in X} \xi(x) \cdot c(x)(*),\ \lambda a.\ \lambda y.\ \sum_{x \in X} \xi(x) \cdot c(x)(a, y) \right\rangle.$$

We show an example of such determinization: the automaton on the right is part of the determinization of the one on the left. The full determinization is an infinite automaton. We show the accessible part when starting from the state $\eta(x_1)$, the Dirac distribution of x_1, and we denote the distributions by formal sums. We omit zero output probabilities.



The trace map $\mathrm{tr}(c)$ associates with each state of the original generative system a subprobability distribution on words, giving the probability to terminate with a given word starting from the given state. For instance, for state x_1 above, we have $\mathrm{tr}(c)(x_1) = \frac{1}{8}ab + \frac{1}{6}ad$.

The dashed map in (16) in Proposition 5 is in this case the inclusion map $\mathcal{D}(A^\star) \to [0, 1]^{A^\star}$. Therefore, again, Proposition 5 yields that "Kleisli" trace and trace via extension semantics "coincide", i.e. $\mathrm{tr}(c)(x)(w) = \mathrm{tr}_{\mathcal{K}\ell}(c)(x)(w)$ for any generative system $c: X \to \mathcal{D}(1 + A \times X)$, any of its states $x \in X$, and any word $w \in A^\star$. Moreover, it shows that the trace map $\mathrm{tr}(c)$ is not just any map from $A^\star$ to $[0, 1]$, but a subprobability distribution on words. This coincidence was shown in [38] in concrete terms.

7.3. Weighted automata

The restrictions imposed on the monad in order for the trace semantics of [18] to work rule out several interesting monads, such as the multiset monads $\mathcal{M}_S$ (including the free vector space monad when S is a field), used for coalgebraic modeling of weighted systems. In this example, we show how the new framework allows us to deal with trace semantics for weighted systems using extension semantics. We consider the same functor $F = 1 + A \times (-)$ as before, with the multiset monad $\mathcal{M}_S$ over a semiring S. Recall that it maps a set X to the set of all finitely supported maps from X to S. Having finite support is crucial for $\mathcal{M}_S$ to be a monad, and one of the reasons why this monad does not fit in the "Kleisli" traces framework of [18]. Similar to probability distributions, we denote multisets by formal sums, now with coefficients in the semiring S. The Kleisli category $\mathcal{K}\ell(\mathcal{M}_S)$ is, for instance, the category of free commutative monoids when $S = \mathbb{N}$, and the category $\mathcal{EM}(\mathcal{M}_S)$ is the category of modules over S.

Coalgebras of the functor $\mathcal{M}_S(1 + A \times (-))$ are precisely weighted automata with weights over the set S. Given a coalgebra $c: X \to \mathcal{M}_S(1 + A \times X)$, each state $x \in X$ has an output weight $c(x)(*) \in S$ and an a-labelled transition into state y with weight $c(x)(\langle a, y\rangle) \in S$.

We instantiate the deterministic automaton functor M with $B = 1$ and the multiset monad $\mathcal{M}_S$, and get $M = S \times (-)^A$ since $\mathcal{M}_S(1) \cong S$. Then there is an extension natural transformation $\mathfrak{e}: \mathcal{M}_S(1 + A \times (-)) \Rightarrow S \times \mathcal{M}_S^A$ given, as for the subdistribution monad, by

$$\mathfrak{e}(\xi) = \big\langle \xi(*),\ \lambda a.\ \lambda x.\ \xi\big(\langle a, x\rangle\big) \big\rangle.$$

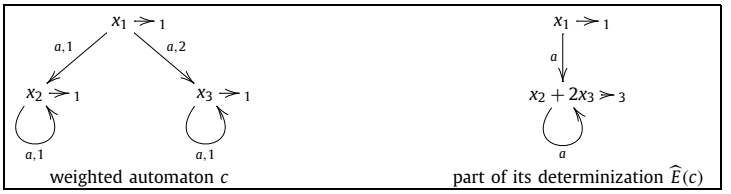
Again, $\mathfrak{e}$ satisfies all the conditions and it is injective.

For weighted automata, just like for generative systems, the determinization construction gives rise to a deterministic automaton with an infinite state-space even if the original automaton is finite. For a weighted automaton $c: X \to \mathcal{M}_S(1 + A \times X)$, the determinization $\widehat{E}(c): \mathcal{M}_S(X) \to S \times \mathcal{M}_S(X)^A$ is given by

$$\widehat{E}(c)(\xi) = \left\langle \sum_{x \in X} \xi(x) \cdot c(x)(*),\ \lambda a.\, \lambda y.\, \sum_{x \in X} \xi(x) \cdot c(x)(a, y) \right\rangle$$

for a multiset $\xi: X \to S$ with finite support $\{x \in X \mid \xi(x) \neq 0\}$. We thus get a trace map $X \to \mathcal{M}_S(X) \to S^{A^\star}$. Notice that the final coalgebra $S^{A^\star}$ in the category of $\mathcal{M}_S$-algebras can be obtained via the adjoint functors $S^{(-)}: \mathbf{Sets}^{\mathrm{op}} \to \mathcal{EM}(\mathcal{M}_S)$ and $\mathrm{Lin}(-, S): \mathcal{EM}(\mathcal{M}_S) \to \mathbf{Sets}^{\mathrm{op}}$ from Subsection 4.1, where $\mathcal{EM}(\mathcal{M}_S)$ is the category of modules over S. Here, S plays the role of D of Subsection 4.1 and the hom-functor is now the linear mappings functor $\mathrm{Lin}(-, S)$. The endofunctors on **Sets** used for this lifting via duality are $1 + A \times (-)$ and $S \times (-)^A$.

The following is a very simple example of determinization, for $S = \mathbb{N}$.



In general, the transitions of the determinization of the example weighted automaton c are given by $k_1x_1 + k_2x_2 + k_3x_3 \xrightarrow{a} (k_1 + k_2)x_2 + (2k_1 + k_3)x_3$ and the termination weight of a state $k_1x_1 + k_2x_2 + k_3x_3$ equals $k_1 + k_2 + k_3$.

The trace map $\mathrm{tr}(c): X \to \mathbb{N}^{A^\star}$ associates with each state x of the original weighted automaton the weighted language that it accepts. For instance, in the example above, we have $\mathrm{tr}(c)(x_1)(\langle\rangle) = 1$ and $\mathrm{tr}(c)(x_1)(a^n) = 3$ for $n \geq 1$.

Remark 3. More specifically, we can consider weighted automata with weights over a field $\mathbb{F}$, which are coalgebras for the functor $\mathcal{M}_{\mathbb{F}}(1 + A \times (-))$, where $\mathcal{M}_{\mathbb{F}}$ is the free vector space monad. This monad is special: the Kleisli category $\mathcal{K}\ell(\mathcal{M}_{\mathbb{F}})$ and the category $\mathcal{EM}(\mathcal{M}_{\mathbb{F}})$ are equivalent, since each vector space has a basis. They are the category **Vec** of vector spaces and linear maps over $\mathbb{F}$. The determinization $\widehat{E}(c)$ of a weighted automaton c coincides then with the *linear weighted automaton* of [8].

In the remainder of this section we consider the quantum walks example from [21], which can be described as a coalgebra $c: \mathbb{Z} + \mathbb{Z} \to \mathcal{M}_{\mathbb{C}}(\mathbb{Z} + \mathbb{Z})$, where the state space $\mathbb{Z} + \mathbb{Z}$ represents positions on a line $\mathbb{Z}$, with a direction (namely $\uparrow$ for the left component in $\mathbb{Z} + \mathbb{Z}$ and $\downarrow$ for the other, downwards direction). This description only involves the (unitary) transition function given by a superposition of left and right steps. Here we adapt this example a little bit so that we can compute the resulting probabilities via traces. We take the functor $F(X) = (\mathbb{Z} + \mathbb{Z}) + X$, and coalgebra $c: \mathbb{Z} + \mathbb{Z} \to \mathcal{M}_{\mathbb{C}}((\mathbb{Z} + \mathbb{Z}) + (\mathbb{Z} + \mathbb{Z})) = \mathcal{M}_{\mathbb{C}}(F(\mathbb{Z} + \mathbb{Z}))$ given by:

$$c(\uparrow k) = 1\ell(k) + \frac{1}{\sqrt{2}} \uparrow (k-1) + \frac{1}{\sqrt{2}} \downarrow (k+1)$$

$$c(\downarrow k) = 1r(k) + \frac{1}{\sqrt{2}} \uparrow (k-1) - \frac{1}{\sqrt{2}} \downarrow (k+1).$$

The right-hand-side of these equations is a formal sum over elements of the set $F(\mathbb{Z} + \mathbb{Z}) = (\mathbb{Z} + \mathbb{Z}) + (\mathbb{Z} + \mathbb{Z})$. What is possibly confusing is that in this quadruple coproduct of integers the left part $\mathbb{Z} + \mathbb{Z}$ is used for output, and the right part $\mathbb{Z} + \mathbb{Z}$ as state, for further computation. In these definitions the first parts $\ell(k)$ and $r(k)$ are the left and right part of this output. The second part involves multiplication with scalars $\pm\frac{1}{\sqrt{2}} \in \mathbb{C}$ and elements $\uparrow (k \pm 1), \downarrow (k \pm 1)$ in the right (state) component of $(\mathbb{Z} + \mathbb{Z}) + (\mathbb{Z} + \mathbb{Z})$.

As machine functor we take $M(X) = \mathcal{M}_{\mathbb{C}}(\mathbb{Z} + \mathbb{Z}) \times X$, with label set $A = 1$, and with final coalgebra $Z = \mathcal{M}_{\mathbb{C}}(\mathbb{Z} + \mathbb{Z})^{\mathbb{N}}$ of streams. The extension natural transformation follows from additivity of the multiset monad $\mathcal{M}_{\mathbb{C}}$ (like in (2), see [12]) with κ_1 and κ_2 denoting the coproduct injections:

$$\begin{array}{ccc} \mathcal{M}_{\mathbb{C}}\big((\mathbb{Z} + \mathbb{Z}) + X\big) & \xrightarrow[\cong]{\mathfrak{e} = \lambda\varphi.\ \langle \varphi \circ \kappa_1, \varphi \circ \kappa_2 \rangle} & \mathcal{M}_{\mathbb{C}}(\mathbb{Z} + \mathbb{Z}) \times \mathcal{M}_{\mathbb{C}}(X) \\ \| & & \| \\ \mathcal{M}_{\mathbb{C}}\big(F(X)\big) & & M\big(\mathcal{M}_{\mathbb{C}}(X)\big) \end{array}$$

The coalgebra $\widehat{E}(c): \mathcal{M}_{\mathbb{C}}(\mathbb{Z} + \mathbb{Z}) \to \mathcal{M}_{\mathbb{C}}(\mathbb{Z} + \mathbb{Z}) \times \mathcal{M}_{\mathbb{C}}(\mathbb{Z} + \mathbb{Z})$ in the category of vector spaces over $\mathbb{C}$, resulting from Theorem 2, gives rise to a trace map $\mathrm{tr}(c): \mathbb{Z} + \mathbb{Z} \to \mathcal{M}_{\mathbb{C}}(\mathbb{Z} + \mathbb{Z})^{\mathbb{N}}$. Thus, for the initial (upwards) state $\uparrow 0 \in \mathbb{Z} + \mathbb{Z}$ we get the probability $P(n, k)$ of ending up after n steps at position $k \in \mathbb{Z}$ on the line via the Born rule:

$$P(n, k) = \big|\mathrm{tr}(c)(\uparrow 0)(n)(\ell k)\big|^2 + \big|\mathrm{tr}(c)(\uparrow 0)(n)(rk)\big|^2.$$

These probabilities are computed in an ad hoc manner in [21].

8. Discussion

In this paper, we have systematically studied trace semantics, bringing together two perspectives: the coalgebraic trace semantics of [18] and the coalgebraic language equivalence via a determinization construction of [36]. Having the two perspectives together enables us to extend the class of systems that fits the framework of [18], while guaranteeing that the coalgebraic trace semantics from [18] fits in the current setting (Proposition 5). We illustrated the whole approach with several non-trivial examples, including weighted automata, quantum walks, simple Segala systems, and pushdown automata.

Our set-up has a certain overlap with [3], but the focus there is on getting coincidence of initial algebras and final coalgebras in categories $\mathcal{EM}(T)$, using stronger assumptions than here, namely commutativity of endofunctors $TF \cong GT$, see Section 6, where we only have a law $TF \Rightarrow GT$.

Acknowledgments

We are grateful to the anonymous referees for valuable comments. The work of Alexandra Silva is partially funded by the ERDF through the Programme COMPETE and by the Portuguese Foundation for Science and Technology, project Ref. FCOMP-01-0124-FEDER-020537 and SFRH/BPD/71956/2010.

References

[1] J.M. Autebert, J. Berstel, L. Boasson, et al., Context-free languages and pushdown automata, Handb. Form. Lang. 1 (1997) 111–174.
[2] S. Awodey, Category Theory, Oxford Logic Guides, OUP, Oxford, 2010.
[3] A. Balan, A. Kurz, On coalgebras over algebras, Theor. Comput. Sci. 412 (38) (2011) 4989–5005.
[4] M. Barr, Ch. Wells, Toposes, Triples and Theories, Springer, Berlin, 1985, Revized and corrected version available from www.cwru.edu/artsci/math/wells/pub/ttt.html.
[5] N. Benton, J. Hughes, E. Moggi, Monads and effects, in: G. Barthe, P. Dybjer, L. Pinto, J. Saraiva (Eds.), APPSEM, in: Lect. Notes Comput. Sci., vol. 2395, Springer, 2000, pp. 42–122.
[6] Filippo Bonchi, Marcello M. Bonsangue, Georgiana Caltais, Jan J.M.M. Rutten, Alexandra Silva, Final semantics for decorated traces, Electron. Notes Theor. Comput. Sci. 286 (2012) 73–86.
[7] F. Borceux, Handbook of Categorical Algebra, Encycl. Math. Appl., vols. 50, 51 and 52, Cambridge Univ. Press, 1994.
[8] M. Boreale, Weighted bisimulation in linear algebraic form, in: Proc. of CONCUR'09, in: Lect. Notes Comput. Sci., vol. 5710, Springer, 2009, pp. 163–177.
[9] J.A. Brzozowski, E. Leiss, Finite automata, and sequential networks, Theor. Comput. Sci. 10 (1980) 19–35.
[10] P. Castro, P. Panangaden, D. Precup, Equivalence relations in fully and partially observable Markov decision processes, in: Proc. IJCAI 2009, 2009, pp. 1653–1658.
[11] A.K. Chandra, D.C. Kozen, L.J. Stockmeyer, Alternation, J. Assoc. Comput. Mach. 28 (1981) 114–133.
[12] D. Coumans, B. Jacobs, Scalars, monads and categories, in: C. Heunen, M. Sadrzadeh (Eds.), Compositional Methods in Physics and Linguistics, Oxford Univ. Press, 2012.
[13] S. Crafa, F. Ranzato, A spectrum of behavioral relations over LTSs on probability distributions, in: Proc. CONCUR 2011, in: Lect. Notes Comput. Sci., vol. 6901, 2011, pp. 124–139.
[14] K. Culik II, J. Karhumaki, Finite automata computing real functions, SIAM J. Comput. 23 (4) (1994) 789–814.
[15] Y. Deng, R. van Glabbeek, M. Hennessy, C. Morgan, Characterising testing preorders for finite probabilistic processes, Log. Methods Comput. Sci. 4 (4) (2008).
[16] E. Doberkat, Eilenberg–Moore algebras for stochastic relations, Inf. Comput. 204 (12) (2006) 1756–1781.
[17] Z. Ésik, W. Kuich, An algebraic generalization of omega-regular languages, in: MFCS, in: Lect. Notes Comput. Sci., vol. 3153, 2004, pp. 648–659.
[18] I. Hasuo, B. Jacobs, A. Sokolova, Generic trace theory via coinduction, Log. Methods Comput. Sci. 4 (2007) 11.
[19] H. Hermanns, A. Parma, R. Segala, B. Wachter, L. Zhang, Probabilistic logical characterization, Inf. Comput. 209 (2) (2011) 154–172.
[20] J.E. Hopcroft, R. Motwani, J.D. Ullman, Introduction to Automata Theory, Languages, and Computation, 3rd edition, Wesley, 2006.
[21] B. Jacobs, Coalgebraic walks, in quantum and Turing computation, in: M. Hofmann (Ed.), FoSSaCS, in: Lect. Notes Comput. Sci., vol. 6604, Springer, 2011, pp. 12–26.
[22] B. Jacobs, Introduction to Coalgebra. Towards Mathematics of States and Observations, Book, in preparation. Version 2 available from www.cs.ru.nl/B.Jacobs/CLG/JacobsCoalgebraIntro.pdf, 2012.
[23] B. Jacobs, A. Silva, A. Sokolova, Trace semantics via determinization, in: L. Schröder, D. Pattinson (Eds.), Coalgebraic Methods in Computer Science, CMCS 2012, in: Lect. Notes Comput. Sci., vol. 7399, Springer, 2012, pp. 109–129.
[24] B. Jacobs, A. Sokolova, Exemplaric expressivity of modal logics, J. Log. Comput. 20 (5) (2010) 1041–1068.
[25] P.T. Johnstone, Adjoint lifting theorems for categories of algebras, Bull. Lond. Math. Soc. 7 (1975) 294–297.
[26] Ch. Kissig, A. Kurz, Generic trace logics, arXiv:1103.3239 [org/abs], 2011.
[27] A. Kock, Strong functors and monoidal monads, Arch. Math. 23 (1972).
[28] E.G. Manes, Algebraic Theories, Springer, Berlin, 1974.
[29] S. Mac Lane, Categories for the Working Mathematician, Springer, Berlin, 1971.
[30] Stefan Milius, Thorsten Palm, Daniel Schwencke, Complete iterativity for algebras with effects, in: Alexander Kurz, Marina Lenisa, Andrzej Tarlecki (Eds.), CALCO, in: Lect. Notes Comput. Sci., vol. 5728, Springer, 2009, pp. 34–48.
[31] E. Moggi, Notions of computation and monads, Inf. Comput. 93 (1) (1991) 55–92.
[32] Philip S. Mulry, Lifting theorems for Kleisli categories, in: Stephen D. Brookes, Michael G. Main, Austin Melton, Michael W. Mislove, David A. Schmidt (Eds.), MFPS, in: Lect. Notes Comput. Sci., vol. 802, Springer, 1993, pp. 304–319.
[33] A. Parma, R. Segala, Logical characterizations of bisimulations for discrete probabilistic systems, in: Proc. FoSSaCS 2007, in: Lect. Notes Comput. Sci., vol. 4423, 2007, pp. 287–301.
[34] D. Pavlović, M. Mislove, J. Worrell, Testing semantics: Connecting processes and process logics, in: M. Johnson, V. Vene (Eds.), Algebraic Methods and Software Technology, in: Lect. Notes Comput. Sci., vol. 4019, Springer, 2006, pp. 308–322.
[35] R. Segala, N.A. Lynch, Probabilistic simulations for probabilistic processes, in: Proc. Concur'94, in: Lect. Notes Comput. Sci., vol. 836, 1994, pp. 481–496.
[36] A. Silva, F. Bonchi, M. Bonsangue, J. Rutten, Generalizing the powerset construction, coalgebraically, in: Proc. FSTTCS 2010, in: LIPIcs. Leibniz Int. Proc. Inform., vol. 8, 2010, pp. 272–283.

[37] A. Silva, F. Bonchi, M. Bonsangue, J. Rutten, Generalizing determinization from automata to coalgebras, Log. Methods Comput. Sci. 9 (1) (2013).
[38] A. Silva, A. Sokolova, Sound and complete axiomatization of trace semantics for probabilistic systems, in: Proc. MFPS 2011, in: Electron. Notes Theor. Comput. Sci., vol. 276, 2011, pp. 291–311.
[39] D. Varacca, G. Winskel, Distributing probability over non-determinism, Math. Struct. Comput. Sci. 16 (1) (2006) 87–113.