# Concurrent Kleene Algebra

C.A.R. Hoare[1], B. Möller[2], G. Struth[3], and I. Wehrman[4]

[1] Microsoft Research, Cambridge, UK
[2] Universität Augsburg, Germany
[3] University of Sheffield, UK
[4] University of Texas at Austin, USA

**Abstract.** A concurrent Kleene algebra offers, next to choice and iteration, operators for sequential and concurrent composition, related by an inequational form of the exchange law. We show applicability of the algebra to a partially-ordered trace model of program execution semantics and demonstrate its usefulness by validating familiar proof rules for sequential programs (Hoare triples) and for concurrent ones (Jones's rely/guarantee calculus). This involves an algebraic notion of invariants; for these the exchange inequation strengthens to an equational distributivity law. Most of our reasoning has been checked by computer.

## 1 Introduction

Kleene algebra [7] has been recognised and developed [18, 19, 8] as an algebraic framework (or structural equivalence) that unifies diverse theories for conventional sequential programming by axiomatising the fundamental concepts of choice, sequential composition and finite iteration. Its many familiar models include relations under union, relational composition and reflexive transitive closure, as well as formal languages under union, concatenation and Kleene star. This paper defines a 'double' Kleene algebra, which adds an operator for concurrent composition. Sequential and concurrent composition are related by an inequational weakening of the equational exchange law $(a \circ b) \bullet (c \circ d) = (a \bullet c) \circ (b \bullet d)$ of two-category or bicategory theory (e.g. [20]). Under certain conditions, this is strengthened to an equational law, by which concurrent composition distributes through sequential. The axioms proposed for a concurrent Kleene algebra are catalogued in Section 4.

The interest of concurrent Kleene algebra (CKA) is two-fold. Firstly, it expresses only the essential properties of program execution; indeed, it represents just those properties which are preserved even by architectures with weakly-ordered memory access, unreliable communications and massively re-ordering program optimisers. Secondly, the modelled properties, though unusually weak, are strong enough to validate the main structural laws of assertional reasoning about program correctness, both in sequential style [12] (as described in Section 5) and in concurrent style [17] (as described in Section 8).

The purpose of the paper is to introduce the basic operations and their laws, both in a concrete representation and in abstract, axiomatic form. We hope in future research to relate CKA to various familiar process algebras, such as the

$\pi$-calculus or CSP, and to clarify the links between their many variants.

Before we turn to the abstract treatment, Section 2 presents our weak semantic model which also is a concrete model of the notion of a CKA. A program is identified with the set of traces of all the executions it may evoke. Each trace consists of the set of events that occur during a single execution. When two subprograms are combined, say in a sequential or a concurrent combination, each event that occurs is an event in the trace of exactly one of the subprograms. Each trace of the combination is therefore the disjoint union of a trace of one of the sub-programs with a trace of the other. Our formal definitions of the program combinators identify them as a kind of separating conjunction [25].

We use a primitive dependence relation between the events of a trace. Its transitive closure represents a direct or indirect chain of dependence. In a sequential composition, it is obviously not allowed for an event occurring in execution of the first operand to depend (directly or indirectly) on an event occurring in execution of the second operand. We take this as our definition of a very weak form of sequential composition. Concurrent composition places no such restriction, and allows dependence in either direction. The above-mentioned exchange law seems to generally capture the interrelation between sequential and concurrent composition in adequate inequational form.

The dependence primitive is intended to model a wide range of computational phenomena, including control dependence (arising from program structure) and data dependence (arising from flow of data). There are many forms of data flow. Flow of data across time is usually mediated by computer memory, which may be private or shared, strongly or only weakly consistent. Flow of data across space is usually mediated by a real or simulated communication channel, which may be buffered or synchronised, double-ended or multiplexed, reliable or lossy, and perhaps subject to stuttering or even re-ordering of messages.

Obviously, it is only weak properties of a program that can be proved without knowing more of the properties of the memory and communication channels involved. The additional properties are conveniently specified by additional axioms, like those used by hardware architects to describe specific weak memory models (e.g. [22]). Fortunately, as long as they are consistent with our fundamental theory, they do not invalidate our development and hence do not require fresh proofs of any of our theorems.

In this paper we focus on the basic concrete CKA model and the essential laws; further technical details are given in the companion paper [15]. The formal proofs of our results can be found in [14]; there we also show a typical input file for the automated theorem prover Prover9 [28] with which all the purely algebraic proofs have been reconstructed automatically.

## 2 Operators on Traces and Programs

We assume a set $EV$ of *events*, i.e., occurrences of primitive actions, and a *dependence relation* $\rightarrow \subseteq EV \times EV$ between them: $p \rightarrow q$ indicates occurrence of a data flow or control flow from event $p$ to event $q$.

A *trace* is a set of events. We use *sets* of events to have greater flexibility, in particular, to accommodate non-interleaving semantics; the more conventional *sequences* can be recovered by adding time stamps or the like to events. A *program* is a set of traces. For example, the program skip, which does nothing, is defined as $\{\emptyset\}$, and the program $[p]$, which has the only event $p$, is $\{\{p\}\}$. The program $\mathsf{false} =_{df} \emptyset$ has no traces, and therefore cannot be executed at all. It serves the rôle of the 'miracle' [23] in the development of programs by stepwise refinement. We have $\mathsf{false} \subseteq P$ for all programs $P$.

Following [16] we will define four operators on programs $P$ and $Q$:

- $P * Q$ fine-grain concurrent composition, allowing dependences between $P$ and $Q$;
- $P \,;\, Q$ weak sequential composition, forbidding dependence of $P$ on $Q$;
- $P \parallel Q$ disjoint parallel composition, with no dependence in either direction;
- $P \,[]\, Q$ alternation – exactly one of $P$ or $Q$ is executed, whenever possible.

For the formal definition let $\rightarrow^+$ be the transitive closure of the dependence relation $\rightarrow$ and let, for trace $tp$, be $dep(tp) =_{df} \{q \mid \exists\, p \in tp : q \rightarrow^+ p\}$. Thus, $dep(tp)$ is the set of events on which some event in $tp$ depends. Therefore, trace $tp$ is independent of trace $tq$ iff $dep(tp) \cap tq = \emptyset$. The use of the transitive closure $\rightarrow^+$ seems intuitively reasonable; an algebraic justification is given in Section 7.

**Definition 2.1** Consider the schematic combination function

$$\mathrm{COMB}(P, Q, C) =_{df} \{tp \cup tq \mid tp \in P \,\wedge\, tq \in Q \,\wedge\, tp \cap tq = \emptyset \,\wedge\, C(tp, tq)\}$$

with programs $P, Q$ and a predicate $C$ in the trace variables $tp$ and $tq$. Then the above operators are given by

$$\begin{array}{lll} P * Q & =_{df} & \mathrm{COMB}(P, Q, \mathsf{TRUE}) \;, \\ P \,;\, Q & =_{df} & \mathrm{COMB}(P, Q,\ dep(tp) \cap tq = \emptyset) \;, \\ P \parallel Q & =_{df} & \mathrm{COMB}(P, Q,\ dep(tp) \cap tq = \emptyset \,\wedge\, dep(tq) \cap tp = \emptyset) \;, \\ P \,[]\, Q & =_{df} & \mathrm{COMB}(P, Q,\ tp = \emptyset \,\vee\, tq = \emptyset) \;. \end{array}$$

**Example 2.2** We illustrate the operators with a mini-example. We assume a set $EV$ of events the actions of which are simple assignments to program variables. We consider three particular events $ax, ay, az$ associated with the assignments $x := x+1, y := y+2, z := x+3$, resp. There is a dependence arrow from event $p$ to event $q$ iff $p \neq q$ and the variable assigned to in $p$ occurs in the assigned expression in $q$. This means that for our three events we have exactly $ax \rightarrow az$. We form the corresponding single-event programs $P_x =_{df} [ax], P_y =_{df} [ay], P_z =_{df} [az]$. To describe their compositions we extend the notation for single-event programs and set $[p_1, \ldots, p_n] =_{df} \{\{p_1, \ldots, p_n\}\}$ (for uniformity we sometimes also write $[\,]$ for skip). Figure 1 lists the composition tables for our operators on these programs. They show that the operator $*$ allows forming parallel programs with race conditions, whereas ; and $\parallel$ respect dependences. □

It is straightforward from the definitions that $*, \parallel$ and $[]$ are commutative and that $[] \subseteq \parallel \subseteq ; \subseteq *$ where for $\circ, \bullet \in \{*, ;, \parallel, []\}$ the formula $\circ \subseteq \bullet$ abbreviates $\forall\, P, Q : P \circ Q \subseteq P \bullet Q$. Further useful laws are the following.

| $*$ | $P_x$ | $P_y$ | $P_z$ |
|---|---|---|---|
| $P_x$ | $\emptyset$ | $[ax, ay]$ | $[ax, az]$ |
| $P_y$ | $[ax, ay]$ | $\emptyset$ | $[ay, az]$ |
| $P_z$ | $[ax, az]$ | $[ay, az]$ | $\emptyset$ |

| $;$ | $P_x$ | $P_y$ | $P_z$ |
|---|---|---|---|
| $P_x$ | $\emptyset$ | $[ax, ay]$ | $[ax, az]$ |
| $P_y$ | $[ax, ay]$ | $\emptyset$ | $[ay, az]$ |
| $P_z$ | $\emptyset$ | $[ay, az]$ | $\emptyset$ |

| $\|\|$ | $P_x$ | $P_y$ | $P_z$ |
|---|---|---|---|
| $P_x$ | $\emptyset$ | $[ax, ay]$ | $\emptyset$ |
| $P_y$ | $[ax, ay]$ | $\emptyset$ | $[ay, az]$ |
| $P_z$ | $\emptyset$ | $[ay, az]$ | $\emptyset$ |

| $[]$ | $P_x$ | $P_y$ | $P_z$ |
|---|---|---|---|
| $P_x$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $P_y$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |
| $P_z$ | $\emptyset$ | $\emptyset$ | $\emptyset$ |

**Fig. 1.** Composition tables

**Lemma 2.3** *Let* $\circ, \bullet \in \{*, ;, \|, []\}$.

1. $\circ$ *distributes through arbitrary unions; in particular,* $\mathsf{false}$ *is an annihilator for* $\circ$*, i.e.,* $\mathsf{false} \circ P = \mathsf{false} = P \circ \mathsf{false}$*. Moreover,* $\circ$ *is isotone w.r.t.* $\subseteq$ *in both arguments.*
2. $\mathsf{skip}$ *is a neutral element for* $\circ$*, i.e.,* $\mathsf{skip} \circ P = P = P \circ \mathsf{skip}$*.*
3. *If* $\bullet \subseteq \circ$ *and* $\circ$ *is commutative then* $(P \circ Q) \bullet (R \circ S) \subseteq (P \bullet R) \circ (Q \bullet S)$*.*
4. *If* $\bullet \subseteq \circ$ *then* $P \bullet (Q \circ R) \subseteq (P \bullet Q) \circ R$*.*
5. *If* $\bullet \subseteq \circ$ *then* $(P \circ Q) \bullet R \subseteq P \circ (Q \bullet R)$*.*
6. $\circ$ *is associative.*

The proofs either can be done by an easy adaptation of the corresponding ones in [16] or follow from more general results in [15]. A particularly important special case of Part 3 is the exchange law

$$(P * Q) ; (R * S) \subseteq (P ; R) * (Q ; S) \tag{1}$$

In the remainder of this paper we shall mostly concentrate on the more interesting operators $*$ and $;$.

Another essential operator is union which again is $\subseteq$-isotone and distributes through arbitrary unions. However, it is *not* $\mathsf{false}$-strict.

By the Tarski-Kleene fixpoint theorems all recursion equations involving only the operators mentioned have $\subseteq$-least solutions which can be approximated by the familiar fixpoint iteration starting from $\mathsf{false}$. Use of union in such a recursions enables non-trivial fixpoints, as will be seen in the next section.

## 3 Quantales, Kleene and Omega Algebras

We now abstract from the concrete case of programs and embed our model into a more general algebraic setting.

**Definition 3.1** A *semiring* is a structure $(S, +, 0, \cdot, 1)$ such that $(S, +, 0)$ is a commutative monoid, $(S, \cdot, 1)$ is a monoid, multiplication distributes over addition in both arguments and 0 is a left and right annihilator with respect to multiplication ($a \cdot 0 = 0 = 0 \cdot a$). A semiring is *idempotent* if its addition is.

The operation + denotes an abstract form of nondeterministic choice; in the concrete case of programs it will denote union (of sets of traces). This explains why + is required to be associative, commutative and idempotent. Its neutral element 0 will take the rôle of the miraculous program $\emptyset$.

In an idempotent semiring, the relation $\leq$ defined by $a \leq b \Leftrightarrow_{df} a + b = b$ is a partial ordering, in fact the only partial ordering on $S$ for which 0 is the least element and for which addition and multiplication are isotone in both arguments. It is therefore called the *natural ordering* on $S$. This makes $S$ into a semilattice with addition as join and least element 0.

**Definition 3.2** A *quantale* [24] or *standard Kleene algebra* [7] is an idempotent semiring that is a complete lattice under the natural order and in which multiplication distributes over arbitrary suprema. The infimum and the supremum of a subset $T$ are denoted by $\sqcap T$ and $\sqcup T$, respectively. Their binary variants are $x \sqcap y$ and $x \sqcup y$ (the latter coinciding with $x + y$).

In particular, quantale composition is *continuous*, i.e., distributes through suprema of arbitrary, not just countable, chains. As an idempotent semiring, every quantale has 0 as its least element. As a complete lattice, it also has a greatest element $\top$. Quantales have been used in many contexts other than that of program semantics, see e.g. the c-semirings of [3] or the general reference [29].

Let now $PR(EV) =_{df} \mathcal{P}(\mathcal{P}(EV))$ denote the set of all programs over the event set $EV$. From the observations in Section 2 the following is immediate:

**Lemma 3.3** *$(PR(EV), \cup, \mathsf{false}, *, \mathsf{skip})$ and $(PR(EV), \cup, \mathsf{false}, ;, \mathsf{skip})$ are quantales. In each of them $\top = \mathcal{P}(EV)$ is the most general program over $EV$.*

**Definition 3.4** In a quantale $S$, the finite and infinite iterations $a^*$ and $a^\omega$ of an element $a \in S$ are defined by $a^* = \mu x \,.\, 1 + a \cdot x$ and $a^\omega = \nu x \,.\, a \cdot x$, where $\mu$ and $\nu$ denote the least and greatest fixpoint operators. The star here should not be confused with the separation operator $*$ above.

It is well known that $(S, +, 0, \cdot, 1, ^*)$ forms a Kleene algebra [18]. From this we obtain many useful laws for free. As examples we mention

$$1 \leq a^* \;, \quad a \leq a^* \;, \quad a^* \cdot a^* = (a^*)^* = a^* \;, \quad (a + b)^* = a^* \cdot (b \cdot a^*)^* \;. \quad \text{(KA)}$$

The finite non-empty iteration of $a$ is defined as $a^+ =_{df} a \cdot a^* = a^* \cdot a$. Again, the plus in $a^+$ should not be confused with the plus of semiring addition.

Since in a quantale the function defining star is continuous, Kleene's fixpoint theorem shows that $a^* = \bigsqcup_{i \in \mathbb{N}} a^i$. Moreover, we have the star induction rules

$$b + a \cdot x \leq x \;\Rightarrow\; a^* \cdot b \leq x \;, \qquad b + x \cdot a \leq x \;\Rightarrow\; b \cdot a^* \leq x \;. \qquad (2)$$

Our main reason for using quantales rather than an extension of conventional Kleene algebra (see e.g. the discussion on Priscariu's synchronous Kleene algebras [27] in Section 9) is the availability of general fixpoint calculus there. A number of our proofs need the principle of fixpoint fusion which is a second-order

principle; in the first-order setting of conventional Kleene algebra only special cases of it, like the above induction rules can be added as axioms.

We now explain the behaviour of iteration in our program quantales. For a program $P$, the program $P^*$ taken in the quantale $(PR(EV), \cup, \mathsf{false}, ;, \mathsf{skip})$ consists of all sequential compositions of finitely many traces in $P$; it is denoted by $P^\infty$ in [16]. The program $P^*$ taken in $(PR(EV), \cup, \mathsf{false}, *, \mathsf{skip})$ consists of all disjoint unions of finitely many traces in $P$; it may be considered as describing all finite parallel spawnings of traces in $P$. The disjointness requirement that is built into the definition of $*$ and ; does not mean that an iteration cannot repeat primitive actions: the iterated program just needs to supply sufficiently many (e.g., countably many) events having the actions of interest as labels. Then in each round of iteration a fresh one of these can be used.

**Example 3.5** With the notation of Example 2.2 let $P =_{df} P_x \cup P_y \cup P_z$. We first look at its powers w.r.t. $*$ composition:

$$P^2 = P * P = [ax, ay] \cup [ax, az] \cup [ay, az] \ ,$$
$$P^3 = P * P * P = [ax, ay, az] \ .$$

Hence $P^2$ and $P^3$ consist of all programs with exactly two and three events from $\{ax, ay, az\}$, respectively. Since none of the traces in $P$ is disjoint from the one in $P^3$, we have $P^4 = P^3 * P = \emptyset$, and hence strictness of $*$ w.r.t. $\emptyset$ implies $P^n = \emptyset$ for all $n \geq 4$. Therefore $P^*$ consists of all traces with at most three events from $\{ax, ay, az\}$ (the empty trace is there, too, since by definition $\mathsf{skip}$ is contained in every program of the form $Q^*$). It coincides with the set of all possible traces over the three events; this connection will be taken up again in Section 6.

It turns out that for the powers of $P$ w.r.t. the ; operator we obtain exactly the same expressions, since for every program $Q = [p] \cup [q]$ with $p \neq q$ we have

$$Q\,;Q = ([p] \cup [q])\,;([p] \cup [q]) = [p]\,;[p] \cup [p]\,;[q] \cup [q]\,;[p] \cup [q]\,;[q] = [p, q] = Q * Q \ ,$$

provided $p \not\mapsto^+ q$ or $q \not\mapsto^+ p$, i.e., provided the trace $[p, q]$ is consistent with the dependence relation. Only in case of a cyclic dependence $p \to^+ q \to^+ p$ we have $Q\,;Q = \emptyset$, whereas still $Q * Q = [p, q]$. □

If the complete lattice $(S, \leq)$ in a quantale is completely distributive, i.e., if $+$ distributes over arbitrary infima, then $(S, +, 0, \cdot, 1, ^*, ^\omega)$ forms an omega algebra in the sense of [6]. Again this entails many useful laws, e.g.,

$$1^\omega = \top \ , \qquad (a \cdot b)^\omega = a \cdot (b \cdot a)^\omega \ , \qquad (a + b)^\omega = a^\omega + a^* \cdot b \cdot (a + b)^\omega \ .$$

Since $PR(EV)$ is a power set lattice, it is completely distributive. Hence both program quantales also admit infinite iteration with all its laws. The infinite iteration $P^\omega$ w.r.t. the composition operator $*$ is similar to the unbounded parallel spawning $!P$ of traces in $P$ in the $\pi$-calculus (e.g. [30]).

## 4 Concurrent Kleene Algebras

That $PR(EV)$ is a double quantale motivates the following abstract definition.

**Definition 4.1** By a *concurrent Kleene algebra* (CKA) we mean a structure $(S,+,0,*,;,1)$ such that $(S,+,0,*,1)$ and $(S,+,0,;,1)$ are quantales linked by the exchange axiom $(a * b)\,;(c * d) \leq (b\,;c) * (a\,;d)$.

This definition implies, in particular, that $*$ and $;$ are isotone w.r.t. $\leq$ in both arguments. Compared to the original exchange law (1) this one has its free variables in a different order. This does no harm, since the concrete $*$ operator on programs is commutative and hence satisfies the above law as well.

**Corollary 4.2** $(PR(EV), \cup, \mathsf{false}, *, ;, \mathsf{skip})$ *is a CKA.*

The reason for our formulation of the exchange axiom here is that this form of the law implies commutativity of $*$ as well as $a\,;b \leq a * b$ and hence saves two axioms. This is shown by the following result.

**Lemma 4.3** *In a CKA the following laws hold.*

1. $a * b = b * a$.
2. $(a * b)\,;(c * d) \leq (a\,;c) * (b\,;d)$.
3. $a\,;b \leq a * b$.
4. $(a * b)\,;c \leq a * (b\,;c)$.
5. $a\,;(b * c) \leq (a\,;b) * c$.

The notion of a CKA abstracts completely from traces and events; in the companion paper [15] we show how to retrieve these notions algebraically using the lattice-theoretic concept of atoms.

## 5 Hoare Triples

In [16] Hoare triples relating programs are defined by $P\,\{Q\}\,R \Leftrightarrow_{df} P\,;Q \subseteq R$. Again, it is beneficial to abstract from the concrete case of programs.

**Definition 5.1** An *ordered monoid* is a structure $(S,\leq,\cdot,1)$ such that $(S,\cdot,1)$ is a monoid with a partial order $\leq$ and $\cdot$ is isotone in both arguments. In this case we define the *Hoare triple* $a\,\{b\}\,c$ by

$$a\,\{b\}\,c \Leftrightarrow_{df} a \cdot b \leq c\ .$$

**Lemma 5.2** *Assume an ordered monoid* $(S,\leq,\cdot,1)$.

1. $a\,\{1\}\,c \Leftrightarrow a \leq c$; *in particular,* $a\,\{1\}\,a \Leftrightarrow \mathsf{TRUE}$. *(skip)*
2. $(\forall\, a,c : a\,\{b\}\,c \Rightarrow a\,\{b'\}\,c) \Leftrightarrow b' \leq b$. *(antitony)*
3. $(\forall\, a,c : a\,\{b\}\,c \Leftrightarrow a\,\{b'\}\,c) \Leftrightarrow b = b'$. *(extensionality)*
4. $a\,\{b \cdot b'\}\,c \Leftrightarrow \exists\, d : a\,\{b\}\,d \wedge d\,\{b'\}\,c$. *(composition)*
5. $a \leq d \wedge d\,\{b\}\,e \wedge e \leq c \Rightarrow a\,\{b\}\,c$. *(weakening)*

*If* $(S,\cdot,1)$ *is the multiplicative reduct of an idempotent semiring* $(S,+,0,\cdot,1)$ *and the order used in the definition of Hoare triples is the natural semiring order then we have in addition*

6. $a\,\{0\}\,c \Leftrightarrow \mathsf{TRUE}$, *(failure)*

*7.* $a \{b + b'\} c \Leftrightarrow a \{b\} c \wedge a \{b'\} c$. *(choice)*

*If that semiring is a quantale then we have in addition*

*8.* $a \{b\} a \Leftrightarrow a \{b^+\} a \Leftrightarrow a \{b^*\} a$. *(iteration)*

Lemma 5.2 can be expressed more concisely in relational notation. Define for $b \in S$ the relation $\{b\} \subseteq S \times S$ between preconditions $a$ and postconditions $c$ by

$$\forall a, c : a \{b\} c \Leftrightarrow_{df} a \cdot b \leq c \ .$$

Then the above properties rewrite into

1. $\{1\} = \leq$.
2. $\{b\} \subseteq \{b'\} \Leftrightarrow b' \leq b$.
3. $\{b\} = \{b'\} \Leftrightarrow b = b'$.
4. $\{b \cdot b'\} = \{b\} \circ \{b'\}$ where $\circ$ means relational composition.
5. $\leq \circ \{b\} \circ \leq \ \subseteq \{b\}$.
6. $\{0\} = \top\!\top$ where $\top\!\top$ is the universal relation.
7. $\{b + b'\} = \{b\} \cap \{b'\}$.
8. $\{b\} \cap I = \{b^+\} \cap I = \{b^*\} \cap I$ where $I$ is the identity relation.

Properties 4 and 2 allow us to determine the weakest premise ensuring that two composable Hoare triples establish a third one:

**Lemma 5.3** *Assume again an ordered monoid* $(S, \leq, \cdot, 1)$. *Then*

$$(\forall a, d, c : a \{b\} d \wedge d \{b'\} c \Rightarrow a \{e\} c) \Leftrightarrow e \leq b \cdot b' \ .$$

Next we present two further rules that are valid in CKAs when the above monoid operation is specialised to sequential composition:

**Lemma 5.4** *Let* $S = (S, +, 0, *, ;, 1)$ *be a CKA and* $a, a', b, b', c, c', d \in S$ *with* $a \{b\} c$ *interpreted as* $a ; b \leq c$.

*1.* $a \{b\} c \wedge a' \{b'\} c' \Rightarrow (a * a') \{b * b'\} (c * c')$. *(concurrency)*

*2.* $a \{b\} c \Rightarrow (d * a) \{b\} (d * c)$. *(frame rule)*

Let us interpret these results in our concrete CKA of programs. It may seem surprising that so many of the standard basic laws of Hoare logic should be valid for such a weak semantic model of programs. E.g., the definition of weak sequential composition allows all standard optimisations by compilers which shift independent commands between the operands of a semicolon. What is worse, weak composition does not require any data to flow from an assignment command to an immediately following read of the assigned variable. The data may flow to a different thread, which assigns a different value to the variable. In fact, weak sequential composition is required for any model of modern architectures, which allow arbitrary race conditions between fine-grain concurrent threads.

The validity of Hoare logic in this weak model is entirely due to a cheat: that we use the same model for our assertions as for our programs. Thus any weakness of the programming model is immediately reflected in the weakness of the assertion language and its logic. In fact, conventional assertions mention the current values of single-valued program variables; and this is not adequate for reasoning about general fine-grain concurrency. To improve precision here, assertions about the history of assigned values would seem to be required.

## 6 Invariants

We now deal with the set of events a program may use.

**Definition 6.1** A *power invariant* is a program $R$ of the form $R = \mathcal{P}(E)$ for a set $E \subseteq EV$ of events.

It consists of all possible traces that can be formed from events in $E$ and hence is the most general program using only those events. The smallest power invariant is $\mathsf{skip} = \mathcal{P}(\emptyset) = \{\emptyset\}$. The term "invariant" expresses that often a program relies on the assumption that its environment only uses events from a particular subset, i.e., preserves the invariant of staying in that set.

**Example 6.2** Consider again the event set $EV$ form Example 2.2. Let $V$ be a certain subset of the variables involved and let $E$ be the set of all events that assign to variables in $V$. Then the environment $Q$ of a given program $P$ can be constrained to assign at most to the variables in $V$ by requiring $Q \subseteq R$ with the power invariant $R =_{df} \mathcal{P}(E)$. The fact that we want $P$ to be executed only in such environments is expressed by forming the parallel composition $P * R$. □

If $E$ is considered to characterise the events that are admissible in a certain context, a program $P$ can be confined to using only admissible events by requiring $P \subseteq R$ for $R = \mathcal{P}(E)$. In the rely/guarantee calculus of Section 8 invariants will be used to express properties of the environment on which a program wants to rely (whence the name $R$).

Power invariants satisfy a rich number of useful laws (see [15] for details). The most essential ones for the purposes of the present paper are the following straightforward ones for arbitrary invariant $R$:

$$\mathsf{skip} \subseteq R \;, \qquad R * R \subseteq R \;. \tag{3}$$

We now again abstract from the concrete case of programs. It turns out that the properties in (3) largely suffice for characterising invariants.

**Definition 6.3** An *invariant* in a CKA $S$ is an element $r \in S$ satisfying $1 \leq r$ and $r * r \leq r$. These two axioms can equivalently be combined into $1 + r * r \leq r$. The set of all invariants of $S$ is denoted by $I(S)$.

We now first give a number of algebraic properties of invariants that are useful in proving the soundness of the rely/guarantee-calculus in Section 8.

**Theorem 6.4** *Assume a CKA $S$, an $r \in I(S)$ and arbitrary $a, b \in S$.*

1. $a \leq r \circ a$ *and* $a \leq a \circ r$.
2. $r \,;\, r \leq r$.
3. $r * r = r = r \,;\, r$.
4. $r \,;\, (a * b) \leq (r \,;\, a) * (r \,;\, b)$ *and* $(a * b) \,;\, r \leq (a \,;\, r) * (b \,;\, r)$.
5. $r \,;\, a \,;\, r \leq r * a$.
6. $a \in I(S) \Leftrightarrow a = a^*$, *where* $^*$ *is taken w.r.t.* $*$ *composition.*

7. *The least invariant comprising $a$ is $a^*$ where $^*$ is taken w.r.t. $*$ composition.*

Next we discuss the lattice structure of the set $I(S)$ of invariants.

**Theorem 6.5** *Assume again a CKA $S$.*
1. *$(I(S), \leq)$ is a complete lattice with least element $1$ and greatest element $\top$.*
2. *For $r, r' \in I(S)$ we have $r \leq r' \Leftrightarrow r * r' = r'$. This means that $\leq$ coincides with the natural order induced by the associative, commutative and idempotent operation $*$ on $I(S)$.*
3. *For $r, r' \in I(S)$ the infimum $r \sqcap r'$ in $S$ coincides with the infimum of $r$ and $r'$ in $I(S)$.*
4. *$r * r'$ is the supremum of $r$ and $r'$ in $I(S)$. In particular, $r \leq r'' \wedge r' \leq r'' \Leftrightarrow r * r' \leq r''$ and $r' \sqcap (r * r') = r'$.*
5. *Invariants are downward closed: $r * r' \leq r'' \Rightarrow r \leq r''$.*
6. *$I(S)$ is even closed under arbitrary infima, i.e., for a subset $U \subseteq I(S)$ the infimum $\sqcap U$ taken in $S$ coincides with the infimum of $U$ in $I(S)$.*

We conclude this section with two laws about iteration.

**Lemma 6.6** *Assume a CKA $S$ and let $r \in I(S)$ be an invariant and $a \in S$ be arbitrary. Let the finite iteration $^*$ be taken w.r.t. $*$ composition. Then*
1. *$(r * a)^* \leq r * a^*$.*
2. *$r * a^* = r * (r * a)^*$.*

## 7 Single-Event Programs and Rely/Guarantee-CKAs

We will now show that our definitions of $*$ and ; for concrete programs in terms of transitive closure of the dependence relation $\rightarrow$ entail two important further laws that are essential for the rely/guarantee calculus to be defined below. In the following theorem they are presented as inclusions; the reverse inclusions already follow from Theorem 6.4.4 for Part 1 and from Lemma 4.3.5, 4.3.4, 4.3.1 and Theorem 6.4.3 for Part 2. Informally, Part 1 means that for acyclic $\rightarrow$ parallel composition of an invariant with a singleton program can be always sequentialised. Part 2 means that for invariants a kind of converse to the exchange law of Lemma 4.3.2 holds.

**Theorem 7.1** *Let $R = \mathcal{P}(E)$ be a power invariant in $PR(EV)$.*
1. *If $\rightarrow$ is acyclic and $p \in EV$ then $R * [p] \subseteq R\,;[p]\,;R$.*
2. *For all $P, Q \in PR(EV)$ we have $R * (P\,;Q) \subseteq (R * P)\,;(R * Q)$.*

In the companion paper [15] we define the composition operators ; and $\|$ in terms of $\rightarrow$ rather than $\rightarrow^+$ and show a converse of Theorem 7.1:
- If Part 1 is valid then $\rightarrow$ is weakly acyclic, viz.

$$\forall\, p, q \in EV : p \rightarrow^+ q \rightarrow^+ p \;\Rightarrow\; p = q\ .$$

This means that $\rightarrow$ allows at most immediate self-loops which cannot be "detected" by our definitions of the operators that require disjointness of the operands. It is easy to see that $\rightarrow$ is weakly acyclic iff its reflexive-transitive closure $\rightarrow^*$ is a partial order.

– If Part 2 is valid then $\rightarrow$ is weakly transitive, i.e.,

$$p \rightarrow q \rightarrow r \;\;\Rightarrow\;\; p = r \vee p \rightarrow r \;.$$

This provides the formal justification why in the present paper we right away defined our composition operators in terms of $\rightarrow^+$ rather than just $\rightarrow$.

As before we abstract the above results into general algebraic terms. The terminology stems from the applications in the next section.

**Definition 7.2** A *rely/guarantee-CKA* is a pair $(S, I)$ such that $S$ is a CKA and $I \subseteq I(S)$ is a set of invariants such that $1 \in I$ and for all $r, r' \in I$ also $r \sqcap r' \in I$ and $r * r' \in I$, in other words, $I$ is a sublattice of $I(S)$. Moreover, all $r \in I$ and $a, b \in S$ have to satisfy $r * (a\,;b) \leq (r * a)\,;(r * b)$.

Together with the exchange law in Lemma 4.3.2, $\circ$-idempotence of $r$ and commutativity of $*$ this implies

$$r * (b \circ c) = (r * b) \circ (r * c) \qquad (*\textit{-distributivity})$$

for all invariants $r \in I$ and operators $\circ \in \{*, ;\}$.

The restriction that $I$ be a sublattice of $I(S)$ is motivated by the rely/guarantee-calculus in Section 8 below.

Using Theorem 7.1 we can prove

**Lemma 7.3** *Let $I =_{df} \{\mathcal{P}(E) \mid E \subseteq EV\}$ be the set of all power invariants over $EV$. Then $(PR(EV), I)$ is a rely-guarantee-CKA.*

*Proof.* We only need to establish closure of $\mathcal{P}(\mathcal{P}(EV))$ under $*$ and $\cap$. But straightforward calculations show that $\mathcal{P}(E) * \mathcal{P}(F) = \mathcal{P}(E \cup F)$ and $\mathcal{P}(E) \cap \mathcal{P}(F) = \mathcal{P}(E \cap F)$ for $E, F \subseteq EV$. □

We now can explain why it was necessary to introduce the subset $I$ of invariants in a rely/guarantee-CKA. Our proof of $*$-distributivity used downward closure of power invariants. Other invariants in $PR(EV)$ need not be downward closed and hence $*$-distributivity need not hold for them.

**Example 7.4** Assume an event set $EV$ with three different events $p, q, r \in EV$ and dependences $p \rightarrow r \rightarrow q$. Set $P =_{df} [p, q]$. Then $P * P = \emptyset$ and hence $P^i = \emptyset$ for all $i > 1$. This means that the invariant $R =_{df} P^* = \mathsf{skip} \cup P = [] \cup [p, q]$ is not downward closed. Indeed, $*$-distributivity does not hold for it: we have $R * [r] = [r] \cup [p, q, r]$, but $R\,;[r]\,;R = [r]$. □

The property of $*$-distributivity implies further iteration laws.

**Lemma 7.5** *Assume a rely/guarantee-CKA $(S, I)$, an invariant $r \in I$ and an arbitrary $a \in S$ and let the finite iteration $^*$ be taken w.r.t. $\circ \in \{*, ;\}$.*

1. $r * a^* = (r * a)^* \circ r = r \circ (r * a)^*$.
2. $(r * a)^+ = r * a^+$.

## 8 Jones's Rely/Guarantee-Calculus

In [17] Jones has presented a calculus that considers properties of the environment on which a program wants to rely and the ones it, in turn, guarantees for the environment. We now provide an abstract algebraic treatment of this calculus.

**Definition 8.1** We define, abstracting from [16], the *guarantee relation* by setting for arbitrary element $b$ and invariant $g$

$$b \;\mathsf{guar}\; g \;\Leftrightarrow_{df}\; b \leq g \;.$$

A slightly more liberal formulation is discussed in [15].

**Example 8.2** With the notation $P_u =_{df} [au]$ for $u \in \{x, y, z\}$ of Example 2.2 we have $P_u \;\mathsf{guar}\; G_u$ where $G_u =_{df} P_u \cup \mathsf{skip} = [au] \cup []$. □

We have the following properties.

**Theorem 8.3** *Let $a, b, b'$ be arbitrary elements and $g, g'$ be invariants of a CKA. Let $\circ \in \{*, ;\}$ and $^{*\circ}$ be the associated iteration operator.*

1. $1 \;\mathsf{guar}\; g$.
2. $b \;\mathsf{guar}\; g \;\wedge\; b' \;\mathsf{guar}\; g' \;\Rightarrow\; (b \circ b') \;\mathsf{guar}\; (g * g')$.
3. $a \;\mathsf{guar}\; g \;\Rightarrow\; a^{*\circ} \;\mathsf{guar}\; g$.
4. *For the concrete case of programs let $G = \mathcal{P}(E)$ for some set $E \subseteq EV$ and $p \in EV$. Then $[p] \;\mathsf{guar}\; G \;\Leftrightarrow\; p \in E$.*

Using the guarantee relation, Jones quintuples can be defined, as in [16], by

$$a \;r\; \{b\}\; g \;s \;\;\Leftrightarrow_{df}\;\; a \;\{r * b\}\; s \;\;\wedge\;\; b \;\mathsf{guar}\; g \;,$$

where $r$ and $g$ are invariants and Hoare triples are again interpreted in terms of sequential composition $;$.

The first rule of the rely/guarantee calculus concerns parallel composition.

**Theorem 8.4** *Consider a CKA $S$. For invariants $r, r', g, g' \in I(S)$ and arbitrary $a, a', b, b', c, c' \in S$,*

$$\begin{array}{l} a \;r\; \{b\}\; g \;c \;\;\wedge\;\; a' \;r'\; \{b'\}\; g' \;c' \;\;\wedge\;\; g' \;\mathsf{guar}\; r \;\;\wedge\; g \;\mathsf{guar}\; r' \;\;\Rightarrow \\ \qquad (a \sqcap a')\; (r \sqcap r')\; \{b * b'\}\; (g * g')\; (c \sqcap c') \;. \end{array}$$

Note that $r \sqcap r'$ and $g * g'$ are again invariants by Lemma 6.5.3 and 6.5.4.

For sequential composition one has

**Theorem 8.5** *Assume a rely/guarantee-CKA $(S, I)$. Then for invariants $r, r'$, $g, g' \in I$ and arbitrary $a, b, b', c, c'$,*

$$a \;r\; \{b\}\; g \;c \;\wedge\; c \;r'\; \{b'\}\; g' \;c' \;\Rightarrow\; a \;(r \sqcap r')\; \{b \,;\, b'\}\; (g * g')\; c'$$

Next we give rules for 1, union and singleton event programs.

**Theorem 8.6** *Assume a rely/guarantee-CKA $(S, I)$. Then for invariants $r, g \in I$ and arbitrary $s \in S$,*

1. $a\ r\ \{1\}\ g\ s \ \Leftrightarrow\ a\ \{r\}\ s$.
2. $a\ r\ \{b + b'\}\ g\ s \ \Leftrightarrow\ a\ r\ \{b\}\ g\ s \wedge a\ r\ \{b'\}\ g\ s$.
3. *Assume power invariants $R = \mathcal{P}(E)$, $G = \mathcal{P}(F)$ for $E, F \subseteq EV$, event $p \notin E$ and let $\rightarrow$ be acyclic. Then $P\ R\ \{[p]\}\ G\ S \Leftrightarrow P\ \{R\,;[p]\,;R\}\ S \wedge [p]\ \mathsf{guar}\ G$.*

Finally we give rely/guarantee rules for iteration.

**Theorem 8.7** *Assume a rely/guarantee-CKA $(S, I)$ and let $^*$ be finite iteration w.r.t. $\circ \in \{*, ;\}$. Then for invariants $r, g \in I$ and arbitrary elements $a, b \in S$,*

$$a\ r\ \{b\}\ g\ a \ \Rightarrow\ a\ r\ \{b^+\}\ g\ a\ ,$$
$$a\ \{r\}\ a \wedge a\ r\ \{b\}\ g\ a \ \Rightarrow\ a\ r\ \{b^*\}\ g\ a\ .$$

We conclude this section with a small example of the use of our rules.

**Example 8.8** We consider again the programs $P_u = [au]$ and invariants $G_u = P_u \cup \mathsf{skip}$ $(u \in \{x, y\})$ from Example 8.2. Moreover, we assume an event $av$ with $v \neq x, y$, $ax \not\rightarrow av$ and $ay \not\rightarrow av$ and set $P_v =_{df} [av]$. We will show

$$P_v\ \mathsf{skip}\ \{P_x * P_y\}\ (G_x * G_y)\ [av, ax, ay]$$

holds. In particular, the parallel execution of the assignments $x := x + 1$ and $y := y + 2$ guarantees that at most $x$ and $y$ are changed. We set $R_x =_{df} G_y$ and $R_y =_{df} G_x$. Then

$$\text{(a)}\ P_x\ \mathsf{guar}\ G_x\ \mathsf{guar}\ R_y\ , \qquad \text{(b)}\ P_y\ \mathsf{guar}\ G_y\ \mathsf{guar}\ R_x\ .$$

Define the postconditions

$$S_x =_{df} [av, ax] \cup [av, ax, ay] \quad \text{and} \quad S_y =_{df} [av, ay] \cup [av, ax, ay]\ .$$

Then

$$\text{(c)}\ S_x \cap S_y = [av, ax, ay]\ , \qquad \text{(d)}\ R_x \cap R_y = \mathsf{skip}\ .$$

From the definition of Hoare triples we calculate

$$P_v\ \{R_x\}\ ([av] \cup [av, ay]) \qquad ([av] \cup [av, ay])\ \{P_x\}\ S_x \qquad S_x\ \{R_x\}\ S_x\ ,$$

since $[av, ax, ay] * [ay] = \emptyset$. Combining the three clauses by Lemma 5.2.4 we obtain

$$P_v\ \{R_x\,;P_x\,;R_x\}\ S_x\ .$$

By Theorem 8.6.3 we obtain $P_v\ R_y\ \{P_x\}\ G_x\ S_x$ and, similarly, $P_v\ R_x\ \{P_y\}\ G_y\ S_y$. Now the claim follows from the clauses (a),(b),(c),(d) and Theorem 8.4. □

In a practical application of the theory of Kleene algebras to program correctness, the model of a program trace will be much richer than ours. It will certainly include labels on each event, indicating which atomic command of the program is responsible for execution of the event. It will include labels on each data flow arrow, indicating the value which is 'passed along' the arrow, and the identity of the variable or communication channel which mediated the flow.

## 9 Related Work

Although our basic model and its algebraic abstraction reflect a non-interleaving view of concurrency, we try to set up a connection with familiar process algebras such as ACP [1], CCS[21], CSP[13], mCRL2 [11] and the $\pi$-calculus [30].

It is not easy to relate their operators to those of CKA. The closest analogies seem to be the following ones.

| CKA operator | corresponding operator |
|---|---|
| $+$ | non-deterministic choice in CSP |
| $*$ | parallel composition $\mid$ in ACP, $\pi$-calculus and CCS |
| $\parallel$ | interleaving $\mid\mid\mid$ in CSP |
| $;$ | sequential composition ; in CSP and $\cdot$ in ACP |
| $[]$ | choice $+$ in CCS and internal choice $\Box$ in CSP |
| $1$ | `SKIP` in CSP |
| $0$ | this is the miracle and cannot be represented in any implementable calculus |

However, there are a number of laws which show the inaccuracy of this table. For instance, in CSP we have $\texttt{SKIP} \Box P \neq P$, whereas CKA satisfies $1 [] P = P$. A similarly different behaviour arises in CCS, ACP and the $\pi$-calculus concerning distributivity of composition over choice.

As the discussion after Theorem 7.1 shows, our basic model falls into the class of partial-order models for true concurrency. Of the numerous works in that area we discuss some approaches that have explicit operators for composition related to our $*$ and $;$. Whereas we assume that our dependence relation is fixed a priori, in the pomset approach [10, 9, 26] is is constructed by the composition operators. The operators there are sequential and concurrent composition; there are no choice and iteration, though. Moreover, no laws are given for the operators. In Winskel's event structures [31] there are choice (sum) and concurrent composition, but no sequential composition and iteration. Again, there are no interrelating laws. Another difference to our approach is that the "traces" are required to observe certain closure conditions.

Among the axiomatic approaches to partial order semantics we mention the following ones. Boudol and Castellani [4] present the notion of trioids, which are algebras offering the operations of choice, sequential and concurrent composition. However, there are no interrelating laws and no iteration. Chothia and Kleijn07 [5] use a double semiring with choice, sequential and concurrent composition, but again no interrelating laws and no iteration. The application is to model quality of service, not program semantics.

The approach closest in spirit to ours are Prisacariu's synchronous Kleene algebras (SKA) [27]. The main differences are the following. SKAs are not quantale-based, but rather an enrichment of conventional Kleene algebras. They are restricted to a finite alphabet of actions and hence have a complete and even decidable equational theory. There is only a restricted form of concurrent composition, and the exchange law is equational rather than equational. Iteration is present but not used in an essential way. Nevertheless, Prisacariu's paper is

the only of the mentioned ones that explicitly deals with Hoare logic. It does so using the approach of Kleene algebras with tests [19]. This is not feasible in our basic model, since tests are required to be below the element 1, and 0 and 1 are the only such elements. Note, however, that Mace4 [28] quickly shows that this is not a consequence of the CKA axioms but holds only for the particular model.

## 10 Conclusion and Outlook

The study in this paper has shown that even with the extremely weak assumptions of our trace model many of the important programming laws can be shown, mostly by very concise and simple algebraic calculations. Indeed, the rôle of the axiomatisation was precisely to facilitate these calculations: rather than verifying the laws laboriously in the concrete trace model, we can do so much more easily in the algebraic setting of Concurrent Kleene Algebras. This way many new properties of the trace model have been shown in the present paper. Hence, although currently we know of no other interesting model of CKA than the trace model, the introduction of that structure has already been very useful.

The discussion in the previous section indicates that CKA is not a direct abstraction of the familiar concurrency calculi. Rather, we envisage that the trace model and its abstraction CKA can serve as a basic setting into which many of the existing other calculi can be mapped so that then their essential laws can be proved using the CKA laws. A first experiment along these lines is a trace model of a core subset of the $\pi$-calculus in [16]. An elaboration of these ideas will be the subject of further studies.

**Acknowledgement** We are grateful for valuable comments by J. Desharnais, H.-H. Dang, R. Glück, W. Guttmann, P. Höfner, P. O'Hearn, H. Yang and by the anonymous referees of CONCUR09.

## References

1. J.A. Bergstra, I. Bethke, A. Ponse: Process algebra with iteration and nesting. The Computer Journal 37(4), 243–258 (1994)
2. Birkhoff, G. Lattice Theory, 3rd ed. Amer. Math. Soc. 1967
3. S. Bistarelli, U. Montanari, F. Rossi: Semiring-based constraint satisfaction and optimization. J. ACM, 44(2):201-236 (1997)
4. G. Boudol, I. Castellani: On the semantics of concurrency: partial orders and transition systems. In: Ehrig, H., Levi, G., Montanari, U. (eds.) CAAP 1987 and TAPSOFT 1987. LNCS 249. Springer 1987, 123–137
5. T. Chothia, J. Kleijn: Q-Automata: modelling the resource usage of concurrent components. Electr. Notes Theor. Comput. Sci. 175(2): 153–167 (2007)
6. E. Cohen: Separation and reduction. In: R. Backhouse, J. Oliveira (eds.): Mathematics of Program Construction (MPC'00). LNCS 1837. Springer 2000, 45–59
7. J. Conway: Regular Algebra and Finite Machines. Chapman&Hall 1971
8. J. Desharnais, B. Möller, G. Struth: Kleene Algebra with domain. Trans. Computational Logic 7, 798–833 (2006)

9. J. Gischer: Partial orders and the axiomatic theory of shuffle. PhD thesis, Stanford University (1984)
10. Grabowski, J.: On partial languages. Fundamenta Informaticae 4(1), 427–498 (1981)
11. J. Groote, A. Mathijssen, M. van Weerdenburg, Y. Usenko. From $\mu$CRL to mCRL2: motivation and outline. In: Proc. Workshop Essays on Algebraic Process Calculi (APC 25). ENTCS 162, 191–196 (2006)
12. C.A.R. Hoare: An axiomatic basis for computer programming. Commun. ACM. 12, 576–585 (1969)
13. C.A.R. Hoare: Communicating sequential processes. Prentice Hall 1985
14. C.A.R. Hoare, B. Möller, G. Struth, I. Wehrman: Concurrent Kleene Algebra. Institut für Informatik, Universität Augsburg, Technical Report 2009-04, April 2009
15. C.A.R. Hoare, B. Möller, G. Struth, I. Wehrman: Foundations of Concurrent Kleene Algebra. Institut für Informatik, Universität Augsburg, Technical Report 2009-05, April 2009
16. C.A.R. Hoare, I. Wehrman, P. O'Hearn: Graphical models of separation logic. Proc. Marktoberdorf Summer School 2008 (forthcoming)
17. C. Jones: Development methods for computer programs including a notion of interference. PhD Thesis, University of Oxford. Programming Research Group, Technical Monograph 25, 1981
18. D. Kozen: A completeness theorem for Kleene algebras and the algebra of regular events. Information and Computation 110, 366–390 (1994)
19. D. Kozen: Kleene algebra with tests. Trans. Programming Languages and Systems 19, 427–443 (1997)
20. S. Mac Lane: Categories for the working mathematician (2nd ed.). Springer 1998
21. R. Milner: A Calculus of Communicating Systems. LNCS 92. Springer 1980
22. J. Misra: Axioms for memory access in asynchronous hardware systems. ACM Trans. Program. Lang. Syst. 8, 142–153 (1986)
23. C. Morgan: Programming from Specifications. Prentice Hall 1990
24. C. Mulvey: &. Rendiconti del Circolo Matematico di Palermo 12, 99–104 (1986)
25. P. O'Hearn: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375, 271–307 (2007)
26. Pratt, V.R.: Modelling concurrency with partial orders. Journal of Parallel Programming 15(1) (1986)
27. C. Prisacariu: Extending Kleene lgebra with synchrony — technicalities. University of Oslo, Department of Informatics, Research Report No. 376, October 2008
28. W. McCune: Prover9 and Mace4. `http://www.prover9.org/` (accessed March 1, 2009)
29. K. Rosenthal: Quantales and their applications. Pitman Research Notes in Math. No. 234 Longman Scientific and Technical 1990
30. D. Sangiorgi, D. Walker: The $\pi$-calculus — A theory of mobile processes. Cambridge University Press 2001
31. G. Winskel: Event structures. In: W. Brauer, W. Reisig, G. Rozenberg (eds.) Advances in Petri Nets 1986. LNCS 255. Springer 1987, 325–392