String Diagrammatic Trace Theory

Matthew Earnshaw
Department of Software Science, Tallinn University of Technology, Estonia

Paweł Sobociński
Department of Software Science, Tallinn University of Technology, Estonia

Abstract

We extend the theory of formal languages in monoidal categories to the multi-sorted, symmetric case, and show how this theory permits a graphical treatment of topics in concurrency. In particular, we show that Mazurkiewicz trace languages are precisely *symmetric monoidal languages* over *monoidal distributed alphabets*. We introduce *symmetric monoidal automata*, which define the class of regular symmetric monoidal languages. Furthermore, we prove that Zielonka's asynchronous automata coincide with symmetric monoidal automata over monoidal distributed alphabets. Finally, we apply the string diagrams for symmetric premonoidal categories to derive serializations of traces.

2012 ACM Subject Classification Theory of computation → Concurrency; Theory of computation → Formal languages and automata theory; Theory of computation → Categorical semantics

Keywords and phrases symmetric monoidal categories, Mazurkiewicz traces, asynchronous automata

Digital Object Identifier 10.4230/LIPIcs.MFCS.2023.43

Funding Estonian Research Council grant PRG1210 and the European Union under Grant Agreement No. 101087529. Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union or European Research Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

Acknowledgements We thank Chad Nester, Mario Román, and Niels Voorneveld for discussions.

1 Introduction

Monoidal languages [12] generalize formal languages of words to formal languages of *string diagrams*. String diagrams [16, 29] are a graphical representation of morphisms in *monoidal categories*. Monoidal categories can be considered *2-dimensional monoids* [6]: just as monoids are categories with one object, whose morphisms are elements of the monoid, (strict) monoidal categories can be defined as 2-categories with one object. Accordingly, *monoidal languages* are subsets of morphisms in free monoidal categories, just as word languages are subsets of free monoids. *Regular* monoidal languages are those specifiable by finitary grammars or automata. Our paper [12] introduced these devices and examined properties of languages in single-sorted, planar monoidal categories. These include regular languages of words and trees, but also languages of *planar* string diagrams that are neither linear nor tree-like.

In this paper, motivated by concurrency theory, we extend this theory to *coloured props*: multi-sorted monoidal categories with symmetries (Section 2). The resulting theory of *symmetric* monoidal languages (Section 3) captures languages of diagrams having multiple colours of string and in which strings may cross, permitting non-planar diagrams. In terms of concurrency, colours represent different *runtimes*, or *threads of execution*.

Indeed, in Section 4 we show that Mazurkiewicz trace languages [21] are exactly symmetric monoidal languages over alphabets of a particular shape called *monoidal distributed alphabets*. In Section 5 we introduce automata for symmetric monoidal languages, defining the class of *regular* symmetric monoidal languages. Then, in Section 6 we show that these are exactly the asynchronous automata of Zielonka [32] when instantiated over monoidal distributed alphabets. Finally, in Section 7 we use the algebra of symmetric *premonoidal* categories to show how serialization of traces can be treated string-diagrammatically.

© Matthew Earnshaw and Paweł Sobociński;
licensed under Creative Commons License CC-BY 4.0
48th International Symposium on Mathematical Foundations of Computer Science (MFCS 2023).
Editors: Jérôme Leroux, Sylvain Lombardy, and David Peleg; Article No. 43; pp. 43:1–43:15
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Related work

Our previous work [12] introduced monoidal languages in the planar, single-sorted case; that is, languages of morphisms in free *pros*. Similar languages of graphs were studied by Bossut [5], but their underlying algebra was not made explicit. Here, we again leverage the algebraic perspective, extending our theory to symmetric multi-sorted monoidal categories (props).

In the introduction to Joyal & Street's foundational work on string diagrams for monoidal categories [16], it is suggested that string diagrams have a connection to the *heaps* of Viennot [30]. Heaps are known to be equivalent to Mazurkiewicz trace monoids (also known as partially commutative monoids) [17], but a precise formulation of the suggested relation with string diagrams has not appeared in the literature until now.

The notion of dependence graph [13] has also been used to give a topological presentation of Mazurkiewicz traces. Our use of the algebra of monoidal categories, rather than graphs, has various advantages. For example, we can apply our language theory for monoidal categories to traces, and we see notions such as asynchronous automata arise naturally from this. It also suggests generalizations of trace languages, in particular going beyond the case of atomic actions (Remark 23). Finally, it brings our work into proximity with the semantics of Petri nets and other formalisms for concurrency based on monoidal categories [2, 24].

2 Monoidal Graphs, Props and their String Diagrams

In this section we recall the basic definitions used in the following, including the specific flavour of monoidal categories known as *props* [20], along with their string diagrams [16, 29]. Just as a category can be presented by a directed graph, (strict) monoidal categories can be presented by *monoidal graphs*, a kind of multi-input, multi-output directed graph.

▶ **Definition 1.** *A monoidal graph $\mathcal{G}$ is a set $B_\mathcal{G}$ of boxes, a set $S_\mathcal{G}$ of sorts, and functions $s, t : B_\mathcal{G} \rightrightarrows {S_\mathcal{G}}^*$ to the free monoid over $S_\mathcal{G}$, giving source and target boundaries of each box.*

The alphabets of monoidal languages will be finite monoidal graphs: those in which $B_\mathcal{G}$ and $S_\mathcal{G}$ are both finite sets. In fact, since we are interested in finite state machines over finite alphabets, we will work exclusively with finite monoidal graphs. Diagrammatically, a (finite) monoidal graph can be pictured as a collection of boxes, labelled by elements of $B_\mathcal{G}$ with strings entering on the left and exiting on the right, labelled by sorts given by the source and target functions. For example, the following depicts the monoidal graph $\mathcal{G}$ with $B_\mathcal{G} = \{\gamma, \gamma'\}, S_\mathcal{G} = \{A, B\}, s(\gamma) = AB, t(\gamma) = ABA, s(\gamma') = A, t(\gamma') = BB$:

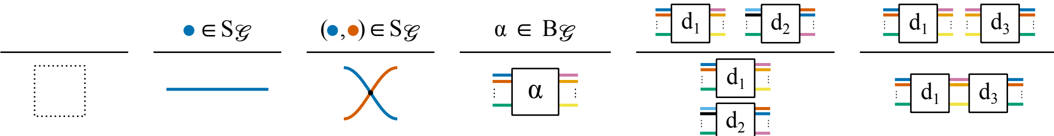
A, B → [γ] → A, B, A   A → [γ'] → B, B

Sorts of a monoidal graph are sometimes called *colours*, since we could equally use different colours of string to represent different sorts, and we shall do so in places below. For a box $\gamma \in B_\mathcal{G}$ we call $s(\gamma)$ and $t(\gamma)$ the *arity* and *coarity* of γ, respectively, and write $\gamma : s(\gamma) \to t(\gamma)$. We will also call γ considered together with its arity and coarity a *generator*.

Monoidal graphs are generating data for monoidal categories. Recall that a *strict monoidal category* is a category $\mathcal{C}$, equipped with a functor $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ (the *monoidal product*) and a unit object $I \in \mathcal{C}$, such that $\otimes$ is associative and unital. A strict monoidal category is *symmetric* if there is a natural family of *symmetry morphisms* $\sigma_{A,B} : A \otimes B \to B \otimes A$, for each pair of sorts, satisfying $\sigma_{B,A} \circ \sigma_{A,B} = 1_{A \otimes B}$. The monoidal product turns the sets of objects and morphisms in $\mathcal{C}$ into monoids. A *prop* is a symmetric strict monoidal category

whose monoid of objects is a free monoid.[1] While the above data can be intimidating to the non-expert, the free prop $\mathscr{F}_\times\mathcal{G}$ on a monoidal graph $\mathcal{G}$ can be described in an intuitive and straightforward way: its arrows are the *string diagrams* generated by $\mathcal{G}$.

▶ **Definition 2.** *The free prop $\mathscr{F}_\times\mathcal{G}$ on a monoidal graph $\mathcal{G}$ has monoid of objects $S_\mathcal{G}^*$ and morphisms* string diagrams *inductively defined as follows: Left to right: the empty diagram*



is a diagram; for every sort, the string on that sort is a diagram; for every pair of sorts, the symmetric braiding is a diagram; the diagram for every generator α is a diagram; for any two diagrams their vertical juxtaposition is a diagram; and for any two diagrams with matching right and left boundaries, the diagram obtained by joining the matching wires is a diagram (their composition). The monoidal product is given on objects by concatenation, on diagrams by juxtaposition, and the unit is the empty word.

The idea is simple: we treat generators like circuit components, and we have a supply of wires (identity morphisms). We also have the ability to cross wires, without tangling them; we do not distinguish over-crossings from under-crossings. A string diagram is then just any (open) circuit that we can build. This notation is sound and complete: an equation between morphisms of strict monoidal categories follows from their axioms if and only if it holds between string diagrams up to planar isotopy [16]. Working with string diagrams rather than the usual term syntax for morphisms is more intuitive, and leads to shorter proofs as the structural equations hold automatically: for example, interchange of morphisms (Figure 1, left), unbraiding of symmetries (centre), and sliding of morphisms past symmetries (right).

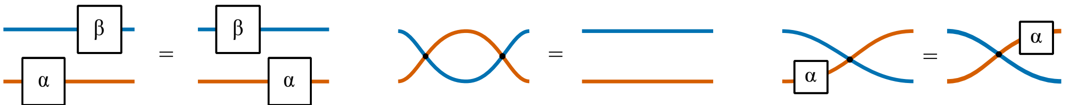


Figure 1 These pairs of string diagrams are equal, reflecting the functoriality of ⊗ (*interchange*), inverses of symmetries, and naturality of symmetries, respectively.

▶ **Definition 3.** *A morphism of monoidal graphs $\varphi : \mathcal{H} \to \mathcal{G}$ is given by functions $B_\varphi : B_\mathcal{H} \to B_\mathcal{G}$ and $S_\varphi : S_\mathcal{H} \to S_\mathcal{G}$ compatible with source and target functions: $S_\varphi^* \circ s = s \circ B_\varphi$ and $S_\varphi^* \circ t = t \circ B_\varphi$, where S_φ^* is the unique monoid homomorphism determined by S_φ.*

Morphisms of monoidal graphs freely generate morphisms of props: strict monoidal functors preserving sorts. Every prop has an underlying monoidal graph whose boxes are all the morphisms of the prop. This extends to an adjunction $\mathscr{F}_\times \dashv \mathscr{U}$ between the categories of monoidal graphs and props, where $\mathscr{U}$ takes the underlying monoidal graph of a prop [16].

Monoidal categories have been applied to the study of both computing and physical processes [8, 9, 18, 25]. In these contexts, the monoidal product represents parallel composition of processes, and interchange reflects the *independence* of processes running in parallel. This is the main feature of monoidal categories that we will leverage in our representation of *traces* (Section 4). The use of multi-sorted props will allow fine-grained control of interchange.

[1] Some literature takes prop to mean that the monoid of objects is generated by a single object (and so isomorphic to $\mathbb{N}$), using the term *coloured* prop for the general case above.

3 Symmetric Monoidal Languages

Our paper [12] treated the case of languages, grammars and automata over single-sorted *pros* (strict monoidal categories *without* symmetries), corresponding to languages of *planar* string diagrams with one string colour. In this section we introduce the *multi-sorted* (or "coloured") *symmetric monoidal languages*, which will be needed in the following to extend monoidal language theory to trace theory. In Section 5 we introduce the corresponding automata.

Just as a classical formal language is a subset of a free monoid, a symmetric monoidal language is a subset of morphisms in a free prop:

▶ **Definition 4.** *Let* Γ *be a finite monoidal graph. A* symmetric monoidal language *over* Γ *is a set of morphisms in the free prop* $\mathscr{F}_\times\Gamma$ *over* Γ.

A morphism of finite directed graphs $G \to \Sigma$, where Σ is a graph with one vertex, amounts to a labelling of the edges of G by edges of Σ. This is the starting point of Walters' definition of regular grammar [31], which inspires the following definition:

▶ **Definition 5.** *A regular monoidal grammar is a morphism of finite monoidal graphs.*

For a regular monoidal grammar $M \xrightarrow{\varphi} \Gamma$, the monoidal graph Γ is the *alphabet*, and the generators of M, with their labelling by φ, correspond to production rules: see Example 7.

In the classical setting of word languages, a morphism of finite directed *graphs* $G \to \Sigma$ determines a *regular language* over Σ once we specify initial and final state vertices in G. In a regular monoidal grammar $M \to \Gamma$, the "vertices" of M are words over S_M, leading to various natural choices of boundary condition (Remark 8). In this paper, we will take initial and final words over S_M^*. Specifying these words defines the symmetric monoidal language of the grammar (Definition 9), and we define the languages arising in this way to be the regular symmetric monoidal languages.

We illustrate these definitions with some pedagogical toy examples. In the remaining sections of this paper, we turn to our extended application in concurrency, and we shall see that Mazurkiewicz trace languages are a natural example of symmetric monoidal languages.

▶ **Example 6.** Let $\varphi : M \to \Gamma$ be the regular monoidal grammar where M and Γ have a single sort $(\bullet)$ and no boxes, with $S_\varphi(\bullet) = \bullet$, and initial and final states $n \in S_M^* \cong \mathbb{N}$. Then the symmetric monoidal language of this grammar is the set of *permutations* of n wires: morphisms consisting only of symmetries and identities.

Props have been used to give syntax and semantics for various kinds of *signal flow graph* and *circuit diagrams* [1, 3, 4]. Intuitively, props are well suited for this purpose since wires may freely cross in a circuit.

▶ **Example 7.** We give a regular monoidal grammar for the syntax of (open) circuits with $n \geqslant 0$ capacitors in series with a single voltage source (Figure 2). The alphabet Γ has a single sort, and boxes four circuit components (Figure 2, left). The monoidal graph M has four sorts $\{S, A, B, C\}$ and four boxes $s : S \to AB, c : A \to A, v : B \to C, s' : AC \to S$. S_φ maps the four sorts to the single sort of Γ, and B_φ maps each box to a circuit component. We can draw the grammar $\varphi : M \to \Gamma$ in a single diagram by drawing the graph for M but replacing each box b with its image under the grammar morphism $B_\varphi(b)$ (Figure 2, centre). The initial and final languages are the single state $\{S\}$. Intuitively, the symmetric monoidal language determined by the grammar is all of the string diagrams $S \to S$ that can be built using the "sorted" boxes of Γ, then forgetting the sorts.

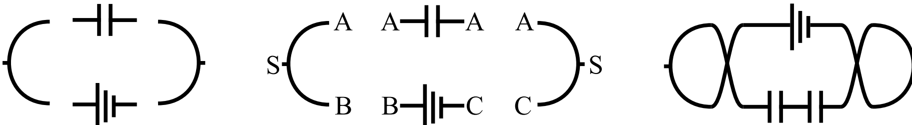


Figure 2 (Left) The alphabet Γ, giving syntax for circuits. (Centre) A regular monoidal grammar over Γ. (Right) An element of the regular symmetric monoidal language determined by this grammar.

▶ Remark 8. As mentioned above, there are various possible choices for the "initial and final states" of a monoidal grammar. In our previous paper [12], we took the empty word, giving languages of scalar string diagrams (i.e. no "dangling wires"): this neatly generalizes tree grammars. More generally, one can take initial and final *regular languages* of states over S_M, as considered by Bossut [5].

The free prop construction can be used to concisely describe the symmetric monoidal language of a regular monoidal grammar, defining the class of *regular symmetric monoidal languages*:

▶ **Definition 9.** *Let* $(\varphi : M \to \Gamma, I, F)$ *be a regular monoidal grammar equipped with regular languages* $I, F \subseteq S_M^*$*. This determines a symmetric monoidal language by taking the image of the set of morphisms* $\bigcup_{i\in I, f\in F} \mathscr{F}_{\times}M(i, f)$ *under* $\mathscr{F}_{\times}\varphi$*, giving a set of morphisms in* $\mathscr{F}_{\times}\Gamma$*. The languages arising in this way are defined to be the* regular *symmetric monoidal languages.*

In this paper, we will only need the case where I, F consist of single words. The slogan for the general case is that 2-dimensional regular languages have 1-dimensional regular boundaries. In Section 5, we will see that regular symmetric monoidal languages may equivalently be specified by *non-deterministic monoidal automata*.

▶ Remark 10. A regular monoidal grammar determines not only a regular symmetric monoidal language, but also a language in any algebraic structure generated by monoidal graphs, including planar monoidal categories (treated in [12]), and premonoidal categories (which we will use in Section 7). This is analogous to the way in which a finite labelled directed graph may generate both a subset of a free monoid, but also a subset of a free group, by freely adding inverses to the graph. Moreover, many properties of planar regular monoidal languages such as their closure properties proved in [12] only use grammars, and hence the same proofs work for languages in these other algebras.

4 Mazurkiewicz Trace Languages as Symmetric Monoidal Languages

The theory of Mazurkiewicz traces [10, 21, 23] provides a simple but powerful model of concurrent systems. Traces are a generalization of *words* in which specified pairs of letters can commute. If we think of letters as corresponding to atomic *actions*, then commuting letters reflect the *independence* of those particular actions and so their possible concurrent execution: ab is observationally indistinguishable from ba if a and b are independent.

In this section, we show that trace languages are symmetric monoidal languages over monoidal graphs of a particular form that we call *monoidal distributed alphabets*. In Section 5 we introduce *symmetric monoidal automata*, which operationally characterize the regular symmetric monoidal languages. In Section 6 we turn to *asynchronous automata* [32], a well-known model accepting exactly the *recognizable* trace languages, and show that these automata are precisely symmetric monoidal automata over monoidal distributed alphabets.

4.1 Independence and distribution

We recall some definitions from Mazurkiewicz trace theory, before recasting them in terms of monoidal languages. Fix a finite set Σ, an alphabet thought of as a set of atomic actions.

▶ **Definition 11.** *An* independence relation *on Σ is a symmetric, irreflexive relation I. The induced* dependence relation*, D_I is the complement of I.*

▶ **Definition 12.** *For I an independence relation, let $\equiv_I$ be the least congruence on Σ^* such that $\forall a, b$: $(a, b) \in I \implies ab \equiv_I ba$. The quotient monoid $\mathcal{T}(\Sigma, I) := \Sigma^*/\equiv_I$ is the* trace monoid*.*

▶ **Definition 13.** *A (Mazurkiewicz)* trace language *over (Σ, I) is a subset of the trace monoid $\mathcal{T}(\Sigma, I)$.*

An element of $\mathcal{T}(\Sigma, I)$ or *trace over* (Σ, I) is thus an equivalence class of words up to commutation of independent letters. A trace language may be thought of as the set of possible observations of a concurrent system's behaviour, in which independent letters stand for actions which may occur concurrently. Independence relations correspond to *distributions*:

▶ **Definition 14** ([23])**.** *A* distribution *of an alphabet Σ is a finite tuple of non-empty alphabets $(\Sigma_1, ..., \Sigma_k)$ such that $\bigcup_{i=1}^{k} \Sigma_i = \Sigma$.*

▶ **Proposition 15** ([23])**.** *A distribution of Σ corresponds to a function $\mathsf{loc} : \Sigma \to \mathscr{P}^+(\{1, ..., k\}) : \sigma \mapsto \{i \mid \sigma \in \Sigma_i\}$.*

Such a function gives the set of "locations" of each action $\sigma \in \Sigma$. In terms of concurrency, we can consider this to be a set of memory locations, threads of execution, or runtimes in which σ participates. In particular, every action has a non-empty set of locations.

A well-known construction [23] allows us to move between independence relations and distributions: locations correspond to maximal cliques in the graph of the dependency relation. We recall this construction in the proof of Proposition 16, which refines this correspondence.

Let Ind_Σ be the poset of independence relations on Σ, with order the inclusion of relations. Similarly, define a preorder Dist_Σ on distributions by $(\Sigma_1, ..., \Sigma_p) \leqslant (\Sigma'_1, ..., \Sigma'_q)$ iff for each pair of distinct elements $a, b \in \Sigma$, if there exists $1 \leqslant j \leqslant q$ such that Σ'_j contains both a and b, then there exists an Σ_i containing both a and b. Finally, quotient this preorder by taking distributions to be equal up to permutation.

▶ **Proposition 16.** *There is a Galois insertion $\mathsf{Ind}_\Sigma \hookrightarrow \mathsf{Dist}_\Sigma$.*

Proof. We construct an injective monotone function $i : \mathsf{Ind}_\Sigma \to \mathsf{Dist}_\Sigma$. Let an independence relation I over Σ be given, with induced dependence relation D_I. Construct the undirected *dependency graph*: vertices are elements of Σ and there is an edge (a, b) for every $(a, b) \in D_I$. Choose an ordering of maximal cliques of D_I, and define a distributed alphabet by taking Σ_i to be the elements of Σ in the maximal clique i. Different orderings give the same distribution up to permutation, and so the same element of Dist_Σ. This is injective since distinct independence relations induce distinct dependency graphs. It is monotone since if $I \subseteq I'$ then the dependency graph D_I is at least as connected as $D_{I'}$, so if a, b both belong to a maximal clique of $D_{I'}$ then they will both belong to a maximal clique of D_I.

We construct a monotone function $r : \mathsf{Dist}_\Sigma \to \mathsf{Ind}_\Sigma$. Let $(\Sigma_1, ..., \Sigma_k)$ be a distribution. Define a relation I by $(a, b) \in I \iff \mathsf{loc}(a) \cap \mathsf{loc}(b) = \varnothing$. This is irreflexive and symmetric, and so an independence relation. r is also clearly well-defined and monotone. Finally it is easy to check that $r \circ i : \mathsf{Ind}_\Sigma \to \mathsf{Ind}_\Sigma$ is the identity. ◀

Put otherwise, though the same independence relation may be induced by many different distributions, independence relations correspond bijectively with the distributions in the image of $i \circ r$, that is, the distributions obtained via the maximal clique construction.

4.2 Symmetric monoidal languages over monoidal distributed alphabets

We now turn to the interpretation of these notions in terms of symmetric monoidal languages. A distribution can be seen as a monoidal graph in which sorts are the locations (runtimes).

▶ **Definition 17.** *A monoidal distributed alphabet is a finite monoidal graph* Γ *with the following properties:*

- Γ *has set of sorts a finite ordinal* $S_\Gamma = \{1 < 2 < \ldots < k\}$ *for* $k \geqslant 1$*,*
- *sorts* $i \in S_\Gamma$ *appear in order in the sources and targets of each generator* $\gamma \in B_\Gamma$*,*
- *each sort* $i \in S_\Gamma$ *appears at most once in each source and target,*
- *for each generator* $\gamma \in B_\Gamma$*, the sources and targets are non-empty and equal:* $s(\gamma) = t(\gamma)$*.*

In brief, every generator in the alphabet is equipped with some set of runtimes, which serve as its source and target, and the runtimes are conserved. Figure 3 gives an example.

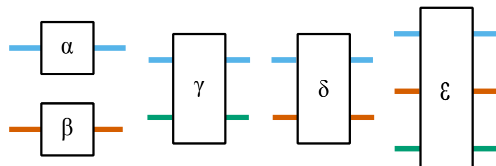


Figure 3 An example of a monoidal distributed alphabet. For example, δ and β are independent but γ and α are not. We use colours for clarity, here blue $= 1 <$ red $= 2 <$ green $= 3$.

This gives us a way of representing distributions as monoidal graphs and vice-versa, if the graph is a monoidal distributed alphabet. Following Proposition 15, we will use $\mathsf{loc}(\vdots\boxed{\gamma}\vdots)$ to mean the arity ($=$ coarity) of a generator $\vdots\boxed{\gamma}\vdots$. Since we choose a finite ordinal for the sorts, we have that:

▶ **Proposition 18.** *Distributed alphabets are in bijection with monoidal distributed alphabets.*

Since the *ordering* of the runtimes is ultimately not relevant to the structure of a trace, we should allow them to freely cross each other in our string diagrams: this is precisely what is enabled by taking the *symmetric* monoidal languages over these alphabets. We also need each runtime to appear once in each element of these languages, so we take the boundaries to be $1 \otimes \ldots \otimes n$, which we will write as $\begin{smallmatrix}1\\ \vdots\\ n\end{smallmatrix}$.

▶ **Definition 19.** *A* monoidal trace language *is a symmetric monoidal language of the form* $L \subseteq \mathscr{F}_\times\Gamma\left(\begin{smallmatrix}1\\ \vdots\\ n\end{smallmatrix}, \begin{smallmatrix}1\\ \vdots\\ n\end{smallmatrix}\right)$ *where* Γ *is a monoidal distributed alphabet.*

Figure 4 gives an example of an element in a monoidal trace language over the monoidal distributed alphabet in Figure 3. We call such morphisms *monoidal traces*, and indeed we shall see below that they are exactly Mazurkiewicz traces. The corresponding string diagram gives an intuitive representation of traces as topological objects.

We now show that monoidal trace languages correspond precisely to Mazurkiewicz trace languages (Theorem 22), by establishing an isomorphism of monoids between trace monoids and monoids of string diagrams generated by monoidal distributed alphabets. Fix a monoidal distributed alphabet Γ. Recall that endomorphism hom-sets in a category are monoids under composition, and that the hom-set $\mathscr{F}_\times\Gamma\left(\begin{smallmatrix}1\\ \vdots\\ n\end{smallmatrix}, \begin{smallmatrix}1\\ \vdots\\ n\end{smallmatrix}\right)$ has elements string diagrams $\begin{smallmatrix}1\\ \vdots\\ n\end{smallmatrix} \to \begin{smallmatrix}1\\ \vdots\\ n\end{smallmatrix}$ over Γ.

Figure 4 An example of a monoidal trace. β is independent of α and γ, but not δ or ϵ. Thus $\alpha\gamma\beta\delta\varepsilon$ and $\beta\alpha\gamma\delta\varepsilon$ are two possible serializations of this trace, corresponding to sliding β past α and γ in the string diagram. We use colours for sorts, blue = 1 < red = 2 < green = 3.

▶ **Lemma 20.** *The hom-set $\mathscr{F}_\times\Gamma\binom{1\;\;1}{\vdots,\vdots}{}_{n\;\;n}$ admits the following presentation as a monoid:*

- *Generators: For each $\vdots\boxed{\gamma}\vdots \in \Gamma$, the string diagram $N(\gamma) : 1 \otimes ... \otimes n \to 1 \otimes ... \otimes n$ built from symmetries, followed by $\vdots\boxed{\gamma}\vdots$ tensored with identities, followed by the inverse symmetry. See Figure 5 for an example.*
- *Equations: $N(\alpha) \mathbin{;} N(\beta) = N(\beta) \mathbin{;} N(\alpha) \iff \mathsf{loc}(\vdots\boxed{\alpha}\vdots) \cap \mathsf{loc}(\vdots\boxed{\beta}\vdots) = \varnothing$, where $\mathbin{;}$ denotes composition of string diagrams in diagrammatic (left-to-right) order.*

Proof. We construct an isomorphism between the monoids. Let $s \in \mathscr{F}_\times\Gamma\binom{1\;\;1}{\vdots,\vdots}{}_{n\;\;n}$ be a string diagram. We can use interchange (Figure 1) to impose a linear order of generators from left to right in the diagram, e.g. $\vdots\boxed{\gamma_1}\vdots, ..., \vdots\boxed{\gamma_n}\vdots$. This is called putting s in *general position*, by perturbing generators at the same horizontal position [16]. We then split the string diagram into a sequence of slices, each containing one generator. For a slice with right (or left) boundary $\overset{k_1}{\underset{k_n}{\vdots}}$, we can use the permutation $\overset{k_1}{\underset{k_n}{\vdots}} \to \overset{1}{\underset{n}{\vdots}}$ followed by its inverse (or vice-versa) to finally obtain s as a sequence $N(\gamma_1) \mathbin{;} ... \mathbin{;} N(\gamma_n)$. Any other possible sequence of generators is obtainable by repeatedly interchanging generators: this is possible if and only if their locations are disjoint. Consequently, this defines a function from $\mathscr{F}_\times\Gamma\binom{1\;\;1}{\vdots,\vdots}{}_{n\;\;n}$ to the monoid presented above. Given that, as argued above, the slicing construction is unique up to interchanging independent generators, this defines a homomorphism. Conversely, given a generator $N(\gamma)$ in the presentation, we map this to the same string diagram in $\mathscr{F}_\times\Gamma\binom{1\;\;1}{\vdots,\vdots}{}_{n\;\;n}$. Again, it follows from interchange that this extends to a homomorphism, inverse to that above. ◀

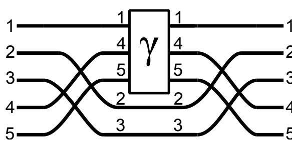


Figure 5 An example of a generator $N(\gamma)$ as in Lemma 20.

We now show that trace monoids are isomorphic to the endomorphism monoids $\mathscr{F}_\times\Gamma\binom{1\;\;1}{\vdots,\vdots}{}_{n\;\;n}$.

▶ **Lemma 21.** *Let I be an independence relation on an alphabet Σ, and Γ the monoidal distributed alphabet induced by the corresponding distribution (Proposition 18). Then there is an isomorphism of monoids $\mathcal{T}(\Sigma, I) \cong \mathscr{F}_\times\Gamma\binom{1\;\;1}{\vdots,\vdots}{}_{n\;\;n}$.*

Proof. We use the presentation of the endomorphism monoid given in Lemma 20. Define a homomorphism $\alpha : \mathscr{F}_\times\Gamma\binom{1\;\;1}{\vdots,\vdots}{}_{n\;\;n} \to \mathcal{T}(\Sigma, I)$ by mapping generators $N(\gamma) \mapsto [\gamma]$. Let $N(\gamma) \mathbin{;} N(\gamma') = N(\gamma') \mathbin{;} N(\gamma)$, then it follows $[\gamma\gamma'] = [\gamma'\gamma]$ in $\mathcal{T}(\Sigma, I)$, since the former holds

iff $\mathsf{loc}(\gamma) \cap \mathsf{loc}(\gamma') = \varnothing$, and so this extends to a homomorphism. Define a homomorphism $\beta : \mathcal{T}(\Sigma, I) \to \mathscr{F}_{\times}\Gamma\binom{1}{\vdots}{n}, \binom{1}{\vdots}{n}$ by mapping generators $[\gamma] \mapsto N(\gamma)$. $[\gamma\gamma'] = [\gamma'\gamma]$ holds iff $\mathsf{loc}(\gamma) \cap \mathsf{loc}(\gamma') = \varnothing$, iff $\mathsf{loc}(\boxed{\gamma}) \cap \mathsf{loc}(\boxed{\gamma'}) = \varnothing$, iff $N(\gamma) \fatsemi N(\gamma') = N(\gamma') \fatsemi N(\gamma)$. Finally it is clear that α and β are inverses, and so witness an isomorphism of monoids. ◀

The following theorem is now immediate: given a monoidal trace language $L \subseteq \mathscr{F}_{\times}\Gamma\left(\begin{smallmatrix}1\\ \vdots\\ n\end{smallmatrix}, \begin{smallmatrix}1\\ \vdots\\ n\end{smallmatrix}\right)$ we obtain a trace language $L' \subseteq \mathcal{T}(\Sigma, I)$ using the isomorphism, and vice-versa:

▶ **Theorem 22.** *Monoidal trace languages are exactly Mazurkiewicz trace languages.*

Lemma 21 also shows that composition of traces corresponds simply to concatenation of the corresponding monoidal traces. Diagrams like Figure 4 are commonplace in the trace literature [11, 32]. Theorem 22 gives a formal basis for these diagrams as elements of symmetric monoidal languages.

▶ Remark 23. The idea of monoidal categories with a runtime is made precise by string diagrams for the *effectful categories* of Román [28]. Free props over monoidal distributed alphabets, considered as monoidal categories with *multiple runtimes* suggest a further generalization of effectful categories, sketched as a setting for concurrency by Jeffrey [14, Section 9.4]. We return to this in Section 7, where effectful (premonoidal) categories will be used to equip a trace language with a new runtime that enforces a strict ordering of events.

5 Symmetric Monoidal Automata

Monoidal automata give an alternative specification of the class of regular monoidal languages: they are analogues of finite-state automata in which transitions have multiple inputs and multiple outputs. Our paper [12] introduced monoidal automata for single-sorted, planar monoidal languages. However, the same data specifies an acceptor for single-sorted symmetric monoidal languages, if we inductively extend to *props*, rather than planar monoidal categories.

In this section we introduce monoidal automata over *multi-sorted* monoidal graphs and show how these recognize (multi-sorted) symmetric monoidal languages. In Section 6, we will see that the asynchronous automata of Zielonka [32] are a natural class of symmetric monoidal automata: those over monoidal distributed alphabets.

▶ **Definition 24.** *A non-deterministic monoidal semi-automaton is:*
- *an input alphabet, given by a finite monoidal graph* Γ,
- *an family of non-empty, finite state sets* $\{Q_c\}_{c\in S_\Gamma}$ *indexed by the sorts of* Γ,
- *for each* $\gamma : c_1...c_n \to c'_1...c'_m$ *in* Γ*, a transition function* $\Delta_\gamma : \prod_{i=0}^n Q_{c_i} \to \mathscr{P}(\prod_{j=0}^m Q_{c'_j})$.

As noted in Section 3, there are several candidates for a notion of initial/final state. In the following, we take initial and final words i, f over $\prod Q_c$. A monoidal semi-automaton equipped with initial and final words turns it into a (non-deterministic) *monoidal automaton.*

For classical NFAs, the assignment $a \mapsto \Delta_a$ extends uniquely to a functor $\Sigma^* \to \mathsf{Rel}$, the inductive extension of the transition structure from letters to words. We can similarly extend monoidal automata to string diagrams. First, we define the codomain prop, $\mathsf{Rel}_{\Gamma,Q}$:

▶ **Definition 25.** *For a family of sets* $\{Q_c\}_{c\in S_\Gamma}$ *indexed by the sorts of* Γ *then* $\mathsf{Rel}_{\Gamma,Q}$ *is the prop with:*
- *set of objects* S_Γ^*,
- *morphisms* $c_1...c_n \to c'_1...c'_m$ *functions* $\prod_{i=1}^n Q_{c_i} \to \mathscr{P}(\prod_{j=1}^m Q_{c_j})$,

- *composition is the usual composition of relations, i.e.* $f \circ g := \mu \circ \mathscr{P}(g) \circ f$, *where* μ *is the canonical map from sets of subsets to subsets,*
- $\otimes$ *is given on objects by concatenation,*
- *and on morphisms* $f : \bigotimes_i c_i \to \bigotimes_j c'_j$ *and* $g : \bigotimes_k d_k \to \bigotimes_l d'_l$ *by* $f \otimes g := \nabla \circ (f \times g)$, *where* ∇ *sends pairs of subsets to their cartesian product,*
- *symmetries* $\sigma : c_1c_2 \to c_2c_1$ *are functions* $Q_{c_1} \times Q_{c_2} \to \mathscr{P}(Q_{c_2} \times Q_{c_1}) : (q, q') \mapsto \{(q', q)\}$.

Note that a non-deterministic monoidal semi-automaton amounts to a morphism of monoidal graphs $\Gamma \to \mathscr{U}\mathsf{Rel}_{\Gamma,Q}$. The adjunction $\mathscr{F}_\times \dashv \mathscr{U}$ implies that there is a unique extension to a strict monoidal functor $\mathscr{F}_\times\Gamma \to \mathsf{Rel}_{\Gamma,Q}$, which we call a non-deterministic *symmetric* monoidal semi-automaton. This functor maps a string diagram to a relation. When this relation relates the initial word to the final word, the string diagram is *accepted*:

▶ **Definition 26.** *Let* $\Delta : \mathscr{F}_\times\Gamma \to \mathsf{Rel}_{\Gamma,Q}$ *be a non-deterministic monoidal automaton with initial and final states* $i, f \in S^*_\Gamma$. *Then the symmetric monoidal language accepted by* Δ *is the set of morphisms* $\mathscr{L}(\Delta) := \{\alpha \in \mathscr{F}_\times\Gamma \mid f \in \Delta(\alpha)(i)\}$.

Intuitively, a run of a symmetric monoidal automaton starts with a *word* of states, whose subwords are modified by transitions corresponding to generators. Identity wires do not modify the states, and symmetries permute adjacent states.

▶ **Observation 27.** *There is an evident correspondence between non-deterministic monoidal automata and regular monoidal grammars. The graphical representation of a grammar (such as Figure 2) makes this most clear: it can also be thought of as the "transition graph" of a non-deterministic monoidal automaton.*

▶ Remark 28. We can further abstract our definition of monoidal automaton by noting that $\mathsf{Rel}_{\Gamma,Q}$ is a sub-prop of the Kleisli category of the powerset monad $\mathscr{P}$, and that this monad could be replaced by another commutative monad [27, Corollary 4.3]. For example, replacing $\mathscr{P}$ with the maybe monad, we obtain deterministic monoidal automata.

6 Asynchronous Automata as Symmetric Monoidal Automata

Asynchronous automata were introduced by Zielonka [32] as a true-concurrent operational model of recognizable trace languages, a well-behaved subclass of trace languages analogous to regular languages. In this section we show they are precisely symmetric monoidal automata over monoidal distributed alphabets, which leads to the following theorem:

▶ **Theorem 29.** *Recognizable trace languages are exactly regular symmetric monoidal languages over monoidal distributed alphabets.*

We recall the definition of asynchronous automata, before turning to monoidal automata.

▶ **Definition 30** (Asynchronous automaton [32])**.** *Let* $(\Sigma_1, ..., \Sigma_k)$ *be a distribution of an alphabet* Σ. *For each* $1 \leqslant i \leqslant k$, *let* Q_i *be a non-empty finite set of states, and for each* $\sigma \in \Sigma$ *take a transition relation* $\Delta_\sigma : \prod_{i \in loc(\sigma)} Q_i \to \mathscr{P}(\prod_{i \in loc(\sigma)} Q_i)$. *This defines a global transition relation on the set* $Q := \prod_{i=1}^k Q_i$ *as follows:*

$$(q_1, ..., q_k) \xrightarrow{\sigma} (q'_1, ...q'_k) \iff q_i = q'_i \text{ for } i \notin loc(\sigma) \text{ and } (q'_{i_1}, ..., q'_{i_j}) \in \Delta_\sigma(q_{i_1}, ..., q_{i_j})$$

where $\{i_1, ..., i_j\} \in loc(\sigma)$. *Finally let* $\vec{i} \in Q, F \subseteq Q$ *be initial and final words of states.*

The global transition relation for σ leaves unchanged those states at locations in the complement of $\mathsf{loc}(\sigma)$, and otherwise acts according to the local transition Δ_σ. An asynchronous automaton has a language over Σ given by the extension of the transition relation

to words. Moreover, asynchronous automata have a language of Mazurkiewicz traces over the distribution of Σ: a trace in $\mathcal{T}(\Sigma, I)$ is accepted when all of its serializations are accepted, which happens when one of its serializations is accepted [32, p. 109]. *Recognizable trace languages* are defined algebraically as those whose syntactic congruence is of finite index [32]. Zielonka's theorem says that they also have an operational characterization:

▶ **Theorem 31** (Zielonka [32]). *Asynchronous automata accept precisely the* recognizable trace languages*.*

Definition 30 closely resembles that of symmetric monoidal automata. Indeed, asynchronous automata are precisely symmetric monoidal automata over monoidal distributed alphabets:

▶ **Proposition 32.** *For an asynchronous automaton $\mathcal{A}$, there is a symmetric monoidal automaton over a monoidal distributed alphabet with the same trace language, and vice-versa.*

Proof. An asynchronous automaton with multiple final state words can be normalized to a single final state word in the usual way by introducing a new final state word and modifying transitions appropriately. Then a symmetric monoidal automaton can be constructed by taking the monoidal distributed alphabet associated to the distribution of Σ (Proposition 18), the same transition relations, initial and final state words. We show that the languages coincide. Let $w \in \mathscr{L}(\mathcal{A})$, and consider the corresponding trace $[w]$. Using Lemma 21, we can produce the corresponding monoidal trace. By construction, this is accepted by the symmetric monoidal automaton defined above. The converse is analogous. ◀

As a corollary, we can invoke Theorem 31 to obtain Theorem 29. In contrast to asynchronous automata, the constructed symmetric monoidal automaton directly accepts traces qua string diagrams, rather than a language of words corresponding to a trace language.

▶ **Observation 33.** *Jesi, Pighizzini, and Sabadini [15] introduced probabilistic asynchronous automata. Initial and final states, and transition relations are replaced by initial and final distributions, and stochastic transitions. These are precisely what are obtained if the powerset monad in our definition of non-deterministic monoidal automaton (Remark 28) is replaced with the* distribution monad *[26], whose Kleisli category has morphisms stochastic matrices.*

7 Serialization via Premonoidal Categories

Trace theorists often consider trace languages to be word languages with the property of *trace-closure* with respect to an independence relation [19]: if $u \in L$ and $u \equiv_I v$ then $v \in L$. These languages arise as preimages of trace languages along the quotient map $q_{\Sigma,I} : \Sigma^* \to \mathcal{T}(\Sigma, I)$. For $L \subseteq \mathcal{T}(\Sigma, I)$ a trace language, $q^{-1}_{\Sigma,I}(L) \subseteq \Sigma^*$ is its *flattening* or *serialization*.

In this section we show that the serialization of monoidal trace languages can be carried out using the algebra and string diagrams of symmetric premonoidal categories. Premonoidal categories are like monoidal categories, except interchange (Figure 1) does not hold in general. The free (symmetric) premonoidal category on a monoidal graph was described using string diagrams by Román [28]. The idea is simple: the string diagrams are the same as for props, but an extra string (the "runtime") threads through each generator, preventing interchange. Figure 6 shows two premonoidal morphisms • ⊗ • → • ⊗ • that are not equal:

In Appendix A, we explain in more detail the construction of the free symmetric premonoidal category $\mathscr{F}_p\Gamma$ on a monoidal graph Γ using string diagrams. In particular, the runtime string appears only once in each string diagram, reflecting that premonoidal categories do not have a tensor product on morphisms. The endomorphism monoid $\mathscr{F}_p\Gamma\left(\begin{smallmatrix}1\\ \vdots\\ n\end{smallmatrix}, \begin{smallmatrix}1\\ \vdots\\ n\end{smallmatrix}\right)$ is now the free monoid over the boxes of Γ, since the runtime prevents interchange:

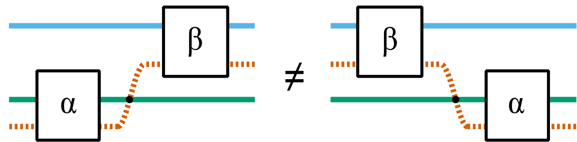


Figure 6 In the free premonoidal category over a monoidal graph, generators are augmented by a string on a new object called the *runtime* (dashed red). This prevents interchange (cf. Figure 1).

▶ **Proposition 34.** *Let* Γ *be a monoidal distributed alphabet. Then* $\mathscr{F}_p\Gamma\begin{pmatrix}1 & 1\\ \vdots, & \vdots\\ n & n\end{pmatrix} \cong B_\Gamma^*$*, where* B_Γ *is the set of boxes of* Γ*.*

Proof (Sketch). By augmenting the generators of Γ with a new runtime, we create a monoidal distributed alphabet in which every generator depends on every other, that is, the independence relation is empty. Thus the corresponding trace monoid is simply B_Γ^*. From here, we can follow the idea of Lemma 21. ◀

We can define a morphism of monoids $\mathfrak{q}_\Gamma : \mathscr{F}_p\Gamma\begin{pmatrix}1 & 1\\ \vdots, & \vdots\\ n & n\end{pmatrix} \to \mathscr{F}_\times\Gamma\begin{pmatrix}1 & 1\\ \vdots, & \vdots\\ n & n\end{pmatrix}$ by presenting $\mathscr{F}_p\Gamma\begin{pmatrix}1 & 1\\ \vdots, & \vdots\\ n & n\end{pmatrix}$ as in Lemma 20, and defining $\mathfrak{q}_\Gamma$ on generators by erasing the runtime string. Theorem 35 then follows immediately from the definitions along with Lemma 21 and Proposition 34:

▶ **Theorem 35.** *For every alphabet* B_Γ*, the following square of monoid homomorphisms commutes, where* q *is the quotient monoid homomorphism.*

$$\begin{array}{ccc} B_\Gamma^* & \xrightarrow{\quad q \quad} & \tau(B_\Gamma, I) \\ {\scriptstyle\cong}\downarrow & & \downarrow{\scriptstyle\cong} \\ \mathscr{F}_p\Gamma\begin{pmatrix}1 & 1\\ \vdots, & \vdots\\ n & n\end{pmatrix} & \xrightarrow{\ \mathfrak{q}_\Gamma\ } & \mathscr{F}_\times\Gamma\begin{pmatrix}1 & 1\\ \vdots, & \vdots\\ n & n\end{pmatrix} \end{array}$$

As a result, the preimage of a monoidal trace language under the morphism $\mathfrak{q}_\Gamma$ corresponds to the serialization of that language.

8 Conclusion

There are several directions in which our theory could be developed. A semi-independence relation drops symmetry from an independence relation: it is simply an irreflexive relation. This gives rise to the theory of semicommutations [7], in which *directed* commutations may occur e.g. $ab \to ba$, but not vice-versa. This allows for a more fine-grained specification of concurrency. In terms of monoidal languages, it suggests consideration of monoidal distributed alphabets in which the sources and targets of generators may differ.

As noted in Remark 23, our treatment of trace languages suggests a generalization of the notion of *effectful category* [28] (which include premonoidal categories), in which there are *multiple runtimes*. This would enable a semantics for concurrent systems in which we can consider not only *atomic* actions, but also actions with input and output types. We plan to pursue this axiomatically in future work.

Mazurkiewicz originally introduced traces to give semantics to Petri nets, and showed that this semantics is compositional with respect to *synchronization* of traces [21]. Petri nets have been given semantics in monoidal categories [2, 22], and so the precise relationship of our monoidal formulation of traces to Petri nets remains to be worked out. In particular, this would involve understanding trace synchronization in terms of monoidal categories.

Finally, proofs of Zielonka's theorem (Theorem 31, see [32] for details) remain highly technical, despite several simplifications since Zielonka's version. Investigation of whether the algebra of monoidal categories might yield further simplifications is an intriguing direction.

References

1. John C. Baez, Brandon Coya, and Franciscus Rebro. Props in network theory. *Theory and Applications of Categories*, 33(25):727–783, 2018.
2. John C Baez, Fabrizio Genovese, Jade Master, and Michael Shulman. Categories of nets. In *2021 36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–13. IEEE, 2021.
3. Guillaume Boisseau and Pawel Sobocinski. String diagrammatic electrical circuit theory. *Electronic Proceedings in Theoretical Computer Science*, 372:178–191, 2022.
4. Filippo Bonchi, Pawel Sobocinski, and Fabio Zanasi. Full abstraction for signal flow graphs. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 515–526, New York, NY, USA, 2015. Association for Computing Machinery. doi:10.1145/2676726.2676993.
5. Francis Bossut, Max Dauchet, and Bruno Warin. A Kleene theorem for a class of planar acyclic graphs. *Inf. Comput.*, 117:251–265, March 1995. doi:10.1006/inco.1995.1043.
6. Albert Burroni. Higher-dimensional word problems with applications to equational logic. *Theoretical Computer Science*, 115(1):43–62, 1993. doi:10.1016/0304-3975(93)90054-W.
7. M Clerbout, M Latteux, and Y Roos. Semi-commutations. In V Diekert and G Rozenberg, editors, *The Book of Traces*. World Scientific, 1995.
8. Bob Coecke, Tobias Fritz, and Robert W. Spekkens. A mathematical theory of resources. *Information and Computation*, 250:59–86, 2016. Quantum Physics and Logic. doi:10.1016/j.ic.2016.02.008.
9. Bob Coecke and Aleks Kissinger. *Picturing quantum processes : a first course in quantum theory and diagrammatic reasoning*. Cambridge University Press, 2017.
10. V Diekert and G Rozenberg. *The Book of Traces*. World Scientific, 1995. doi:10.1142/2563.
11. Volker Diekert and Anca Muscholl. On distributed monitoring of asynchronous systems. In Luke Ong and Ruy de Queiroz, editors, *Logic, Language, Information and Computation*, pages 70–84, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.
12. Matthew Earnshaw and Paweł Sobociński. Regular Monoidal Languages. In Stefan Szeider, Robert Ganian, and Alexandra Silva, editors, *47th International Symposium on Mathematical Foundations of Computer Science (MFCS 2022)*, volume 241 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 44:1–44:14, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.MFCS.2022.44.
13. Hoogeboom H J and G Rozenberg. Dependence graphs. In V Diekert and G Rozenberg, editors, *The Book of Traces*. World Scientific, 1995.
14. Alan Jeffrey. Premonoidal categories and a graphical view of programs. *Preprint*, 1998.
15. S. Jesi, G. Pighizzini, and N. Sabadini. Probabilistic asynchronous automata. *Mathematical systems theory*, 29(1):5–31, February 1996. doi:10.1007/BF01201811.
16. André Joyal and Ross Street. The geometry of tensor calculus, I. *Advances in Mathematics*, 88(1):55–112, 1991. doi:10.1016/0001-8708(91)90003-P.
17. C. Krattenthaler. The theory of heaps and the Cartier-Foata monoid. In P. Cartier and D. Foata, editors, *Commutation and Rearrangements*. European Mathematical Information Service, 2006.
18. Elena Di Lavore, Giovanni de Felice, and Mario Román. Coinductive streams in monoidal categories, 2022. arXiv:2212.14494.
19. Hendrik Maarand and Tarmo Uustalu. Reordering derivatives of trace closures of regular languages. In *30th International Conference on Concurrency Theory (CONCUR 2019)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2019.

20 Saunders MacLane. Categorical algebra. *Bulletin of the American Mathematical Society*, 71(1):40–106, 1965.
21 Antoni Mazurkiewicz. Basic notions of trace theory. In J. W. de Bakker, W. P. de Roever, and G. Rozenberg, editors, *Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency*, pages 285–363, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg.
22 José Meseguer and Ugo Montanari. Petri nets are monoids. *Information and Computation*, 88(2):105–155, 1990. doi:10.1016/0890-5401(90)90013-8.
23 Madhavan Mukund. Automata on distributed alphabets. In *Modern Applications of Automata Theory*, pages 257–288. World Scientific, 2012. doi:10.1142/9789814271059_0009.
24 Chad Nester. Concurrent Process Histories and Resource Transducers. *Logical Methods in Computer Science*, Volume 19, Issue 1, January 2023. doi:10.46298/lmcs-19(1:7)2023.
25 Dusko Pavlovic. Monoidal computer I: Basic computability by string diagrams. *Information and Computation*, 226:94–116, 2013. Special Issue: Information Security as a Resource. doi:10.1016/j.ic.2013.03.007.
26 Paolo Perrone. Distribution monad (nlab entry), 2019. , Last accessed 2023-03-13. URL: https://ncatlab.org/nlab/show/distribution+monad.
27 John Power and Edmund Robinson. Premonoidal categories and notions of computation. *Mathematical Structures in Computer Science*, 7(5), 1997. doi:10.1017/S0960129597002375.
28 Mario Román. Promonads and string diagrams for effectful categories. In *ACT '22: Applied Category Theory, Glasgow, United Kingdom, 18 - 22 July, 2022*, 2022. doi:10.48550/arXiv.2205.07664.
29 P. Selinger. A survey of graphical languages for monoidal categories. In B. Coecke, editor, *New Structures for Physics*, pages 289–355. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. doi:10.1007/978-3-642-12821-9_4.
30 Gérard Xavier Viennot. Heaps of pieces, I : Basic definitions and combinatorial lemmas. In Gilbert Labelle and Pierre Leroux, editors, *Combinatoire énumérative*, pages 321–350, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.
31 R.F.C. Walters. A note on context-free languages. *Journal of Pure and Applied Algebra*, 62(2):199–203, 1989. doi:10.1016/0022-4049(89)90151-5.
32 Wieslaw Zielonka. Notes on finite asynchronous automata. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, 21(2):99–135, 1987.

A Symmetric Strict Premonoidal Categories and Functors

We recall the definitions of (symmetric) strict premonoidal categories and their functors. For more details, see the papers [27, 28].

▶ **Definition 36.** *A strict premonoidal category is a category $\mathcal{C}$ equipped with:*
- *for each pair of objects $A, B \in \mathcal{C}$ an object $A \otimes B$,*
- *for each object $A \in \mathcal{C}$ a functor $A \lhd -$ whose action on objects sends B to $A \otimes B$,*
- *for each object $A \in \mathcal{C}$ a functor $- \rhd A$ whose action on objects sends B to $B \otimes A$, and*
- *a unit object I,*

such that,
- *for each $A \in \mathcal{C}$, strict unitality $I \otimes A = A = A \otimes I$ holds, and*
- *for each triple $A, B, C \in \mathcal{C}$, strict associativity $A \otimes (B \otimes C) = (A \otimes B) \otimes C$ holds.*

The families of functors $A \lhd -, - \rhd A$ are called the *whiskerings* with A: in a premonoidal category we do not have a tensor product of morphisms in general, but we can put an identity on either side of a morphism. A morphism $f : A \to B \in \mathcal{C}$ is *central* if for every morphism $g : C \to D$, $(B \lhd g) \circ (f \rhd C) = (f \rhd C) \circ (A \lhd g)$, in other words, f is central if it interchanges with every other morphism g.

▶ **Definition 37.** *A strict premonoidal category is symmetric if it is further equipped with a natural isomorphism whose components* $c_{A,B} : A \otimes B \to B \otimes A$ *are central and such that* $c_{B,A} \circ c_{A,B} = 1_{A \otimes B}$.

▶ **Definition 38.** *A strict premonoidal functor* $F : \mathcal{C} \to \mathcal{D}$ *is a functor sending central morphisms to central morphisms and such that* $F(I_{\mathcal{C}}) = I_{\mathcal{D}}, F(X \otimes Y) = F(X) \otimes F(Y)$.

A.1 String Diagrams for Premonoidal Categories

We recall the construction of the free symmetric strict premonoidal category over a monoidal graph. This is a special case of the construction of free *effectful categories* in [28, Section 2.3].

We first define the *runtime monoidal graph* over a monoidal graph, which augments the generators with a new wire:

▶ **Definition 39.** *Let* $\mathcal{G}$ *be a monoidal graph. Let* R *be a sort disjoint from* $S_{\mathcal{G}}$*. The* runtime monoidal graph $\mathcal{G}_R$ *has sorts* $S_{\mathcal{G}} + \{R\}$ *and for each generator* $\gamma : S_1 ... S_n \to S'_1 ... S'_m$ *in* $\mathcal{G}$ *a generator* $\gamma : RS_1 ... S_n \to RS'_1 ... S'_m$.

Graphically we can depict $\mathcal{G}_R$ as in Figure 7 (right):

Figure 7 Left: A monoidal graph $\mathcal{G}$. Right: the associated runtime monoidal graph $\mathcal{G}_R$, where the new sort R is drawn as a dashed string.

▶ **Definition 40.** *The symmetric* runtime monoidal category *is the free prop* $\mathscr{F}_{\times}\mathcal{G}_R$ *on* $\mathcal{G}_R$.

▶ **Theorem 41.** *The free symmetric strict premonoidal category* $\mathscr{F}_p\mathcal{G}$ *on a monoidal graph* $\mathcal{G}$ *has set of objects* $S_{\mathcal{G}}$ *and a morphism* $S_1 \otimes ... \otimes S_n \to S'_1 \otimes ... \otimes S'_m$ *is a morphism* $R \otimes S_1 \otimes ... \otimes S_n \to R \otimes S'_1 \otimes ... \otimes S'_m$ *in the symmetric runtime monoidal category.*

Proof. The proof follows [28, Theorem 2.14], in the case where $\mathscr{V}$ is empty, and taking instead the free symmetric strict monoidal category. ◀

In particular note that we no longer have a tensor product of morphisms in $\mathscr{F}_pG$, since the runtime must appear only once in each domain and codomain, but we do have whiskerings for each object.

Consequently the string diagrams for morphisms $A \to B$ in $\mathscr{F}_p\mathcal{G}$ are just morphisms $R \otimes A \to R \otimes B$ in the symmetric runtime monoidal category [28, Corollary 2.15].