Synchronous Kleene Algebra vs. Concurrent Kleene Algebra

Cristian Prisacariu

Department of Informatics, University of Oslo,
P.O. Box 1080 Blindern, N-0316 Oslo, Norway.
E-mail: cristi@ifi.uio.no

Abstract. In this year's CONCUR conference Concurrent Kleene Algebra (CKA) is presented as a general formalism for reasoning about concurrent programs. Also recently *Synchronous Kleene Algebra* ($\mathcal{SKA}$) was investigated by this author with the purpose of representing and reasoning about actions/programs that have a notion of concurrency in the style of synchrony of the SCCS calculus.[1] CKA has, at first sight, striking resemblances with $\mathcal{SKA}$. We discuss this model of concurrency in relation with $\mathcal{SKA}$. Our discussion focuses on the underlying ideas and intuitions of the two models. The discussion is also casted into the partial orders model of concurrency to get more insights into the two algebras.

1 Introduction

Kleene algebra formalizes axiomatically the structures of regular expressions and finite automata (see e.g. [1, 4]). Kleene algebra with tests [5] combines Kleene algebra with a Boolean algebra and can encode the propositional Hoare logic, therefore it can model while programs. In one form or another, Kleene algebras appear in various formalisms in computer science: relation algebras, logics of programs and in particular propositional dynamic logic, regular expressions and formal language theory.

Synchrony is a model for concurrency which was introduced in the process algebra community in Milner's SCCS [6], but detaches form the general interleaving approach. On the other hand it is a concept which does not belong to the partial order model of concurrency either. The synchrony concept proves highly expressive and robust; SCCS can represent CCS (i.e. asynchrony) as a sub-calculus, and a great number of synchronizing operations can be defined in terms of the basic SCCS operators.

The notion of *synchrony* has different meanings in different areas of computer science. Here we take the distinction between synchrony and asynchrony as presented in the SCCS calculus and implemented in e.g. the Esterel synchronous programming language. We understand *asynchrony* as when two concurrent systems execute at indeterminate relative speeds (i.e. their actions may have different noncorelated durations); whereas in the *synchrony* model each of the two concurrent systems execute instantaneously a single action at each time step. The reasoning is governed by the assumption of a global clock and an assumption of eagerness (i.e. at each time step all possible concurrent actions are performed). The SCCS concurrency operator $\times$ over processes is different from the classical $\|$ of CCS.

[1] A preliminary version of $\mathcal{SKA}$ was presented in [9] with details available in [8].

2 Discussions

Synchronous Kleene algebra combines synchrony and Kleene algebra. $\mathcal{SKA}$ is a σ-algebra with signature $\sigma = \{+, \cdot, \times, ^*, \mathbf{0}, \mathbf{1}, \mathcal{A}_B\}$. The non-constant functions of σ are: "$+$" for *choice* of two actions, "$\cdot$" for *sequence* of two actions (or concatenation), "$\times$" for *synchronous composition* of two actions, and "*" to model *repeated execution* of one action. The axioms of $\mathcal{SKA}$ are those of Kleene algebra together with eight more for the synchrony operator $\times$ making it commutative, associative, with identity element $\mathbf{1}$, zero element $\mathbf{0}$, and distributive over $+$ in both arguments. Two more special axioms are the idempotence over the *basic actions* of $\mathcal{A}_B$ and the synchrony axiom:

$$(\alpha_\times \cdot \alpha) \times (\beta_\times \cdot \beta) = (\alpha_\times \times \beta_\times) \cdot (\alpha \times \beta) \quad \forall \alpha_\times, \beta_\times \in \mathcal{A}_B^\times \tag{1}$$

Recently Concurrent Kleene algebra (CKA) was proposed in [3].[2] CKA is defined as two quantales $(S, +, ;, 0, 1)$ and $(S, +, *, 0, 1)$ related by an exchange axiom.[3] Quantales are idempotent semirings which are also complete lattices under the natural order $\leq$ of the semiring (i.e. have the extra constraint of a top element). The natural order is defined as $\alpha \leq \beta \triangleq \alpha + \beta = \beta$. Similarly, $\mathcal{SKA}$ forms also two idempotent semirings; and both $\mathcal{SKA}$ and CKA have $\times$ and $*$ commutative. What differentiates the two is the synchrony axiom on the one hand and the exchange axiom on the other, and as we see later, also the choice of models.

Both algebras can model Hoare-style reasoning about sequential programs. On top, both algebras can reason about some form of concurrent programs: CKA can model Jone's rely/guarantee calculus, whereas $\mathcal{SKA}$ can reason in the style of Qwicki and Gries (i.e., shared-variables concurrency and interference) about synchronous programs.

CKA theory relies heavily on the natural order $\leq$; the exchange axiom is defined in terms of it. Generally, an intuitive understanding of the natural order of a semiring is that $\leq$ states that the left operand has less behavior than the right operand, or in other words, the right operand specifies behavior which includes all the behavior specified by the left operand (and possibly more). The exchange axiom entails two properties of CKA relevant for our discussion:

$$(\alpha * \beta); (\alpha' * \beta') \leq (\alpha; \alpha') * (\beta; \beta') \tag{2}$$

$$\alpha; \beta \leq \alpha * \beta \tag{3}$$

Equation (2) is similar to the synchrony axiom. It is more general because it considers α and β general actions and not only synchronous actions like $\alpha_\times, \beta_\times \in \mathcal{A}_B^\times$ (i.e. the basic actions closed only under $\times$). On the other hand it is less informative than the synchrony axiom because it only states inclusion of behaviors and not equality. One may read (2) as: "All behaviors coming from putting two concurrent compositions in sequence are captured by putting the respective sequences in concurrent composition."

Equation (3) does not hold in $\mathcal{SKA}$ and has no similar counterpart either. In $\mathcal{SKA}$ sequence composition and synchronous composition of two complex actions have different behaviors. Equation (3) states that the concurrent composition captures all the

[2] See the related technical reports at `http://www.informatik.uni-augsburg.de/de/forschung/reports/`

[3] ; and $*$ correspond to respectively $\cdot$ and $\times$ in $\mathcal{SKA}$.

behavior of the sequential composition. This is the same as in the "concurrency as interleaving" approach where all the behaviors coming from all the possible interleavings are contained in the concurrent composition. In CKA one has this because of the commutativity of $*$ ($\alpha;\beta+\beta;\alpha \leq \alpha * \beta$).

The closest relation to interleaving that $\mathcal{SKA}$ can make is with the *shuffle* operator from regular languages which has a position between the interleaving approach and the partial orders approach to concurrency. If we were to ignore the branching information in our actions then we can view $\times$ as an *ordered shuffle*. The shuffling of two sequences of actions in $\mathcal{SKA}$ walks step by step (on the $\cdot$ operation) and shuffles the basic actions found (locally).

Now we take a look at the models of the two algebraic formalisms and also relate them with the partial orders model.

For CKA the models are given as elements of $\mathcal{P}(\mathcal{P}(E))$ where E is a set of events equipped with a dependency relation $\rightarrow$ (no transitivity or acyclicity requirements as with partial orders). A model is a set of traces, where a trace is just a set of events. For $\mathcal{SKA}$ the *standard interpretation* of the actions is given by defining a homomorphism $\hat{I}_{\mathcal{SKA}}$ which takes any action of $\mathcal{SKA}$ into a corresponding *regular set of synchronous strings* (i.e. a subset of strings from $\mathcal{P}(\mathcal{A}_B)^*$).

Synchronous strings, as related to the traces of CKA, also have a dependency relation which is a restricted partial order. We now isolate a subclass of pomsets which define exactly this partial order. Later we come back to CKA and our discussion.

The theory of pomsets [7] is among the first in concurrency theory to make a distinction between events E and actions A. A pomset is a partially ordered set of events labeled non-injective by actions. Pomsets extend the idea of strings, which are linearly ordered multisets, to partially ordered multisets. Formally, a *pomset* is the isomorphism class (w.r.t. the events) of the structure $(E, A, <, \mu)$ where $\mu : E \rightarrow A$ labels events by action names. Two events that are incomparable by the partial order $<$ are permitted to occur concurrently. A pomset describes only one execution of the concurrent system.[4] Pomsets are more expressive then our synchronous strings.

Theorem 1. *Synchronous strings are completely characterized by synchronous pomsets. A synchronous pomset is a pomset where the partial order respects the constraint:*

all maximal independent sets *are disjoint,* uniquely labeled, *and* completely ordered,

where an independent set of events *is* $X \subseteq E$ *s.t.* $\forall e_i, e_j \in X$ *then* $(e_i \not< e_j) \wedge (e_j \not< e_i)$. *An independent set is* uniquely labeled *iff the labeling function is injective on* X*; i.e.* $\mu|_X$ *is injective. Two independent sets* X_i, X_j *are* completely ordered *iff if* $\exists e_i \in X_i,\ \exists e_j \in X_j$ *with* $e_i < e_j$ *then* $\forall e_i \in X_i,\ \forall e_j \in X_j$ *is the case that* $e_i < e_j$.

We can find operations on synchronous pomsets corresponding to the $\mathcal{SKA}$'s $+$, $\cdot$, and *. For the $\times$ of $\mathcal{SKA}$ we did not find a straightforward equivalent for synchronous pomsets. Moreover, we are not sure if there is a *pomset definable* operation (as in terminology of [2]). The first candidate was the concurrency operation $||$, but this breaks the *completely ordered* requirement.

[4] The same as a trace in CKA and a synchronous string in $\mathcal{SKA}$.

The pomset concurrency operation $||$ behaves like the $*$ of CKA, meaning that $||$ combines two pomsets with disjoint events by making the union of the events and the union of the partial orders (no new dependencies are added). Similarly, in CKA one can view two traces as having their own dependency relation (as $\rightarrow$ restricted to each trace) and $*$ makes union of their events with no restriction on the dependency relations.

In CKA the dependency relation is not manipulated, it is given. CKA processes specify subsets of events, and each subset has attached the predefined $\rightarrow$ restricted to its events only. In $\mathcal{SKA}$ and pomsets the partial order is changed with each application of an operator; e.g., sequential composition adds dependencies. The approach of CKA is similar to that of separation logic where one reasons about a big (given) program by separating it into smaller independent programs. On the other hand the partial orders model has a constructivist view where big programs are constructed from smaller programs (i.e. the partial order is constructed).

The sequential composition of two traces in CKA returns the union of the events only if there is no dependency from the events of the left operand to events of the right operand. In the constructivist view of pomsets and $\mathcal{SKA}$ two pomsets are composed sequentially by adding a dependence between any event of the right operand and all the events of the left operand. This implies the same constraint as CKA imposes. In $\mathcal{SKA}$ we achieve the separation of the dependencies by construction, whereas in CKA it is achieved by choosing the separation depending on the predefined $\rightarrow$ relation.

Concluding, we remind that the purpose of our discussions was to get a better understanding of the underlying intuitions behind CKA in relation to the ideas behind $\mathcal{SKA}$ and pomsets. As a last remark, it would have been interesting to also cast CKA into pomsets because we would thus have a common formal ground for comparison. But this is not immediate because the dependency relation of the CKA model does not have the constraints of a partial order (i.e. is more general).

References

1. J. H. Conway. *Regular Algebra and Finite Machines*. Chapman and Hall, 1971.
2. J. L. Gischer. *Partial Orders and the Axiomatic Theory of Shuffle*. PhD thesis, CS, Stanford University, 1984.
3. T. Hoare, B. Möller, G. Struth, and I. Wehrman. Concurrent Kleene Algebra. In *20th International Conference on Concurrency Theory (CONCUR'09)*, LNCS. Springer, 2009.
4. D. Kozen. A completeness theorem for kleene algebras and the algebra of regular events. *Information and Computation*, 110(2):366–390, 1994.
5. D. Kozen. Kleene Algebra with Tests. *ACM TOPLAS'97*, 19(3):427–443, 1997.
6. R. Milner. Calculi for synchrony and asynchrony. *TCS*, 25:267–310, 1983.
7. V. R. Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.
8. C. Prisacariu. Extending Kleene Algebra with Synchrony. Technical Report 376, Univ. Oslo, October 2008.
9. C. Prisacariu. Extending Kleene Algebra with Synchrony: Completeness and Decidability. In T. Uustalu and J. Vain, editors, *20th Nordic Workshop on Programming Theory*, 2008.