12-30-2015

Concurrent Kleene Algebra with Tests and Branching Automata

Peter Jipsen
Chapman University, jipsen@chapman.edu

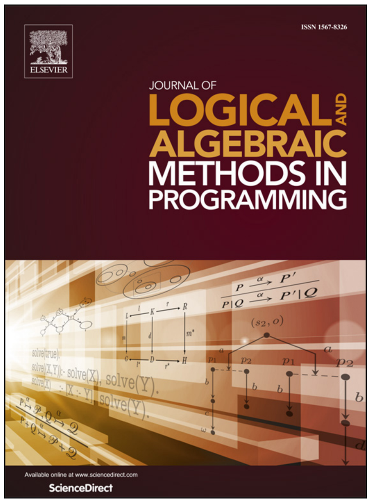
M. Andrew Moshier
Chapman University, moshier@chapman.edu

Recommended Citation

P. Jipsen, M. Andrew Moshier, Concurrent Kleene algebra with tests and branching automata, J. Log. Algebraic Methods Program. (2015), http://dx.doi.org/10.1016/j.jlamp.2015.12.005

Concurrent Kleene algebra with tests and branching automata

Peter Jipsen, M. Andrew Moshier



PII: S2352-2208(15)00151-0
DOI: http://dx.doi.org/10.1016/j.jlamp.2015.12.005
Reference: JLAMP 101

To appear in: *Journal of Logical and Algebraic Methods in Programming*

Received date: 23 November 2014
Revised date: 26 December 2015
Accepted date: 28 December 2015

Highlights

- Guarded strings are generalized to guarded series-parallel strings giving a concrete language model for concurrent Kleene algebra with tests that satisfies the weak exchange law.
- We define a new simpler type of deterministic and nondeterministic branching automata and prove that the fork-regular languages and the series-rational languages coincide.
- To express basic concurrent algorithms, we define concurrent deterministic flowchart schemes and relate them to concurrent Kleene algebras with tests.
- A purely relational CKAT model is defined based on binary relations over a positive sepration algebra.

Concurrent Kleene Algebra with Tests and Branching Automata

Peter Jipsen, M. Andrew Moshier

Chapman University, Orange, California 92866, USA

Abstract

We introduce concurrent Kleene algebra with tests (CKAT) as a combination of Kleene algebra with tests (KAT) of Kozen and Smith with concurrent Kleene algebras (CKA), introduced by Hoare, Möller, Struth and Wehrman. CKAT provides a relatively simple algebraic model for reasoning about semantics of concurrent programs. We generalize guarded strings to *guarded series-parallel strings*, or gsp-strings, to give a concrete language model for CKAT. Combining nondeterministic guarded automata of Kozen with branching automata of Lodaya and Weil one obtains a model for processing gsp-strings in parallel. To ensure that the model satisfies the weak exchange law $(x||y)(z||w) \leq (xz)||(yw)$ of CKA, we make use of the subsumption order of Gischer on the gsp-strings. We also define *deterministic* branching automata and investigate their relation to (nondeterministic) branching automata.

To express basic concurrent algorithms, we define concurrent deterministic flowchart schemas and relate them to branching automata and to concurrent Kleene algebras with tests.

Keywords: Concurrent Kleene algebra, Kleene algebra with tests, parallel programming models, deterministic fork-join automata, series-parallel strings, weak exchange law, positive separation algebra, flowchart schemas

1. Introduction

Relation algebras and Kleene algebras with tests have been used to model specifications and programs, while automata and coalgebras have been used to model state-based transition systems and object-oriented programs. Since processor speeds are leveling off, multi-core architectures and cluster-computing are becoming the norm. However there is little agreement on how to efficiently develop software for these technologies or how to model them with suitably abstract and simple principles. A main feature of using algebra is compositionality, but modeling and verifying concurrent systems in compositional ways is

Email addresses: jipsen@chapman.edu (Peter Jipsen), moshier@chapman.edu (M. Andrew Moshier)

$$\begin{tikzcd} \text{KA} \arrow[r] \arrow[d] & \text{KAT} \arrow[d] \\ \text{CKA} \arrow[r] & \text{CKAT} \end{tikzcd}$$

Figure 1: Relations between classes of Kleene algebras: $\rightarrow$ adds tests, $\downarrow$ adds concurrency

non-trivial because the communication mechanisms of concurrency break compositionality. The recent development of concurrent Kleene algebra [9, 10, 12] builds on an algebraic computational model that is well understood and has numerous applications. Hence it is useful to explore which aspects of Kleene algebras can be lifted fairly easily to the concurrent setting, and whether the simplicity of regular languages and guarded strings can be preserved along the way.

This paper is concerned with four classes of algebras and their relationships (Fig. 1): Kleene algebra (KA), Kleene algebra with tests (KAT), concurrent Kleene algebra (CKA) and the newly defined concurrent Kleene algebra with tests (CKAT). Each of them is finitely axiomatized by simple (quasi)equations based on the equational axioms of idempotent semirings. However they differ in expressive power with respect to the programming language concepts that they can express. KA is the algebraic model of regular languages which can express nondeterministic choice $+$, sequential composition $\cdot$ and finite unbounded iteration *. It also includes the two constants $0, 1$ representing the programs `abort` and `skip`. KAT adds complementation $^-$ restricted to a Boolean subalgebra of tests, allowing it to express `if-then-else` and `while-do`. CKA adds concurrent composition $||$ to KA which models the shuffle operator on strings, as well as parallel composition of pomsets and concurrent programs, while CKAT combines the features of KAT and CKA, using the signature $+, 0, ||, \cdot, 1, ^*, ^-$.

In each of the four classes of interest, a set of generators Σ represent basic (indivisible) programs or actions, the term algebra over Σ represents abstract programs (or compound actions) written in the syntax of the signature, while the free algebra shows which programs are semantically equivalent, i.e., have the same computational effect. In general it can be difficult to check this equivalence in the free algebra, but for KA and KAT the semantic models of all rational languages and all guarded rational languages give a concrete representation of the free algebra. Using automata that recognize these languages, it is possible to decide if two terms of the term algebra become identified in the free algebra, hence program equivalence is testable.

An automaton can also be viewed as an (abstract) implementation of a program (= term) in the sense that it can execute primitive commands and produce sets of traces that represent specific runs of the nondeterministic program. For KA the traces are sequences of generators, so the set of all traces is the set Σ^* of all strings with symbols from Σ. For KAT there are two types of generators: basic programs in Σ and basic tests $T = \{t_1, \dots, t_m\}$. From the basic tests a set $\Gamma = 2^T$ of guards (or atomic tests) is constructed and the relevant notion of a trace is now a *guarded string* starting and ending with a guard, and otherwise alternating basic programs with guards.

For CKA many interesting results have been obtained by Lodaya and Weil [20, 21] using traces that are partially ordered multisets (or pomsets) of Pratt [23] and Gischer [7], but restricted to the class of series-parallel pomsets called sp-posets (detailed definitions are given later). This is related to the set-based traces and dependency relation used in [9, 10, 12] to motivate the laws of CKA. However the model of Lodaya and Weil does not satisfy the weak exchange law $(x||y)(z||w) \leq (xz)||(yw)$ of CKA (see Section 3 for an example). Gischer introduced a subsumption order on pomsets (recalled below) and showed that using sets of pomsets that are downward closed under this order produces a model that does satisfy the weak exchange law. Another approach is used by Hoare et. al. in [11] for their Resource Model, which is a predicate transformer model that satisfies the weak exchange law. We follow Gischer's approach, with the consequence that in this model $x||y$ means that the programs x, y are allowed to be run in parallel, but can also be run sequentially in either order.

Our aim is to investigate how guarded strings can be extended to handle concurrent composition with a similar approach as for sp-posets in [21]. The main contribution is to define a model of CKAT that satisfies the weak exchange law. In this setting the set Γ of guards is given the structure of a positive separation algebra [5] where the separating conjunction determines when two guarded series-parallel strings can be composed concurrently.

We also define a simpler notion of deterministic branching automaton. In the guarded case we extend the nondeterministic automata of Lodaya and Weil to accept guarded series parallel strings. Further we define a trace model for CKAT and give some examples of flowchart schemes to indicate how abstract programs of CKAT relate to some simple concurrent while-programs with assignments. Finally we show how the structure of a separation algebra on Γ can be used to introduce a concurrent composition on binary relations, hence giving a relational model for concurrent programs that identifies two such programs if they have the same input-output relation. The predicate transformer Resource Model of [11] mentioned above also uses a separation algebra to define the concurrent composition of two predicate transformers, but the exact relationship between our relational model and the Resource Model is a topic of future research.

2. Kleene algebra and deterministic automata

Recall that an idempotent semiring is of the form $(A, +, 0, \cdot, 1)$ such that $(A, \cdot, 1)$ is a monoid (i.e., $\cdot$ is associative $(xy)z = x(yz)$ and $x1 = x = 1x$), $(A, +, 0)$ is a (join-)semilattice with bottom (i.e., $+$ is associative, commutative $x + y = y + x$, idempotent $x + x = x$, and $x + 0 = x$) and $x(y + z) = xy + xz$, $(x + y)z = xz + yz$, $x0 = 0 = 0x$.

A *Kleene algebra* $(A, +, 0, \cdot, 1, ^*)$ is an idempotent semiring $(A, +, 0, \cdot, 1)$ with a unary operation * that satisfies the (quasi)identities $x^* = 1 + x + x^*x^*$, $xy \leq y \implies x^*y \leq y$ and $yx \leq y \implies yx^* \leq y$.

An important example of a Kleene algebra is $(\mathcal{P}(\Sigma^*), \cup, \emptyset, \cdot, \Sigma, ^*)$ where Σ is a (usually finite) set of letters, and for subsets X, Y of Σ^*, $X \cdot Y = \{vw :$

$v \in X, w \in Y\}$, $X^0 = 1 = \Sigma$, $X^{n+1} = X \cdot X^n$ and $X^* = \bigcup_{n<\omega} X^n$. A homomorphism R is defined from the term algebra $T_{\mathrm{KA}}(\Sigma)$ to $\mathcal{P}(\Sigma^*)$ by *evaluation*, i.e.,

- $R(p) = \{p\}$ for $p \in \Sigma$, $R(0) = \emptyset$, $R(1) = \Sigma$
- $R(r + s) = R(r) \cup R(s)$, $R(r \cdot s) = R(r) \cdot R(s)$ and $R(s^*) = R(s)^*$.

A subset L of Σ^* is a *rational language* if $L = R(s)$ for some Kleene algebra term s. The subalgebra $R_\Sigma = \{R(s) : s \in T_{\mathrm{KA}}(\Sigma)\}$ is the algebra of rational languages, and the completeness theorem for Kleene algebra, due to Kozen [15], states that R_Σ is isomorphic to the free Kleene algebra $F_{\mathrm{KA}}(\Sigma)$.

We now recall the definition of a deterministic automaton as given for example in [13]. A *deterministic automaton* $\mathcal{X}$ over a set Σ of letters is of the form $(X, \Sigma, \delta, x_0, F)$ where x_0 is the initial state, $\delta : X \times \Sigma \to X$ is the transition function, and F is the set of final states. As usual, the function δ is extended inductively to $X \times \Sigma^*$ by $\delta(x, \varepsilon) = x$ and for all $x \in X, w \in \Sigma^*$ and $a \in \Sigma$, $\delta(x, wa) = \delta(\delta(x, w), a)$. The language accepted by $\mathcal{X}$ is $L(\mathcal{X}) = \{w \in \Sigma^* : \delta(x_0, w) \in F\}$.

An automaton is *finite* if the set X is finite, and a subset of Σ^* is a *regular language* if it is of the form $L(\mathcal{X})$ for some finite automaton $\mathcal{X}$. Kleene's theorem states that a subset of Σ^* is a rational language if and only if it is a regular language.

3. Kleene algebras with concurrency

A *concurrent Kleene algebra* (CKA) [12] is of the form $(A, +, 0, ||, \cdot, 1, ^*)$ such that $(A, +, 0, \cdot, 1, ^*)$ is a Kleene algebra, $(A, +, 0, ||, 1)$ is a commutative idempotent semiring and the weak exchange law holds: for all $x, y, z, w \in A$

$$(x||y)(z||w) \leq xz||yw.$$

Note that the same element 1 is an identity for $\cdot$ and $||$, so the weak exchange law implies $(x||y)w \leq x||yw$, $x(z||w) \leq xz||w$ and $yz \leq y||z$. The definition of CKA sometimes also includes a unary operation $^{\circledast}$ for finite iterations of $||$, but this operation is not used in the present paper. A *generalized CKA* is one in which the weak exchange law is not required to hold.

To describe models of CKA we need the closely related concepts of pomsets and series-parallel strings, which we recall here. More details about these concepts can be found in [3]. Let $||$ be a commutative associative binary operation symbol, and for any set S define $S^{||}$ to be the free commutative monoid generated by S. Note that the elements of $S^{||}$ can be identified with finite multisets over S so, for example, $p||p||q$ is written $\{\!|p, p, q|\!\}$. The set Σ^{sp} of *series-parallel strings* (or sp-strings) over Σ is defined as the smallest set S that satisfies $\Sigma \cup S^* \cup S^{||} \subseteq S$. The operation $||$ is considered a *parallel composition*, and does not interact with sequential composition, except that the empty string ε and the empty multiset $\{\!|\,|\!\}$ are identified. Note that the set Σ^{sp} can be defined directly as the Σ-generated free bimonoid $(\Sigma^{sp}, \cdot, ||, 1)$ subject to the additional identity $x||y = y||x$.

We have defined sp-strings as terms (modulo the equations of free bi-monoids), but they can also be represented uniquely as N-free pomsets (see [7, 8]). Since we make use of these normal forms for sp-strings, we recall the definition here. A Σ-labeled poset is of the form $(P, \leq, \ell)$ where $(P, \leq)$ is a partially ordered set and $\ell : P \to \Sigma$ is a labeling function. Another labeled poset $(P', \leq', \ell')$ is isomorphic to $(P, \leq, \ell)$ if there is a bijection $f : P' \to P$ such that $x \leq' y \Leftrightarrow f(x) \leq f(y)$, and $\ell'(x) = \ell(f(x))$ for all $x, y \in P'$. A *pomset* is an isomorphism class of Σ-labeled posets, and it is N-free if the underlying poset does not have an induced subposet shaped like an N, i.e., with 4 elements a, b, c, d such that $a, b < c$ and $b < d$ but $a \not\leq d$. The sequential composition of two pomsets $(P, \leq, \ell), (P', \leq', \ell')$ is defined by their ordinal sum, i.e., $(P \cup P', \leq \cup \leq' \cup (P \times P'), \ell \cup \ell')$, where we assume that P, P' are disjoint. Parallel composition is simply disjoint union (in each component).

Gischer [7] defined a partial order on pomsets, called the *subsumption order* $\sqsubseteq$, as follows: $(P, \leq, \ell) \sqsubseteq (P', \leq', \ell')$ if there exists a bijection $f : P' \to P$ such that $x \leq' y \Rightarrow f(x) \leq f(y)$, and $\ell'(x) = \ell(f(x))$ for all $x, y \in P'$ (see Figure 2 for a fragment of the subsumption order). The intuition is that pomsets are higher in the subsumption order if they have fewer sequential dependencies, hence downward-closed sets of pomsets contain all (partial or full) sequentializations of the maximal elements. This ensures that the weak exchange law $(x||y)(z||w) \sqsubseteq yz||xw$ holds on pomsets and on downwards-closed sets of pomsets.

Since N-free pomsets are in one-to-one correspondence with sp-strings (modulo the equations of free bi-monoids), the subsumption order induces an order on $(\Sigma^{sp}, \cdot, ||, \sqsubseteq)$, making this structure into an *ordered* bimonoid, i.e., a bimonoid in which both operations are order-preserving. Gischer [7] proved that the set of N-free pomsets is the free algebra generated by Σ in the variety of partially ordered bi-monoids that satisfy the weak exchange law $(x||y)(z||w) \sqsubseteq yz||xw$. A shorter proof of this result is given by Bloom and Esik in [3].

The set of all subsets of Σ^{sp} is not a model of CKA under subset inclusion with $\cdot, ||$ lifted to sets, since the singletons $\{(p||q)(r||s)\}$ and $\{qr||ps\}$ are incomparable for $p, q, r, s \in \Sigma$, hence the weak exchange law fails. However, if one restricts to all downward-closed subsets

$$\mathcal{D}n(\Sigma^{sp}) = \{X \subseteq \Sigma^{sp} : y \sqsubseteq x \in X \implies y \in X \text{ for all } x, y\}$$

then one obtains a model of CKA by defining

$$\begin{aligned} X||Y &= \{z \in \Sigma^{sp} : z \sqsubseteq x||y \text{ for some } x \in X, y \in Y\} \\ X \cdot Y &= \{xy : x \in X, y \in Y\} \\ X + Y &= X \cup Y. \end{aligned}$$

Now, e.g., $\{qr\}||\{ps\} = \{(p||q)(r||s), qr||ps, pqrs, pqsr, psqr, qprs, qpsr, qrps\} \supseteq \{(p||q)(r||s)\} = \{p||q\}\{r||s\}$. The downward closure indicates that parallel composition is interpreted in a permissive way, meaning that if $p||q$, then p, q could

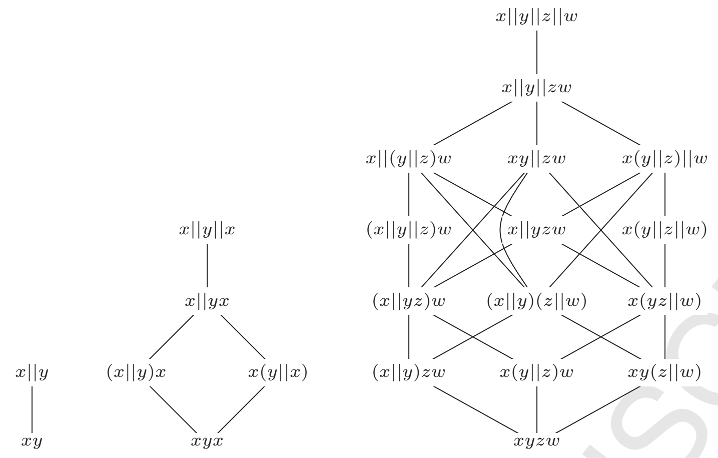


Figure 2: The Gischer subsumption order on some gsp-strings with ≤ 4 elements. The curved line is the weak exchange law, and the other order relations follow from it.

be run in parallel or sequentially (in either order). Gischer [7] proved that the idempotent semiring axioms and the weak exchange law hold for $\mathcal{D}n(\Sigma^{sp})$ ordered by $\subseteq$. Defining $X^* = \bigcup_{n\in\omega} X^n$, it is straight forward to verify that all the CKA axioms are satisfied. We summarize these observations in the following result.

Theorem 1. *The algebra* $(\mathcal{D}n(\Sigma^{sp}), \cup, \emptyset, ||, \cdot, \{1\}, ^*)$ *is a concurrent Kleene algebra.*

4. Deterministic and nondeterministic branching automata

We now modify the automata of Section 2 to include concurrency. A *deterministic branching automaton* over Σ is of the form

$$\mathcal{X} = (X, \Sigma, \delta, \delta_\mathrm{f}, x_0, F)$$

where $(X, \Sigma, \delta, x_0, F)$ is an automaton and $\delta_\mathrm{f} : X \times \mathbb{N}^X \to X$ is the branching transition function. Here $\mathbb{N}^X$ is considered as the collection of multisets over X. For a state x and a multiset of states $\{\!|y_1, \ldots, y_n|\!\}$ the branching transition function produces a state $y = \delta_\mathrm{f}(x, \{\!|y_1, \ldots, y_n|\!\})$. This corresponds to processing an sp-string $u_1|| \ldots ||u_n$ by *forking* into n processes in state x that process the u_i in parallel starting from x, and if the ith process reaches state y_i for $i = 1, \ldots, n$ then all these states are *joined* into the state y. The states x, y are called a *fork-join pair.*

As for ordinary automata, the transition function is extended by induction to $X \times \Sigma^{sp}$ as follows for $x \in X$, $a \in \Sigma$ and $w, u_i \in \Sigma^{sp}$:

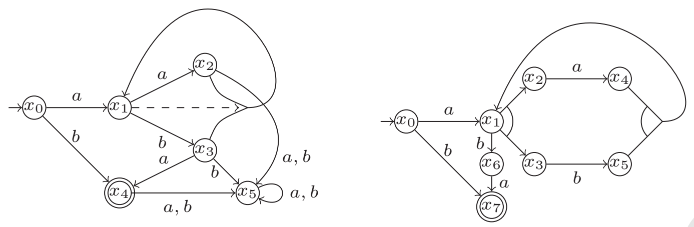


Figure 3: A deterministic branching automaton that accepts $b + a(a||b)^*ba$, and the corresponding nondeterministic automaton of Lodaya and Weil.

$$\delta(x, \varepsilon) = x,\ \delta(x, wa) = \delta(\delta(x, w), a) \text{ and}$$
$$\delta(x, w(u_1|| \dots ||u_n)) = \delta_{\mathrm{f}}(z, \{\!|\delta(z, u_1), \dots, \delta(z, u_n)|\!\}) \text{ where } z = \delta(x, w).$$

An sp-string w is accepted by a branching automaton if $\delta(x_0, w) \in F$. However the branching automata defined above have implicit looping forks, so it is possible that a fork-join appears nested within itself during the trace that accepts w (e.g. this happens if the join loops back into a thread of the fork-join, see [21, Sec. 4.2]). The *fork-acyclic language* $L^{\mathrm{fa}}(\mathcal{X})$ *accepted by* $\mathcal{X}$ is the set of all sp-strings that are accepted by $\mathcal{X}$ without using nested fork-joins. Following the terminology of Lodaya and Weil, a fork-acyclic language that is accepted by a finite automaton is called *fork-regular*.

Figure 3 shows an example comparing a deterministic branching automaton with a branching automaton of Lodaya and Weil. The latter needs more states even though it is nondeterministic. The first automaton has a junk state x_5, and the dotted line points from the fork state x_1 to the corresponding join state. All other (infinitely many) multisets of states have an (implicit) arrow to the junk state (these implicit arrows are not shown on the diagram). Both automata accept the term $b + a(a||b)^*ba$.

A *nondeterministic branching automaton* over Σ is of the form

$$\mathcal{X} = (X, \Sigma, \delta, \delta_{\mathrm{f}}, x_0, F)$$

where X, x_0, F are as before, but $\delta, \delta_{\mathrm{f}}$ now map to $\mathcal{P}(X)$. Hence from a state $x \in X$ there may be many (or no) states that are reached via a given action $a \in \Sigma$ or via a multiset in $\mathbb{N}^X$. Again, the transition function is extended by induction to $X \times \Sigma^{sp}$: $\delta(x, \varepsilon) = \{x\}$, $\delta(x, wa) = \bigcup\{\delta(z, a) : z \in \delta(x, w)\}$ and $\delta(x, w(u_1|| \dots ||u_n)) = \bigcup\{\delta_{\mathrm{f}}(z, \{\!|z_1, \dots, z_n|\!\}) : z \in \delta(x, w), z_i \in \delta(z, u_i), i = 1, \dots, n\}$. An sp-string w is accepted by such an automaton if $F \cap \delta(x_0, w) \neq \emptyset$.

The *series-rational terms* are defined as the absolutely free terms over the signature $+, \cdot, ||, ^*, 1$ of concurrent Kleene algebras, with variables ranging over Σ. If the iterated parallel composition $^{\circledast}$ is included (where $a^{\circledast} = a + a||a +$

$a||a||a+\ldots$), then we obtain the set of *series-parallel-rational terms*. The set $\mathcal{P}(\Sigma^{sp})$ of all sp-languages is a generalized concurrent Kleene algebra under the operations $L+M = L\cup M$, $L\cdot M = \{uv : u\in L, v\in M\}$, $L||M = \{u||v : u\in L, v\in M\}$, $L^* = \bigcup\{L^n : n\in\omega\}$ and $1 = \{\varepsilon\}$. The *series-rational languages* are exactly the members of the subalgebra of $\mathcal{P}(\Sigma^{sp})$ generated by the singleton languages $\{a\}$ for $a\in\Sigma$, and the *series-parallel-rational languages* are the ones obtained if we also include iterated parallel composition during the subalgebra generation process.

The *width* $wd(u)$ of an sp-string u is defined inductively by $wd(a) = 1$ for $a\in\Sigma$, $wd(uv) = \max\{wd(u), wd(v)\}$ and $wd(u||v) = wd(u)+wd(v)$. Alternatively, the width of an sp-string is the maximum number of incomparable elements in the underlying poset (see the previous section and [7] for details on pomsets and their underlying posets). An sp-language L is *of bounded width* if there exists an integer n such that $wd(w)\leq n$ for all $w\in L$. It is easy to see that a series-parallel-rational language has bounded width if and only if it is a series-rational language.

Lodaya and Weil [20, 21] proved a version of Kleene's Theorem for sp-languages. In particular, they proved that regular sp-languages of bounded width coincide with fork-regular languages and with series-rational languages, but their notion of branching automaton is different from the one used in this section. We give more details about the difference in Section 6 where we define guarded branching automata in the style of Lodaya and Weil.

We now show that Kleene's Theorem also holds for the branching automata defined in the current section, using an argument similar to the standard one for finite automata. The fork-join automata of Lodaya and Weil do not lend themselves to this type of constructive proof of Kleene's theorem since sequential composition can result in so-called misbehaved automata [21, Example 4.7]. This is excluded for the type of automata used here since a join transition is always matched to a specific state at which the process forked.

Although Kleene's Theorem was already proved by Lodaya and Weil [20], the proof below is shorter and the automata are new. In particular, the result shows that the branching automata introduced here are equivalent to Lodaya and Weil's branching automata.

Theorem 2. *Let L be a subset of Σ^{sp} for a finite set Σ. Then L is a fork-regular language if and only if L is a series-rational language.*

Proof. Suppose $\mathcal{X}$ is a finite nondeterministic branching automaton over a finite set Σ. We can assume that $\mathcal{X}$ contains only binary fork-joins, since k-ary fork-joins can be constructed from binary ones. The states of $\mathcal{X}$ will be $X = \{1,\ldots,n\}$, and the start state is 1. We need to construct a series-rational term t such that $L(\mathcal{X})$ is obtained by evaluating t in $\mathcal{P}(\Sigma^{sp})$. Let R^k_{ij} be the set of all sp-strings w such that starting in state i and using only intermediate states $1,\ldots,k$ the automaton $\mathcal{X}$ processes w and reaches state j along a fork-acyclic trace. By giving an inductive definition of R^k_{ij} it follows that each of these languages is series rational:

$$R^0_{ij} = \{a : j \in \delta(i,a)\} \cup \{\varepsilon : i = j\}$$

$$R^k_{ij} = R^{k-1}_{ij} \cup R^{k-1}_{ik}(R^{k-1}_{kk})^* R^{k-1}_{kj} \cup R^{k-1} \text{ where}$$

$$R^{k-1} = \bigcup \{R^{k-1}_{ix}((R^{k-1}_{xy_1} || R^{k-1}_{xy_2})R^{k-1}_{yx})^*(R^{k-1}_{xy_1} || R^{k-1}_{xy_2})R^{k-1}_{yj} : \\ y \in \delta_{\mathrm{f}}(x, \{\!| y_1, y_2 |\!\}) \text{ and } k \in \{x, y_1, y_2, y\} \subseteq \{1, 2, \ldots, k\}\}.$$

Since Σ is finite, each R^0_{ij} is a finite set of actions, and hence generated by a term r^0_{ij} that is the sum of these actions. Since the automaton is finite, R^{k-1} is a finite union, hence R^k_{ij} is generated by a term r^k_{ij} that is a finite sum of terms constructed from r^{k-1}_{ij} using CKA operations. Finally, the term t is the finite sum of terms r^n_{1j} where j ranges over all final states of $\mathcal{X}$.

Conversely, suppose L is a series-rational language, and let t be the $\circledast$-free concurrent Kleene algebra term that defines the language L. The automaton $\mathcal{X}$ is constructed in the usual way by combining automata for subexpressions of t. Here we outline the steps for sequential and parallel composition and refer to the traditional proof for the remaining details (see e.g. [13]).

Suppose $\mathcal{Y} = (Y, \Sigma, \delta, \delta_{\mathrm{f}}, y_0, F)$ and $\mathcal{Y}' = (Y', \Sigma, \delta', \delta'_{\mathrm{f}}, y'_0, F')$ are nondeterministic automata for subterms s, s' of t. An automaton $\mathcal{Z} = (Z, \Sigma, \delta'', \delta''_{\mathrm{f}}, z_0, F'')$ for ss' is obtained by taking Z to be the disjoint union of Y, Y' and defining $z_0 = y_0$ and $F'' = F'$. At every final state of $\mathcal{Y}$, we duplicate the outgoing actions of y'_0 as well as all the fork-join multiset transitions. Specifically, for all $y \in F$, let $\delta''(y,a) = \delta(y,a) \cup \delta'(y'_0, a)$ and for a multiset $M = \{\!| y_1, \ldots, y_n |\!\}$,

$$\delta''_{\mathrm{f}}(y, M) = \begin{cases} \delta_{\mathrm{f}}(y, M) & \text{if } M \subseteq Y \\ \delta'_{\mathrm{f}}(y'_0, M) & \text{if } M \subseteq Y' \\ \emptyset & \text{otherwise.} \end{cases}$$

Given these definitions, one then checks that $L^{\mathrm{fa}}(\mathcal{Z}) = L^{\mathrm{fa}}(\mathcal{Y}) \cdot L^{\mathrm{fa}}(\mathcal{Y}')$.

To construct a nondeterministic automaton for $L^{\mathrm{fa}}(\mathcal{Y}) || L^{\mathrm{fa}}(\mathcal{Y}')$ we take Z to be the disjoint union of Y and Y' together with one new state z_1 and let $F'' = \{z_1\}$. We identify y_0 and y'_0 and define this identified state to be the initial state z_0. Finally, for each pair of states $y \in F, y' \in F'$, we let $\delta''_{\mathrm{f}}(z_0, \{\!| y, y' |\!\}) = \{z_1\}$. As in the sequential case, all other transitions on multisets with states from both Y and Y' are defined to be the empty set. The resulting automaton then satisfies $L^{\mathrm{fa}}(\mathcal{Z}) = L^{\mathrm{fa}}(\mathcal{Y}) || L^{\mathrm{fa}}(\mathcal{Y}')$. □

One can also show that every series-rational language is accepted by a deterministic branching automaton, but the argument for determinising nondeterministic automata is more technical than for standard (nonbranching) automata.

We note that our branching automata can be recast in coalgebraic form, as has been done for standard automata in [4]. This is useful if one wants to reason about the category of branching automata, or apply the results more generally to structured state spaces (rather than simply sets of states). However we do not need this generality for our current results, so we do not pursue this further here.

5. Concurrent Kleene algebras with tests

A brief review of Kleene algebras with tests

We now consider the addition of Boolean tests to this setup. Recall from [18] that a *Kleene algebra with tests* (KAT) is an idempotent semiring with a Boolean subalgebra of tests and a unary Kleene-star operation that plays the role of finite (unbounded) iteration. More precisely, it is a two-sorted algebra of the form $\mathbf{A} = (A, A', +, 0, \cdot, 1, ^-, ^*)$ where A' is a subset of A, $(A, +, 0, \cdot, 1, ^*)$ is a Kleene algebra and $(A', +, 0, \cdot, 1, ^-)$ is a Boolean algebra (the complementation operation $^-$ is only defined on A').

Let Σ be a set of *basic program symbols* $p, q, r, p_1, p_2, \ldots$ and T a set of *basic test symbols* $t, t_1, t_2, \ldots$, where we assume that $\Sigma \cap T = \emptyset$. Elements of T are Boolean generators, and we write 2^T for the set of *guards* (or *atomic tests*), given by characteristic functions on T and denoted by $\alpha, \beta, \gamma, \alpha_1, \alpha_2, \ldots$

The collection of guarded strings over $\Sigma \cup T$ is $GS_{\Sigma,T} = 2^T \times \bigcup_{n<\omega} (\Sigma \times 2^T)^n$, and a typical guarded string is denoted by $\alpha_0 p_1 \alpha_1 p_2 \alpha_2 \ldots p_n \alpha_n$, or by $\alpha_0 w \alpha_n$ for short, where $\alpha_i \in 2^T$, $p_i \in \Sigma$ and $w = p_1 \alpha_1 p_2 \alpha_2 \ldots p_n$. Note that for finite T the members of $2^T \subseteq GS_{\Sigma,T}$ can be identified with the atoms of the free Boolean algebra generated by T.

Concatenation of guarded strings is via the *guarded sequential product*:

$$\alpha v \beta \diamond \gamma w \delta = \begin{cases} \alpha v \beta w \delta & \text{if } \beta = \gamma \\ \text{undefined} & \text{otherwise.} \end{cases}$$

For subsets L, M of $GS_{\Sigma,T}$ define

- $L + M = L \cup M$,
- $LM = \{v \diamond w : v \in L, w \in M \text{ and } v \diamond w \text{ is defined}\}$,
- $0 = \emptyset$, $1 = 2^T$, $\overline{B} = GS_{\Sigma,T} \setminus B$ for $B \subseteq 2^T$ and
- $L^* = \bigcup_{n<\omega} L^n$ where $L^0 = 1$ and $L^n = LL^{n-1}$ for $n > 0$.

Then $(\mathcal{P}(GS_{\Sigma,T}), \mathcal{P}(2^T), \cup, \emptyset, \cdot, 2^T, ^*, ^-)$ is a Kleene algebra with tests, called the *full language model*. Note that if $L, M \subseteq 2^T$ are sets of atomic tests then $LM = L \cap M$. Consider the map G from KAT terms over $\Sigma \cup T$ to this concrete model defined by

- $G(t) = \{\alpha \in 2^T : \alpha(t) = 1\}$ for $t \in T$,
- $G(p) = \{\alpha p \beta : \alpha, \beta \in 2^T\}$ for $p \in \Sigma$,
- $G(p+q) = G(p) + G(q)$, $G(pq) = G(p)G(q)$, $G(p^*) = G(p)^*$, for any terms p, q and

- $G(0) = 0$, $G(1) = 1$, $G(\bar{b}) = \overline{G(b)}$ for any Boolean term b.

The *rational language model* $\mathbf{G}_{\Sigma,T}$ is the subalgebra of $\mathcal{P}(GS_{\Sigma,T})$ generated by $\{G(t) : t \in T\} \cup \{G(p) : p \in \Sigma\}$. In fact $\mathbf{G}_{\Sigma,T}$ is the free KAT and its members are the *rational guarded languages*. Subsets of 2^T are called Boolean tests, and other members of $\mathbf{G}_{\Sigma,T}$ are called programs.

A *nondeterministic guarded automaton* is a tuple $\mathcal{X} = (X, \Sigma, T, \delta, x_0, F)$ where $\delta \subseteq X \times (\Sigma \cup \mathcal{P}(2^T)) \times X$ is the transition relation and $F \subseteq X$ is the set of final states. Acceptance of a guarded string w by $\mathcal{X}$ starting from initial state x_0 and ending in state x_f is defined recursively by:

- If $w = \alpha \in 2^T$ then w is accepted iff for some $n \geq 1$ there is a path $x_0 t_1 x_1 t_2 \ldots x_{n-1} t_n x_\mathrm{f}$ in $\mathcal{X}$ of n test transitions $t_i \in \mathcal{P}(2^T)$ such that $\alpha \in t_i$ for $i = 1, \ldots, n$.

- If $w = \alpha p v$ then w is accepted iff there exist states x_1, x_2 such that α is accepted ending in state x_1, there is a transition labeled p from x_1 to x_2 (i.e., $x_2 \in t(x_1)(p)$) and v is accepted by $\mathcal{X}$ starting from initial state x_2.

Finally, w is *accepted by $\mathcal{X}$ starting from x_0* if the ending state x_f is indeed a final state, i.e., satisfies $x_\mathrm{f} \in F$.

Kozen [16] proved that the equational theory of KAT is decidable in PSPACE. Moreover KAT is much more expressive than Kleene algebra since it can faithfully express "if b then p else q" by the term $bp + \bar{b}q$, "while b do p" using $(bp)^*\bar{b}$, and it also interprets Hoare logic.

Adding concurrency to Kleene algebras with tests

After this rather brief discussion of the language semantics and algebraic semantics of KAT, we now describe how these definitions generalize to handle concurrency. Intuitively, elements P, Q of a concurrent Kleene algebra with tests can be thought of as concurrent programs or program fragments, and they are represented by sets of "multi-threaded computations". The operation that needs to be added to KAT is the concurrent composition $P||Q$. Whereas in the sequential model the computation paths are guarded strings, we now need to be able to place two such sequential strings "next to each other", and then we also need to be able to sequentially compose such "concurrent strings" etc. A convenient way to visualize the semantic objects that we would like to construct is to view sequential composition as vertical concatenation (top to bottom) and concurrent composition as horizontal concatenation.

So for example, given two guarded strings $\alpha_0 v \alpha_m$ and $\beta_0 w \beta_n$ we would like to construct

$$\begin{array}{c|c} \alpha_0 & \beta_0 \\ v & w \\ \alpha_m & \beta_n \end{array}$$

As with the guarded sequential product, this concurrent operation is not always defined. The *guards* or *atomic tests* are generalized to *partial functions*

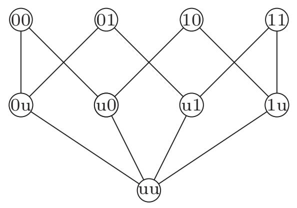


Figure 4: The natural order on Γ when $T = \{s, t\}$

from the set of basic tests T to $2 = \{0, 1\}$. Such a partial function α has domain $\mathrm{dom}(\alpha) \subseteq T$ and is given by its graph $\{(t, \alpha(t)) : t \in \mathrm{dom}(\alpha)\}$. The choice of partial function for guards is motivated by the examples in Section 8 where the state of a flowchart program is determined by a partial function from variables to values, and an atomic test is essentially a complete description of such a state.

We denote the set of all guards by Γ, and define a partial binary operation $\oplus$ on Γ by

$$\alpha \oplus \beta = \begin{cases} \alpha \cup \beta & \text{if } \mathrm{dom}(\alpha) \cap \mathrm{dom}(\beta) = \emptyset \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The operation $\oplus$ comes from separation logic [22, 25], and the partial algebra $(\Gamma, \oplus, \mathbf{u})$ is an example of a *separation algebra* [5], where $\mathbf{u}$ is the empty partial function. Specifically, this means $\oplus$ has $\mathbf{u}$ as unit element ($x \oplus \mathbf{u} = x = \mathbf{u} \oplus x$), and is associative, commutative and cancellative:

$$(x \oplus y) \oplus z = x \oplus (y \oplus z) \qquad x \oplus y = y \oplus x \qquad x \oplus z = y \oplus z \implies x = y.$$

Here an equation is satisfied in Γ if, for all assignments of values to variables, both sides are undefined, or both are defined and equal, while a quasiequation is satisfied if, whenever all premises are defined on both sides and are equal, then the conclusion is defined on both sides and the equality holds.

The operation $\oplus$ defined above is also *positive*: $\alpha \oplus \beta = \mathbf{u}$ implies $\alpha = \mathbf{u}$. Some readers may notice that positive separation algebras are also known in quantum logic as *generalized effect algebras* [6], but we do not elaborate on possible connections in this direction.

Recall the definition of sp-strings and the set Σ^{sp} of all such strings from Section 4. We enlarge this set to $(\Sigma \cup \Gamma)^{sp}$ and define the following operation of *guarded parallel composition*: For $\alpha, \beta, \gamma, \delta \in \Gamma$ and $v, w \in (\Sigma \cup \Gamma)^{sp}$ let

$$\alpha v \beta | \gamma w \delta = \begin{cases} (\alpha \oplus \gamma)(v || w)(\beta \oplus \delta) & \text{if } \alpha \oplus \gamma, \beta \oplus \delta \text{ are defined} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$\alpha | \gamma v \delta = \gamma v \delta | \alpha = \begin{cases} (\alpha \oplus \gamma) v (\alpha \oplus \delta) & \text{if } \alpha \oplus \gamma, \alpha \oplus \delta \text{ are defined} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

$$\alpha | \gamma = \alpha \oplus \gamma$$

Note that in this definition, $v||w$ is the parallel composition of sp-strings, represented by the multiset $\{\!|v, w|\!\}$. The set of *guarded series parallel strings*, or *gsp-strings* for short, is defined as the smallest set $GSP_{\Sigma,T}$ that has Γ and $\Gamma\times\Sigma\times\Gamma$ as subsets and is closed under the guarded sequential product $\diamond$ as well as the guarded concurrent product $|$. For example, if $\Sigma = \{p, q\}$ and $T = \{s, t\}$ then Γ has nine elements $\mathbf{u} = \emptyset$, $\alpha = \{(s, 1)\}$, $\beta = \{(s, 0)\}$, $\gamma = \{(t, 1)\}$, $\delta = \{(t, 0\}$, $\alpha \oplus \gamma$, $\alpha \oplus \delta$, $\beta \oplus \gamma$ and $\beta \oplus \delta$, so for example, the following expressions are gsp-strings:

$$\alpha,\ \mathbf{u},\ \alpha p\alpha,\ \alpha p\beta,\ (\alpha\oplus\gamma)(p||q)(\alpha\oplus\delta),\ (\alpha\oplus\gamma)(p||q)(\alpha\oplus\delta)q\beta,\ \ldots$$

If α, β are guards with overlapping domains then $\alpha|\beta$ is undefined. In particular, if $\alpha \neq \mathbf{u}$ then $\alpha|\alpha$ is undefined. Note that parallel composition of gsp-strings is commutative, which is already reflected in our choice of notation: $v||w = \{\!|v, w|\!\} = \{\!|w, v|\!\} = w||v$. Moreover it is associative, which means that in these "strings", multisets are not members of multisets, i.e., $(u||v)||w$ is normalized to $u||v||w$. This ensures that

$$(\alpha p\beta|\gamma q\delta)|\lambda r\mu = (\alpha\oplus\gamma\oplus\lambda)(p||q||r)(\beta\oplus\delta\oplus\mu) = \alpha p\beta|(\gamma q\delta|\lambda r\mu).$$

Recall that the collection of *all* subsets of guarded strings forms a Kleene algebra with tests. However, for concurrent KAT this would prevent $1 = \Gamma$ from being an identity for concurrent composition, and it would also lead to the failure of the weak exchange law of CKA. To address these two issues, we introduce an ordering on gsp-strings based on Gischer's [7] subsumption order, and consider only subsets of gsp-strings that are upward-closed in this order.

In any commutative partial monoid, the *natural order* is defined by $\alpha \le \beta$ if and only if there exists γ such that $\alpha\oplus\gamma = \beta$. It is easy to see that this relation is reflexive and transitive since a monoid has an identity relation and is associative. Cancellativity and positivity imply that the relation is also antisymmetric, hence it is a partial order. For example, Figure 4 shows the natural order on the set Γ of all partial functions from $T = \{s, t\}$ to $\{0, 1\}$. The bottom element labeled uu is the partial monoid identity $\mathbf{u}$.

We aim to extend this partial order to all gsp-strings by incorporating the Gischer subsumption order. Hence we first extend it from Γ to $\Sigma\cup\Gamma$ by agreeing that for $p \in \Sigma$ and $x \in \Sigma\cup\Gamma$ we have $p \le x$ if and only if $p = x$. So a basic program is incomparable with any other basic program or guard.

As mentioned in the previous section on CKA, a term $v \in (\Sigma\cup\Gamma)^{sp}$ is represented by a pomset $(P_v, \le_v, \ell_v)$ where $\ell_v : P_v \to \Sigma\cup\Gamma$ is a labeling function. We denote by $\hat{P}_v$ the set of all elements in P_v that are labeled by members of Σ, so $\hat{P}_v = \{x \in P_v : \ell_v(x) \in \Sigma\}$. The *guarded dual subsumption order* $\sqsubseteq$ on $v, w \in (\Sigma\cup\Gamma)^{sp}$ is given by $v \sqsubseteq w$ if and only if there exists an order-preserving function $f : P_v \to P_w$ such that f restricts to a bijection from $\hat{P}_v$ to $\hat{P}_w$ and for all $x \in P_v$ we have $\ell_v(x) \le \ell_w(f(x))$.

Our semantic CKAT model will be based on $GSP_{\Sigma,T} \subseteq (\Sigma\cup\Gamma)^{sp}$ so we mostly make use of the restriction of $\sqsubseteq$ to gsp-strings. Recall that in a poset, b is a *cover* of a if a is below b and there is no element strictly between a, b. In this case we also say that a is a *co-cover* of b.

Lemma 3. *If T is finite then $(GSP_{\Sigma,T}, \sqsubseteq)$ has no infinite chains, and it is decidable if $v \sqsubseteq w$ holds for two gsp-strings.*

Proof. Note that if $v \sqsubseteq w$ holds then v, w have the same finite number, say n, of elements labeled by basic programs of Σ and these sets of labels coincide. They may differ on the number of elements from Γ (= partial functions from T to 2), but these elements are covers or co-covers of basic programs, and each basic program has exactly one cover and one co-cover. Hence there are at most $3n$ elements in v and w, so there are at most $(3n)!$ gsp-strings above and below any v. □

For a poset $(P, \leq)$, we use the notation $\mathcal{U}p(P)$ for the set of all upward-closed subsets of P. The language model over gsp-strings is defined as in the case of guarded strings, except that we use only upward-closed subsets, replace 2^T by $(\Gamma, \leq)$ where $\leq$ is the natural order, redefine complementation as pseudocomplementation, and we now have to define the additional operation of concurrent composition. For $B \in \mathcal{U}p(\Gamma)$ and $L, M \in \mathcal{U}p(GSP_{\Sigma,T})$ let

- $1 = \Gamma, \quad \overline{B} = \bigcup\{C \in \mathcal{U}p(\Gamma) : B \cap C = \emptyset\}$, and
- $L||M = \{w : u|v \sqsubseteq w$ for some $u \in L, v \in M$ and $u|v$ is defined$\}$.

This makes $\mathcal{U}p(GSP_{\Sigma,T})$ into a complete concurrent Kleene algebra. Note the use of upward-closed sets rather than downward closed sets as in Theorem 1. The reason is that the natural order on Γ needs to be compatible with the subsumption order, hence the latter is reversed in the definition of guarded dual subsumption.

The map G from the beginning of this section is modified and extended homomorphically to all terms of KAT with $||$ as follows:

- $G(t) = \{\alpha \in \Gamma : \alpha(t) = 1\}$ for $t \in T$,
- $G(p) = \{\alpha p \beta : \alpha, \beta \in \Gamma\}$ for $p \in \Sigma$,
- $G(p+q) = G(p) \cup G(q)$, $G(pq) = G(p)G(q)$, $G(p||q) = G(p)||G(q)$, $G(p^*) = G(p)^*$, for any terms p, q and
- $G(0) = 0$, $G(1) = 1$, $G(\bar{b}) = \overline{G(b)}$ for any term b without occurrences of any basic program $p \in \Sigma$.

The bi-Kleene algebra of *series-rational gsp-languages*, denoted by $\mathbf{C}_{\Sigma,T}$, is the subalgebra generated by $\{G(t) : t \in T\} \cup \{G(p) : p \in \Sigma\}$. The phrase "series-rational" comes from the paper by Lodaya and Weil [21], where they use the name series-rational sp-language for the members of their language model.

With this language model as guide, we now define a *concurrent Kleene algebra with tests* (CKAT) as an algebra $\mathbf{A} = (A, A', +, 0, ||, \cdot, 1, ^*, ^-)$ where

1. $(A, +, 0, \cdot, 1, ^*)$ is a Kleene algebra,
2. $(A, +, 0, ||, 1)$ is a commutative semiring with 0 and identity 1,

3. $(A', +, 0, \cdot, 1)$ is a bounded distributive lattice with $A' \subseteq A$,
4. the pseudocomplement $\bar{b} = \sum\{c \in A' : b \cdot c = 0\}$ exists for all $b \in A'$, and
5. the weak exchange law $(x||y)(z||w) \le xz||yw$ holds for all $x, y, z, w \in A$.

Although the fourth axiom appears to be infinitary, it can be replaced by the (two) quasiequations $c \le \bar{b} \iff b \cdot c = 0$ for all $b, c \in A'$, and these in turn are equivalent to the identities $b\overline{b\bar{c}} = b\bar{c}$, $\bar{0} = 1$ and $\bar{1} = 0$. Further details about pseudocomplemented distributive lattices can be found, e.g., in [2], although this paper uses only basic properties of these algebras.

An alternative would be to use a Heyting algebra for the test elements of a CKAT since that would be a good match with separation logic. However it would change the syntax (replacing $^-$ with the Heyting implication $\to$) and the subalgebra of tests would lose the property of being locally finite (the 1-generated free Heyting algebra is infinite whereas any finitely generated pseudocomplemented distributive lattice is finite).

In the proof below, we use the notation $\uparrow L = \{w : v \sqsubseteq w \text{ for some } v \in L\}$.

Theorem 4. $(\mathcal{U}p(GSP_{\Sigma,T}), \mathcal{U}p(\Gamma), \cup, \emptyset, ||, \cdot, \Gamma, ^*, ^-)$ *is a concurrent Kleene algebra with tests. If $T = \emptyset$ then the $^-$-free reduct of this algebra is isomorphic to the* CKA $\mathcal{D}n(\Sigma^{sp})$ *from Theorem 1.*

Proof. It is easy to see that $\mathcal{U}p(GSP_{\Sigma,T})$ is closed under the operations. In particular, the operation $L||M$ is defined to be the upward closure of $\{u|v : u \in L, v \in M\}$ to ensure that the result is in $\mathcal{U}p(GSP_{\Sigma,T})$, whereas $\cup, \cdot, ^*, ^-$ preserve the property of being upward-closed.

The KA axioms for $\cup, \emptyset, \cdot, \Gamma, ^*$ are also straight forward to check since they hold for all subsets of $GSP_{\Sigma,T}$. Similarly $||$ is associative, commutative, distributes over $\cup$ and has $\emptyset$ as a zero. To see that Γ is an identity for $||$, we note that $L||\{\mathbf{u}\} = \uparrow L \subseteq L$, hence $L||\Gamma \subseteq L$. For the reverse inclusion, $w \in L||\Gamma$ implies $v|\alpha \sqsubseteq w$ for some $v \in L$ and $\alpha \in \Gamma$. Since $v \sqsubseteq v|\alpha$ and L is upward-closed, it follows that $v \in \uparrow L = L$.

To prove that the weak exchange law holds, let $L, M, P, Q \in \mathcal{U}p(GSP_{\Sigma,T})$ and suppose $r \in (L||M)(P||Q)$. Then $r = x \diamond y$ for some $x \in L||M$ and $y \in P||Q$. It follows that $x \sqsupseteq \alpha s \alpha' | \beta t \beta'$, $y \sqsupseteq \gamma u \gamma' | \delta v \delta'$, $\alpha' \oplus \beta' = \gamma \oplus \delta$ and $r \sqsupseteq (\alpha \oplus \beta)(s||t)(\alpha' \oplus \beta')(u||v)(\gamma' \oplus \delta')$, where $\alpha s \alpha' \in L$, $\beta t \beta' \in M$, $\gamma u \gamma' \in P$ and $\delta v \delta' \in Q$.

Let $\kappa = \alpha' \oplus \beta'$. Since L, M, P, Q are upward-closed, $\alpha s \kappa \in L$, $\beta t \kappa \in M$, $\kappa u \gamma' \in P$ and $\kappa v \delta' \in Q$. Using the definition of the guarded subsumption order we see that

$$r \sqsupseteq (\alpha \oplus \beta)(s||t)(\alpha' \oplus \beta')(u||v)(\gamma' \oplus \delta') \sqsupseteq (\alpha \oplus \beta)(s\kappa u||t\kappa v)(\gamma' \oplus \delta')$$

where the order-preserving function f between the gsp-strings sends the two elements labeled κ to the element labelled $\alpha' \oplus \beta'$ and is the unique order-preserving bijection on the other elements. The right hand gsp-string is an element in $LP||MQ$, hence r is also in this upward-closed set.

If $T = \emptyset$ then $\Gamma = \{\mathbf{u}\}$, hence $\mathcal{U}p(\Gamma) = \{\emptyset, \{\mathbf{u}\}\} = \{0, 1\}$. The set $GSP_{\Sigma,\emptyset}$ is isomorphic to Σ^{sp}, where the isomorphism g is given by deleting all the guards in the gsp-string. This map lifts to an isomorphism $\hat{g}$ between the algebras. $\square$

We do not include iterated concurrent composition (i.e., parallel star) in the definition of a CKAT since this operation prevents the generalization of Kleene's theorem to gsp-languages (see [21, Sec. 3] for further discussion). However the notation $||_{i=1}^{n} p_i$ is used to abbreviate the (fixed length) term $p_1||p_2||\cdots||p_n$ that corresponds to a parallel for-loop.

There are several other models of concurrency that are similar to our approach but they differ in the details. In particular the notion of synchronous Kleene algebra (SKA) by Prisacariu [24] is quite close to CKA, and the cited paper also contains an extension to SKA with tests (SKAT). However the models in that paper do not satisfy the weak exchange law of CKA (though they do satisfy the stronger equational version on a subset of idempotent basic actions). For more details about the relationship between SKA and other concurrency models, including pomsets and CKA, see [24, Sec. 4].

6. Automata over guarded series-parallel strings

The notion of nondeterministic automaton for gsp-strings is based on the one for guarded strings, but it is expanded with fork and join transitions taken from the branching automata of Lodaya and Weil [21]. Specifically a *guarded branching automaton* is of the form

$$\mathcal{X} = (X, \delta, \delta_{\text{fork}}, \delta_{\text{join}}, F)$$

where

- (X, δ, F) is a guarded automaton,
- $\delta_{\text{fork}} \subseteq X \times \mathcal{M}(X)$ and
- $\delta_{\text{join}} \subseteq \mathcal{M}(X) \times X$.

Here X is a set of states and $\mathcal{M}(X)$ is the collection of multisets of X with more than one element. As for guarded automata, δ is the transition function and F is the the set of final states. The relations δ_{fork} and δ_{join} give the *fork* and *join* relations respectively.

Fork transitions in δ_{fork} are denoted $(x, \{\!|x_1, x_2, \ldots, x_n|\!\})$, and if the multiset has n elements they are called forks of arity n. The join transitions of arity n are defined similarly, but with the order of the two components reversed.

The acceptance condition for gsp-strings needs to be defined carefully since it substantially extends the one for guarded strings. Intuitively one can think of an automaton as evaluating the acceptance condition for parallel parts of the input string concurrently on separate processors. In many cases, when large scale parallel programs are run on a distributed cluster of computers, (part of) the program code is distributed to all the available processors and executes in separate environments until at an appropriate point results are communicated back to a subset of the processors (perhaps a single one) and combined into a new state. This fork and join paradigm is of course a fairly restricted model of concurrent programming, which has the merit of being quite simple and algebraic since it avoids syntactic annotations for named channels and other more architecture-dependent features. It also meshes well with our generalization of guarded strings and with the laws of concurrent Kleene algebra.

For the actual definition of acceptance we do not need to have separate copies of automata, instead we simply map the parallel parts of a gsp-string into the same automaton. Looking back at the recursive definition of acceptance for a (non-concurrent) guarded string relative to an initial state x_0, it is apparent that this condition is equivalent to finding a path from x_0 to some final state x_f such that the basic program symbols in the string match with symbols along the path in the same order, and if $p_{i-1}\alpha_i p_i$ occurs in the guarded string then there is a path $\beta_1 \dots \beta_{n_i}$ of tests $\beta_k \geq \alpha_i$ along edges of the automaton that lie between the edges matched by p_{i-1} and p_i. For gsp-strings we define a similar "embedding" into the automaton where parallel branches correspond to a fork transition, followed by parallel (not necessarily disjoint) paths along matching edges until they reach a join transition. The precise recursive definition is as follows: A *weak guarded series parallel string* (or wgsp-string for short) is a gsp-string but possibly without the first and/or last guard. Acceptance of a wgsp-string w by $\mathcal{X}$ starting from initial state x_0 and ending at state x_f, is defined recursively by:

- If $w = \alpha \in \Gamma$ then w is accepted if and only if for some $n \geq 1$ there is a sequential path $x_0 t_1 x_1 t_2 \dots x_{n-1} t_n x_\mathrm{f}$ in $\mathcal{X}$ (i.e., (x_{i-1}, t_i, x_i) is an edge in $\mathcal{X}$) of n test transitions $t_i \in \mathcal{P}(\Gamma)$ such that $\alpha \in t_i$ for $i = 1, \dots, n$.

- If $w = p \in \Sigma$ then w is accepted if and only if there exists a transition labelled p from x_0 to x_f.

- If $w = u_1|u_2| \dots |u_m$ for $m > 1$ then w is accepted if and only if there exist a fork $(x_0, \{\!|x_1, \dots, x_m|\!\})$ and a join $(\{\!|y_1, \dots, y_m|\!\}, x_\mathrm{f})$ in $\mathcal{X}$ such that u_i is accepted starting from x_i and ending at y_i for all $i = 1, \dots, m$.

- If $w = uv$ then w is accepted if and only if there exist a state x such that u is accepted ending in state x and v is accepted by $\mathcal{X}$ starting from initial state x and ending at x_f.

Finally, w is *accepted by $\mathcal{X}$ starting from* x_0 if $x_\mathrm{f} \in F$.

In the second recursive clause the fork transition corresponds to the creation of n separate processes that can work in parallel on the acceptance of the wgsp-strings $u_1, \dots, u_n$. The matching join-operation then corresponds to a communication or merging of states that terminates these processes and continues in a single thread.

The sets of gsp-strings that are accepted by a finite automaton are not necessarily upward-closed in the guarded dual subsumption order, hence we define a *regular gsp-language* to be a set of gsp-strings that is the upward closure of a set of all gsp-strings that are accepted by some finite automaton.

For sets of (unguarded) strings, the regular languages and the series-rational languages (i.e., those built from Kleene algebra terms) coincide. However, Lodaya and Weil pointed out that this is not the case for sp-strings, since for example the language $\{p, p||p, p||p||p, \ldots\}$ is regular, but not a series-rational language. As for sp-strings, the *width* of a gsp-string is the maximal cardinality of an antichain in the underlying poset. A (g)sp-language is said to be of *bounded width* if there exists $n < \omega$ such that every member of the language has width less than n. Intuitively this means that the language can be accepted by a machine that has at least n (virtual) processors. The series-rational languages are of bounded width since concurrent iteration was not included as one of the operations of CKAT.

The condition of bounded width can be rephrased as a restriction on the automaton. A run of $\mathcal{X}$ is called *fork-acylic* if a matching fork-join pair never occurs as a matched pair nested within itself. The automaton is fork-acylic if all the accepted runs of $\mathcal{X}$ are fork-acyclic. Lodaya and Weil prove that if a language is accepted by a fork-acyclic automaton then it has bounded width, and their proof applies equally well to gsp-languages.

At this point it is not clear whether the algebra of regular gsp-languages is isomorphic to the free CKAT, or if guarded branching automata give a decision procedure for the equational theory of concurrent Kleene algebras with tests.

7. Trace semantics for concurrent Kleene algebras with tests

Kozen and Tiuryn [19] (see also [16]) show how to provide trace semantics for programs (i.e. terms) of Kleene algebra with tests. This is based on an elegant connection between computation traces in a Kripke frame and guarded strings. Here we point out that this connection extends very simply to the setting of concurrent Kleene algebras with tests, where traces are related to labeled Hasse diagrams of posets and these objects in turn are associated with guarded series-parallel strings.

Adapting the Kripke frames for KAT, we define a *separation frame* over Σ, T to be a structure $(K, \oplus, \mathbf{u}, m_K)$ where K is a set of *states* that is the base set of a positive separation algebra $(K, \oplus, \mathbf{u})$, $m_K : \Sigma \to \mathcal{U}p(K \times K)$ and $m_K : T \to \mathcal{U}p(K)$. Here the upward closure is computed with respect to the natural order of K, defined as before by $s \leq t$ if and only if there exists $r \in K$ such that $s \oplus r = t$. The upward closure in $K \times K$ is calculated in the cartesian product of the poset $(K, \leq)$ with itself.

A gsp-trace τ in K is essentially a gsp-string with the guards replaced by states in K, such that whenever a triple $spt \in K \times \Sigma \times K$ is a subtrace of τ then $(s, t) \in m_K(p)$. The set of all gsp-traces over K is denoted GSP_K. The subsumption order $\sqsubseteq$ on GSP_K is defined exactly as for gsp-strings, and likewise for the coalesced product $\sigma \diamond \tau$ of two gsp-traces σ, τ (if σ ends at the same state as where τ starts) as well as the parallel product $\sigma|\tau$ (where $\oplus$ is used to combine the first two states as well as the last two states, while the remaining parts of the gsp-traces are combined using the parallel composition of their pomsets). These partial operations lift to sets X, Y of gsp-traces by

- $XY = \{\sigma \diamond \tau : \sigma \in X, \tau \in Y$ and $\sigma \diamond \tau$ is defined$\}$
- $X||Y = \{\rho : \sigma|\tau \sqsubseteq \rho$ for some $\sigma \in X, \tau \in Y$ and $\sigma|\tau$ is defined$\}$.

The proof of the following result is very similar to the preceeding proof for $GSP_{\Sigma,T}$.

Theorem 5. *For any separation frame $(K, \oplus, \mathbf{u}, m_K)$ the algebra*

$$(\mathcal{U}p(GSP_K), \mathcal{U}p(K), \cup, \emptyset, ||, \cdot, K, ^*, ^-)$$

is a CKAT*.*

For a subset B of a poset $(K, \sqsubseteq)$, we use the notation $\downarrow B$ for the set $\{x \in K : x \leq y$ for some $y \in B\}$. Programs (= terms of CKAT) are interpreted in K using a modified inductive definition of Kozen and Tiuryn [19] extended by a clause for $||$:

- $[\![p]\!]_K = \{spt : (s,t) \in m_K(p)\}$ for $p \in \Sigma$
- $[\![0]\!]_K = \emptyset$ and $[\![b]\!]_K = m_K(b)$ for $b \in T$
- $[\![\bar{b}]\!]_K = K \setminus \downarrow m_K(b)$ and $[\![p+q]\!]_K = [\![p]\!]_K \cup [\![q]\!]_K$
- $[\![pq]\!]_K = ([\![p]\!]_K)([\![q]\!]_K)$ and $[\![p^*]\!]_K = \bigcup_{n<\omega} [\![p]\!]_K^n$
- $[\![p||q]\!]_K = [\![p]\!]_K || [\![q]\!]_K$.

Each gsp-trace τ has an associated gsp-string gsp(τ) obtained by replacing every state s in τ with the corresponding unique guard $\alpha \in \Gamma$ that satisfies $s \in [\![\alpha]\!]_K$. It follows that gsp(τ) is the unique guarded sp-string over Σ, T such that $\tau \in [\![\text{gsp}(\tau)]\!]_K$. As a result there is a similar connection between gsp-trace semantics and gsp-strings as in [19] (the proof is by induction on the structure of p).

Theorem 6. *For a separation frame $(K, \oplus, \mathbf{u}, m_K)$, program p and gsp-trace τ, we have $\tau \in [\![p]\!]_K$ if and only if $gsp(\tau) \in G(p)$, whence $[\![p]\!]_K = gsp^{-1}(G(p))$.*

The trace model for guarded strings has many applications since each trace in $[\![p]\!]_K$ can be interpreted as a sequential run of the program p starting from the first state of the trace. The gsp-trace model provides a similar interpretation for programs that fork and join threads during their runs. Each gsp-trace in $[\![p]\!]_K$ is a representation of the basic programs and tests that were performed during the possibly concurrent execution of the program p. Note that there are no explicit fork and join transitions in a gsp-trace since, unlike a gsp-automaton (which has to allow for nondeterministic choice), whenever a state in a gsp-trace has several immediate successor states, this is the result of a fork, and similarly states with several immediate predecessors represent a join.

While series-parallel traces are more complex than linear traces, they can, like gsp-strings, still be represented by planar lattice diagrams where parallel composition is denoted by placing traces next to each other (with partial sum of the start states and end states), and sequential composition is given by placing traces vertically above each other (with only one connecting state between them).

The gsp-trace semantics are useful for analysing the behavior of threads that communicate only indirectly with other concurrent threads via joint termination in a single state. While this is a restricted model of concurrency, it has a simple algebraic model based on Kleene algebras with tests, and it satisfies the laws of concurrent Kleene algebra including the weak exchange law.

At the end of the next section we present another model where states are assignments from variables to values. In such a setting the notion of separation algebra is used to guarantee the absence of race conditions in concurrent programs.

8. Concurrent flowchart schemes and Kleene algebras with tests

In order to code specific concurrent algorithms, we first recall flowchart schemes and then extend them with fork and join statements. Flowchart schemes were originally introduced by Ianov [14], and later related to Kleene algebras with tests by Angus and Kozen [1]. They are defined over a standard first-order signature consisting of finitely many function symbols $f, g, \ldots$ and predicate symbols $P, Q, \ldots$, each with a fixed arity. Terms are built from variables $\{x_i, y_i, z_i : i = 1, 2, \ldots\}$ and function symbols, and an expression $t(\mathbf{x}, \mathbf{y})$ indicates that the term t uses (some) variables from the sequences $\mathbf{x} = x_1, \ldots, x_m$ and $\mathbf{y} = y_1, \ldots, y_n$.

Deterministic flowchart schemes are finite directed graphs with nodes labeled by statements. A start statement with one outgoing edge is followed by assignment statements $\mathbf{y} := \mathbf{t}(\mathbf{x}, \mathbf{y})$ with one outgoing edge, test statements $P(\mathbf{t}(\mathbf{y}))$ with two outgoing edges labeled T and F and halt statements with no outgoing edges. A start statement has no incoming edges and all other statements have a finite non-zero number of incoming edges. Here $\mathbf{t} = t_1, \ldots, t_n$ are terms, $\mathbf{x}$ are *input variables*, $\mathbf{y}$ are *work variables*, and P is a predicate symbol. An assignment statement $y_1, \ldots, y_n := t_1, \ldots, t_n$ is between sequences of the same length, where the terms t_i on the right are first all evaluated and then assigned to their corresponding variable on the left. Given extra work variables, such an assignment can be simulated by a sequential list of simple assignments of the form $y := t$.

The general form of a deterministic flowchart scheme is given in Figure 5. The unique *start statement* must be followed directly by an *initialization statement* that assigns values to all work variables using only the input variables. Every halt statement is preceded by a *finalization statement* that assigns values to all *output variables* $\mathbf{z} = z_1, \ldots, z_l$. In Figure 5 the close coupling of the start statement with initialization, and likewise of finalization with the halt statement, is suggested by drawing them close together. These syntactic restrictions ensure that the output values are a function of the input values.

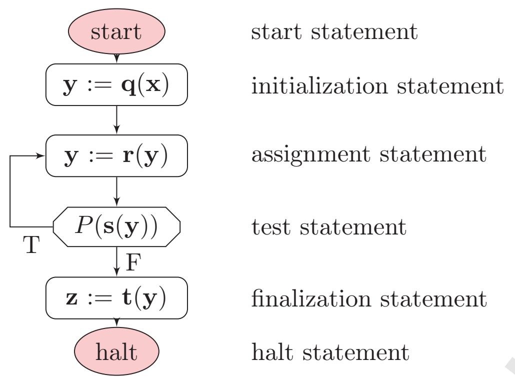

start
$\mathbf{y} := \mathbf{q}(\mathbf{x})$
$\mathbf{y} := \mathbf{r}(\mathbf{y})$
$P(\mathbf{s}(\mathbf{y}))$
T
F
$\mathbf{z} := \mathbf{t}(\mathbf{y})$
halt
start statement
initialization statement
assignment statement
test statement
finalization statement
halt statement

Figure 5: A flowchart scheme showing the different types of statements

Concurrent deterministic flowchart schemes are defined as above, but with two more statement types: *fork* and *join*. Each fork statement has $k > 1$ outgoing edges followed directly by initialization statements $\mathbf{y_i} := \mathbf{r_i}(\mathbf{x}, \mathbf{y})$ for $i = 1, \ldots, k$. Here $\mathbf{y_i} = y_{i1}, \ldots, y_{in_i}$ is a sequence of work variables distinct from all other variables. Operationally, when a fork is processed, the current process is suspended, the initialization statements of the k new processes are evaluated, and then these processes continue concurrently. The work variables of the suspended process can be accessed by the new processes, but this can lead to race conditions where two concurrent processes modify/read the same variable, resulting in potential nondeterminism. A flowchart scheme is called *cautious* if any two paths starting from the same fork reach the same join, and (after initialization) only have access to disjoint sets of variables. This separation condition ensures the absence of race conditions so that the flowchart scheme is deterministic. After the fork statement, each computation path evolves its own state, but at the join statement the k states are merged (by set union) into a common state.

Note that each strand of a fork-join block has its own initialization statement, and that the work variables of a strand are separated from all other variables. This simplifies reasoning about the semantics of such concurrent processes, and also implies that concurrent schemes are reasonable models for distributed computing. All strands of a fork are initialized concurrently, before any of them are processed. After the initialization step, they can be processed in any order, in parallel or sequentially, without affecting the semantics. Thus strands do not necessarily correspond to parallel threads, they merely indicate which parts of the flowchart scheme can be processed in parallel.

As an example of a concurrent scheme, consider the flowchart in Figure 6. For an associative operation $\oplus$, it evaluates the term $x_1 \oplus x_2 \oplus \cdots \oplus x_8$ concurrently, using three iterations of a 4-ary fork-join. Note that this flowchart can easily be translated into a CKAT term of the form

$$p(b(p_1q_1||p_2q_2||p_3q_3||p_4q_4)q)^*\bar{b}r$$

where p is the initial assignment, b is the test $h > 0$, p_iq_i are the assignments between the fork and join, q is the decrement and r is the finalization.

Figure 7 shows how to construct a concurrent flowchart that implements a parallel for-loop. Strictly speaking, this is not a scheme, since the symbols $+, -$ are to be interpreted as integer addition and subtraction. The first strand of the fork also has paths that do not end at the join, though one can observe that all actual computation paths (traces) out of the fork do in fact arrive at the join. With the help of this **forpar** construct, a more general version of the term-evaluation algorithm is given in Figure 8.

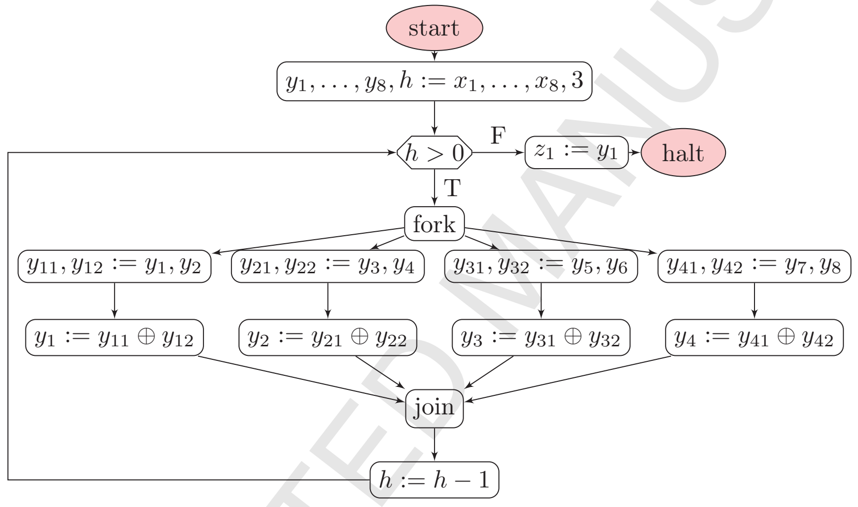

start
$y_1, \ldots, y_8, h := x_1, \ldots, x_8, 3$
$h > 0$
F
$z_1 := y_1$
halt
T
fork
$y_{11}, y_{12} := y_1, y_2$
$y_{21}, y_{22} := y_3, y_4$
$y_{31}, y_{32} := y_5, y_6$
$y_{41}, y_{42} := y_7, y_8$
$y_1 := y_{11} \oplus y_{12}$
$y_2 := y_{21} \oplus y_{22}$
$y_3 := y_{31} \oplus y_{32}$
$y_4 := y_{41} \oplus y_{42}$
join
$h := h - 1$

Figure 6: A concurrent scheme for calculating $z_1 := x_1 \oplus x_2 \oplus \cdots \oplus x_8$

We now construct a concurrent Kleene algebra with tests that gives trace semantics for concurrent flowchart schemes. Since Kleene algebras are closely connected to automata, we first consider how flowcharts can be translated to automata on guarded strings.

Flowcharts have nodes labelled by actions or tests, whereas automata have edges labelled by actions and nodes correspond to states (that corespond to code points in the program represented by the automaton or flowchart). To convert a flowchart into a guarded automaton, it suffices to move the action labels from the nodes to the outgoing edges. For a test statement b that has two outgoing edges, the label b is moved to the *true* edge and $\bar{b}$ is moved to the *false* edge. The (now unlabeled) nodes of the flowchart will then correspond to the states

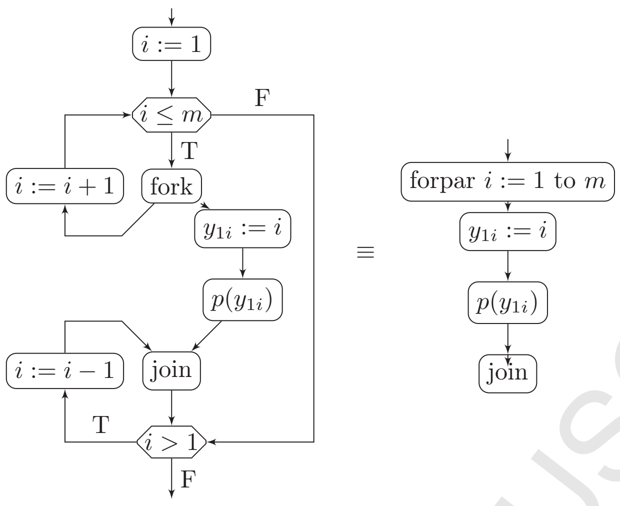


Figure 7: Implementing **forpar** using binary fork and join

of the automaton. To obtain a deterministic automaton from a deterministic flowchart, a junk state can be added as a target for all actions that do not lead to an accepted computation. Forks and joins of flowcharts correspond to forks and joins in Lodaya and Weil's branching automata, but note that a k-ary join node in the flowchart expands into k states in the automaton (see Figure 9).

We now define a CKAT model that provides trace semantics for concurrent flowchart schemes. Let $N = \{x_i, y_i, z_i : i = 1, 2, 3, \dots\}$ be a namespace of variables and let V be a set of values (e. g. $V = \mathbb{Z}$). The set of states is

$$X = \{s : s \text{ is a partial function from } N \text{ to } V \text{ with finite domain}\}.$$

Thus a state $s \in X$ specifies the values for a finite set $D = \mathrm{dom}(s)$ of variables. As in separation logic [25, 22], states r, s are said to be *separated* if $\mathrm{dom}(r) \cap \mathrm{dom}(s) = \emptyset$, denoted $r \perp s$. Recall that X^{sp} is the set of all sp-strings over the set X. An sp-string is called an *sp-trace* if

1. its underlying poset has a largest and a smallest element,
2. any two incomparable states are separated, and
3. if $s_1, s_2, \dots, s_k$ are all the covers or all the co-covers of state r then $\mathrm{dom}(r) = \mathrm{dom}(s_1) \cup \dots \cup \mathrm{dom}(s_k)$.

The *trace semantics* of a concurrent flowchart scheme p is the set $[p]$ of all sp-traces that are finite execution traces of the flowchart. For a flowchart that is expressible as a CKAT algebra term, $[p]$ can be calculated by evaluating the term in the following way.

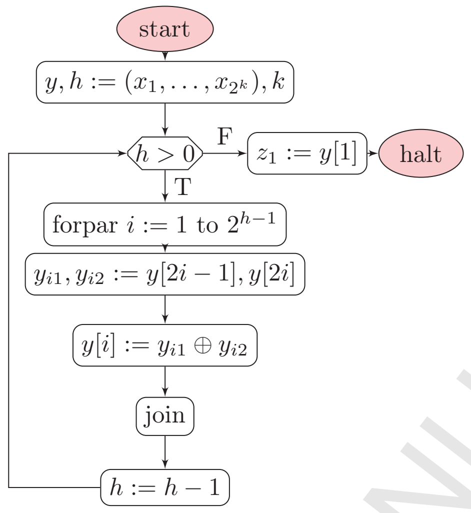

start
$y, h := (x_1, \ldots, x_{2^k}), k$
$h > 0$
F
T
$z_1 := y[1]$
halt
forpar $i := 1$ to 2^{h-1}
$y_{i1}, y_{i2} := y[2i-1], y[2i]$
$y[i] := y_{i1} \oplus y_{i2}$
join
$h := h - 1$

Figure 8: Using **forpar** to calculate $x_1 \oplus x_2 \oplus \cdots \oplus x_{2^k}$

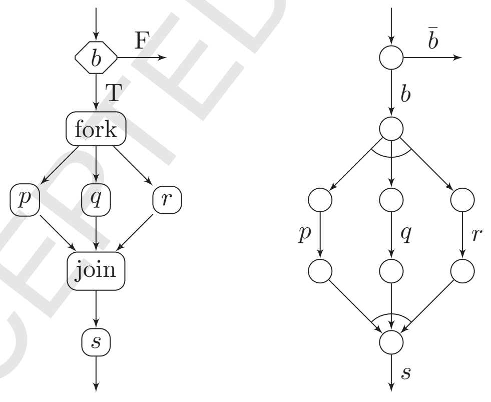


Figure 9: Correspondence between flowcharts and automata

For an assignment such as $y := t(x_1, \ldots, x_n)$, the semantics are

$$[y := t(\mathbf{x})] = \{(s, s') \in X^2 : \mathbf{x} \in \mathrm{dom}(s) = \mathrm{dom}(s') \text{ and } s' = s[y \mapsto t(s(x_1), \ldots, s(x_n))\}.$$

For a test $P(y_1, \ldots, y_n)$, the semantics are a set of length-one sequences

$$[P(y_1, \ldots, y_n)] = \{(s) \in X^1 : y_1, \ldots, y_n \in \mathrm{dom}(s) \text{ and } P(s(y_1), \ldots, s(y_n))\}.$$

A sequence of states $(s_1, \ldots, s_n)$ is also written simply as $s_1 s_2 \ldots s_n$ and is called a *linear* trace. As for gsp-strings, sequential composition of sp-traces uses the coalesced product $\diamond$ which is well-defined since each sp-trace has a first and a last element:

$$rur' \diamond svs' = \begin{cases} rusvs' & \text{if } r' = s \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The concurrent composition is based on a *separated* product:

$$rur'|svs' = \begin{cases} (r \cup s)(u||v)(r' \cup s') & \text{if } r \perp s \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Note that here $||$ is the parallel composition of sp-strings from Section 5. The definitions for some of the less obvious special cases are $r \diamond s = r$ if $r = s$, else undefined; $rr'|svs' = (r \cup s)v(r' \cup s')$ if $r \perp s$; and $r|svs' = rr|svs'$. The associativity and commutativity of the operation $|$ is easily checked.

Let X^{spt} be the set of all sp-traces. As usual, one now extends the above two operations to subsets by

- $R \cdot S = \{v \diamond w : v \in R, w \in S \text{ and } v \diamond w \text{ is defined}\}$
- $R||S = \{w : u \mid v \sqsubseteq w \text{ for some } u \in R, v \in S \text{ and } u \mid v \text{ is defined}\}$
- $R + S = R \cup S$
- $0 = \emptyset$, $1 = X^1$, $\overline{B} = X^1 \setminus B$ and
- $R^* = \bigcup_{n<\omega} R^n$.

With these operations one obtains the CKAT $(\mathcal{U}p(X^{spt}), \mathcal{U}p(X), +, 0, ||, \cdot, 1, ^*, ^-)$. By first choosing an interpretation I for the function symbols and predicate symbols, and then generating the subalgebra of all assignment statements and tests, one obtains the algebra of all concurrent programs generated by the functions and tests of the interpretation I.

A subset of X^{spt} determines an obvious binary relation, by mapping each sp-trace svs' to the pair (s, s'). This map from trace semantics to denotational semantics is a homomorphism from a CKAT to a relational Kleene algebra, where the tests are subsets of the identity. If one starts out with a sequential program and modifies it to run concurrently on a multicore processor or a distributed system, then this homomorphism is useful for checking that the concurrent version and the sequential version still satisfy the same input/output relation.

The algebra of binary relations in the previous paragraph can also be constructed directly (and more generally) by starting with any positive separation algebra $(\Gamma, \oplus, \mathbf{u})$ and defining partial operations for sequential and parallel products on $\Gamma^2 = \Gamma \times \Gamma$ as follows:

$$(a,b) \diamond (c,d) = \begin{cases} (a,d) & \text{if } b = c \\ \text{undefined} & \text{otherwise} \end{cases}$$

$$(a,b) \mid (c,d) = \begin{cases} (a \oplus c, b \oplus d) & \text{if } a \perp c, b \perp d \\ \text{undefined} & \text{otherwise.} \end{cases}$$

The subsumption order $(a,b) \sqsubseteq (c,d)$ is given by the pointwise natural order, and the partial operations $\diamond, \mid$ are lifted to $\mathcal{U}p(\Gamma^2)$ in the same way as in the preceding gsp-string models. The proof of our last result is very similar to the proof of Theorem 4.

Theorem 7. *For any positive separation algebra* $(\Gamma, \oplus, \mathbf{u})$*, the two-sorted algebra* $(\mathcal{U}p(\Gamma^2), \mathcal{U}p(id_\Gamma), \cup, \emptyset, ||, \cdot, 1, ^*, ^-)$ *is a* CKAT.

As mentioned at the end of the introduction, Hoare et. al. [11] also use a separation algebra to define their Resource Model. An interesting topic of future research is to compare this predicate transformer model to the relational model given here.

9. Conclusion

Many theoretical models of concurrency have been proposed and studied during the last five decades. Here we have taken an algebraic approach starting from Kleene algebra with tests and adapting this model to concurrent Kleene algebra and bounded-width series-parallel language models. This provides semantics for concurrency based on standard notions such as regular languages and automata. The addition of tests allows KAT to express standard imperative programming constructs such as `if-then-else` and `while-do`. Adding concurrency into this elegant algebraic model is likely to lead to new applications such as verifying compiler optimizations targeting multicore architectures or modeling computations on large distributed clusters. In recent years, several programming languages have added fork-join commands (e.g. Java) or related constructs such as spawn-sync (e.g. cilk) or async (c++11). Concurrent deterministic flowchart schemes are closely related to concurrent Kleene algebras with test, and are able to express a variety of concurrent algorithms in a general formalism that can be adapted to many types of parallel hardware, from multicore processors to distributed computing clusters.

We would like to thank the anonymous referees for many useful comments that have substantially improved this article.

[1] Angus, A., Kozen, D.: Kleene algebra with tests and program schematology. Technical Report 2001-1844, Computer Science Department, Cornell University (2001).

[2] Balbes, R., Dwinger, P.: Distributive Lattices, University of Missouri Press, 1974.

[3] Bloom, S. L., Esik, Z.: Free shuffle algebras in language varieties. Theoretical Computer Science, 163 (1996) 55–98.

[4] Bonchi, F., Bonsangue, M. M., Hansen, H. H., Panangaden, P., Rutten, J. J. M. M., Silva, A.: Algebra-coalgebra duality in Brzozowski's minimization algorithm. ACM Trans. on Computational Logic, 15 (2014), no. 1, 29.

[5] Calcagno, C., O'Hearn, P. W., Yang, H.: Local action and abstract separation logic. Proceedings of the 22nd Annual IEEE Symposium on Logic in Computer Science, 2007, 366–378.

[6] Foulis, D. J., and Bennett, M. K.: Effect algebras and unsharp quantum logics, Foundations of Physics 24, 10, 1994, 1331–1352.

[7] Gischer, L.: The equational theory of pomsets. Theoretical Computer Science 62 (1988) 299–224.

[8] Grabowski, J.: On partial languages. Ann. Soc. Math. Polon. Ser. IV Fund. Math. IV(2) (1981), 427–498.

[9] Hoare, C. A. R., Möller, B., Struth, G., Wehrman, I.: Foundations of concurrent Kleene algebra. Relations and Kleene algebra in computer science, Lecture Notes in Comput. Sci., 5827, Springer, 2009, 166–186.

[10] Hoare, C. A. R., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra. CONCUR 2009—concurrency theory, Lecture Notes in Comput. Sci., 5710, Springer, 2009, 399–414.

[11] Hoare, C. A. R., Hussain, A., Möller, B., O?Hearn, P. W., Petersen, R. L., Struth, G.: On locality and the exchange law for concurrent processes. CONCUR 2011—concurrency theory, Lecture Notes in Comput. Sci., 6901, Springer, 2011, 250–264.

[12] Hoare, C. A. R., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra and its foundations. J. Log. Algebr. Program. 80 (2011), no. 6, 266–296.

[13] Hopcroft, J., Ullman, J.: Introduction to Automata Theory, Languages and Computation. Addison-Wesley, Reading, MA, 1979.

[14] Ianov, Y. I.: The logical schemes of algorithms. In Problems of Cybernetics, Vol. 1, Pergamon Press, 1960, 82–140.

[15] Kozen, D.: A completeness theorem for Kleene algebras and the algebra of regular events, Infor. and Comput., 110:2, 1994, 366–390.

[16] Kozen, D.: Automata on guarded strings and applications. 8th Workshop on Logic, Language, Informations and Computation—WoLLIC'2001 (Brasília). Mat. Contemp. 24 (2003), 117–139.

[17] Kozen, D.: On the representation of Kleene algebras with tests. Mathematical foundations of computer science 2006, Lecture Notes in Comput. Sci., 4162, Springer, 2006, 73–83.

[18] Kozen, D., Smith, F.: Kleene algebra with tests: completeness and decidability. Computer science logic (Utrecht, 1996), Lecture Notes in Comput. Sci., 1258, Springer, 1997, 244–259.

[19] Kozen, D., Tiuryn, J.: Substructural logic and partial correctness. ACM Trans. Computational Logic, 4(3) (2003) 355–378.

[20] Lodaya, K., Weil, P.: Series-parallel posets: algebra, automata and languages. In proceedings of STACS 98, Lecture Notes in Comput. Sci., 1373, Springer, 1998, 555–565.

[21] Lodaya, K., Weil, P.: Series-parallel languages and the bounded-width property. Theoret. Comput. Sci. 237 (2000), no. 1-2, 347–380.

[22] O'Hearn, P. W.: Resources, Concurrency, and Local Reasoning. Invited paper, CONCUR 2004, London, Lecture Notes in Comput. Sci., 3170, Springer, 2004, 49–67.

[23] Pratt, V.: Modelling concurrency with partial orders. Internat. J. Parallel Prog. 15 (1) (1986) 33–71.

[24] Prisacariu, C.: Synchronous Kleene algebra. J. of Logic and Algebraic Prog. 79 (2010) 608–635.

[25] Reynolds, J. C.: Separation logic: a logic for shared mutable data structures. Invited paper, Proc. 17th IEEE Conference on Logic in Computer Science, LICS 2002, 55–74.