Concurrent Kleene Algebra and its Foundations

Tony Hoare
Microsoft Research, Cambridge, UK

B. Möller
Universität Augsburg, Germany

Georg Struth
University of Sheffield, UK

Ian Wehrman
University of Texas at Austin, USA

Abstract

A Concurrent Kleene Algebra offers two composition operators, related by a weak version of an exchange law: when applied in a trace model of program semantics, one of them stands for sequential execution and the other for concurrent execution of program components [22]. After introducing this motivating concrete application, we investigate its abstract background in terms of a primitive independence relation between the traces. On this basis, we develop a series of richer algebras; the richest validates a proof calculus for programs similar to that of a Jones style rely/guarantee calculus. On the basis of this abstract algebra, we finally reconstruct the original trace model, using the notion of atoms from lattice theory.

Keywords: concurrency, dependence, algebra, Hoare calculus, rely/guarantee calculus

1. Introduction

Kleene algebra [10] has been recognised and developed [26, 27, 11] 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 binary relations, with operators for union, relational composition and reflexive transitive closure, as well as formal languages, with operators for union, relational composition and reflexive transitive closure, as well as formal languages, with operators for union, concatenation and Kleene star.

This paper defines a ‘double’ Kleene algebra, which adds an operator for concurrent composition. In fact, we summarise a whole family of algebras with these two operators under the common heading of concurrent Kleene algebra (CKA). Sequential composition ; and concurrent composition ∗ are related by the law \((a ∗ b) ∗ (c ∗ d) ≤ (a ; c) ∗ (b ; d)\), an inequational weakening of the corresponding equational exchange law of two-category or bicategory theory (cf. [29]). For special elements \(r\) that a.o. satisfy \(r ∗ r = r\) this weak form can be strengthened to the equational law \(r ∗ (a ; b) = (r ∗ a) ; (r ∗ b)\), by which concurrent composition distributes through sequential.

The interest of CKAs is two-fold. First, they express in their most general form the essential properties of program execution; indeed, they represent just those properties which are preserved even by concurrent...
architectures with weakly ordered memory access, unreliable communications and massively re-ordering
program optimisers. Second, 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 [10]
and in concurrent style [25].

The purpose of the paper is to introduce the basic operators and their laws, and study them both in
their concrete representation and in their 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.

In our concrete model of program semantics, 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 sub-programs 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 [35].

The model includes the notion of a primitive dependence relation between the events of a trace. The
transitive closure of primitive dependence represents a direct or indirect chain of dependence, which imposes
time constraints on the ordering of the occurrence of events. 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 liberal form of
sequential composition. Concurrent composition places no such restriction, and allows dependence in either
direction. The above-mentioned exchange law describes a kind of mutual distribution which captures the
interrelation between sequential and concurrent composition in an equational 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, only weak properties of a program 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 (cf. [32]).
Fortunately, as long as they are consistent with our fundamental theory, variations in these additional actions
do not invalidate our development and hence do not require fresh proofs of any of our theorems.

We demonstrate that our program algebra, even in its abstract form, admits a very concise validation
of familiar proof rules for sequential programs (Hoare triples) and for concurrent programming (Jones’s
rely/guarantee calculus).

On the foundational side, we show that the above-mentioned distribution law and a related one are equiv-
alent to transitivity and acyclicity of the dependence relation, respectively; the traces obeying a generalised
version of the second law are characterised in terms of convexity w.r.t dependence.

Finally, we introduce the notion of an event-based concurrent Kleene algebra which reconstructs the
concrete trace model in terms of the more abstract order-theoretic notions of atoms and irreducible elements.
We show that in such algebras the dependence relation can be recovered from the operators of sequential
and concurrent composition.

Most of our reasoning has been checked by computer using the automated theorem proving system
Prover9/Mace4 [30].

The paper is organised as follows. Section 2 summarises the definitions of the trace model and its
essential operators. In Section 3 we develop an abstract calculus of independence relations, which then is
algebraised in Section 4. After that, Section 5 presents idempotent semirings and quantales as fundamental
algebraic structures. In Section 6 we give axiomatisations of various concurrent structures that offer two
operators for concurrent and sequential composition, related by the above-mentioned inequational exchange
law. In Section 7 we give a more abstract view of the composition operators used in the concrete trace
model. Section 8 enriches the setting by operators for finite and infinite iteration, which leads to concurrent
Kleene and omega algebras. In Section 9 we present an algebraic view of Hoare triples, which serve as basic ingredients of the rely/guarantee calculus of later sections. As a preparation for that, Section 10 gives a definition of invariants. In Section 11 we establish the mentioned equivalence of two fundamental laws with (weak) acyclicity and transitivity of the basic dependence relation. The results are used in Section 12 to define a further class of algebras that are tailored to the needs of the rely/guarantee calculus presented in Section 13 and, in a simplified form, in Section 14. Finally, Section 15 and Section 16 develop the notion of event-based concurrent algebras and reconstruct the trace model and the dependence relation in terms of that notion. Section 17 presents related work, while Section 18 contains conclusion and outlook. Appendix A summarises the laws characterising the most important algebraic structures involved. Appendix B shows a sample input file for the automated theorem prover Prover9.

2. Operators on Traces and Programs

In this section we present a concrete model of Concurrent Kleene Algebra which serves as a motivation for the abstract algebraic treatment in later sections.

We assume a set \( EV \) of events, which are occurrences of primitive actions, together with a dependence relation \( \rightarrow \subseteq EV \times EV \) between them: \( e \rightarrow f \) indicates a flow of data or control from event \( e \) to event \( f \).

**Definition 2.1.** A trace is a set of events; the set of all traces over \( EV \) is denoted by \( TR(EV) = \mathcal{P}(EV) \). A program is a set of traces; the set of all programs is denoted by \( PR(EV) = \mathcal{P}(TR(EV)) \).

We keep the definition of traces and programs so liberal to accommodate systems with very loose coupling of events; e.g., “conventional” linear traces can be obtained by including unique time stamps into the events and defining the dependence relation such that it respects time.

Examples of very simple programs are the following. The program \( \text{skip} \), which does nothing, is defined as \( \{\emptyset\} \), and the program \( \{e\} \), which does only \( e \in EV \), is \( \{\{e\}\} \). The program \( \text{false} = df \emptyset \) has no traces, and therefore cannot be executed at all. In the context program development by stepwise refinement, it serves the rôle of the ‘miracle’ [33].

Following [20] we study four operators on programs \( P \) and \( Q \):

- \( P \ast Q \) : 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 – only one of \( P \) or \( Q \) is executed, if at all.

Details will be given below.

To express the restrictions in this list we introduce the following notion.

**Definition 2.2.** We call a trace \( tp \) independent of a trace \( tq \), written \( tp \nleftrightarrow tq \), if there are no dependence arrows from events of \( tq \) to events of \( tp \):

\[
 tp \nleftrightarrow tq \iff \neg \exists e \in tp, f \in tq : f \rightarrow e .
\]

The intention is that all events of \( tp \) should occur before all those in \( tq \), and hence \( tp \) may in no way depend on its “future” \( tq \).

Now, for each operator \( \circ \in \{\ast, ;, \parallel, |\} \) we define an associated binary relation \( (\circ) \) between traces such that for programs \( P, Q \) we can generically set

\[
P \circ Q = df \{ tp \cup tq \mid tp \in P \land tq \in Q \land tp (\circ) tq \} .
\]

From this definition it is immediate that \( \circ \) distributes through arbitrary unions of families of programs and hence is \( \subseteq \)-isotone and \( \text{false} \)-strict, i.e., \( \text{false} \circ P = \text{false} = P \circ \text{false} \). Moreover, if \( (\circ) \) is symmetric then \( \circ \) is commutative.
are commutative and are symmetric. Moreover, the exchange law allows forming parallel programs with race conditions, whereas \( ; \) and \( \parallel \) do not. Example 2.3. We illustrate the operators with a small example. Assume a set \( \{a, b, c\} \) of events the actions of which are simple assignments to program variables. We consider three particular events that are simple assignments to program variables. We form the corresponding single-event programs \( P_x = \{x := 1, y := z := 2\} \), \( P_y = \{x := 2, y := z := 1\} \), \( P_z = \{x := 3, y := z := 2\} \). To describe their compositions we extend the notation for single-event programs and set \( \{e_1, \ldots, e_n\} =_{df} \{e_1, \ldots, e_n\} \) (for uniformity we sometimes also write \( [\ ] \) for \( \text{skip} \)). 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. □

The above operations on traces are lifted pointwise to programs by setting \( P \circ Q =_{df} \{p \circ q | p \in P \land q \in Q\} \). It is straightforward from the definitions that the liftings of \( \cdot \), \( \parallel \) and \( [\ ] \) are commutative and that \( [\ ] \subseteq [\ ] \subseteq \cdot \subseteq * \), where for operators \( \cdot, \circ \in \{*, \parallel, [\ ]\} \) the formula \( \cdot \subseteq \circ \) abbreviates \( \forall P, Q : P \cdot Q \subseteq P \circ Q \).

We can now also explain informally why the exchange law

\[
(P * R) ; (Q * S) \subseteq (P ; Q) * (R ; S)
\]

mentioned in the introduction is valid. In both programs, dependence arrows from \( Q \) to \( P \) and from \( S \) to \( R \) need to be excluded. However, in the program on the left-hand side all events from the \( P * R \) part have to occur before all those of the \( Q * S \) part, so that, in addition, dependence arrows from \( Q \) to \( R \) and from \( S \) to \( Q \) need to be excluded. Therefore the left program may have fewer traces than the right one. This is illustrated by the diagram

\[
P \xrightarrow{R} Q \quad \subseteq \quad P \xrightarrow{R} Q
\]

In the next section we will develop a simple calculus that allows a formal verification of this and related laws on a general basis.

<table>
<thead>
<tr>
<th>(*)</th>
<th>(P_x)</th>
<th>(P_y)</th>
<th>(P_z)</th>
<th>(;)</th>
<th>(P_x)</th>
<th>(P_y)</th>
<th>(P_z)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P_x)</td>
<td>(\emptyset)</td>
<td>([ax, ay])</td>
<td>([ax, az])</td>
<td>(P_x)</td>
<td>(\emptyset)</td>
<td>([ax, ay])</td>
<td>([ax, az])</td>
</tr>
<tr>
<td>(P_y)</td>
<td>([ax, ay])</td>
<td>(\emptyset)</td>
<td>([ay, az])</td>
<td>(P_y)</td>
<td>([ax, ay])</td>
<td>(\emptyset)</td>
<td>([ay, az])</td>
</tr>
<tr>
<td>(P_z)</td>
<td>([ax, az])</td>
<td>(\emptyset)</td>
<td>([ay, az])</td>
<td>(P_z)</td>
<td>(\emptyset)</td>
<td>([ay, az])</td>
<td>(\emptyset)</td>
</tr>
</tbody>
</table>

\[
\begin{array}{ccc}
\parallel & | & \parallel \\
\hline
\emptyset & | & \emptyset \\
\emptyset & | & \emptyset \\
| & | & |
\end{array}
\]

Figure 1: Composition tables

Now the above informal descriptions are captured by the definitions

\[
\begin{align*}
\text{tp}(*) & \iff \text{tp} \cap \text{tp} = \emptyset, \\
\text{tp}(;) & \iff \text{tp} (\text{tp} \cap \text{tp}) \cap \text{tp} = \emptyset, \\
\text{tp}([\ ] & \iff \text{tp} (\text{tp} \cap \text{tp}) \cap \text{tp} \cap \text{tp} = \emptyset, \\
\text{tp}([\ ] & \iff \text{tp} (\text{tp} \cap \text{tp}) \cap \text{tp} \cap \text{tp} = \emptyset.
\end{align*}
\]

It is clear that \( ([\ ] \subseteq [\ ] \subseteq (\cdot) \subseteq (*\cdot) \) and that \((*, [\ ]), ([\ ]), (\cdot)\) are symmetric. Moreover, \(\text{skip} \circ P = P = P \circ \text{skip} \).

Example 2.3. We illustrate the operators with a small example. 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 \( x := x + 1, y := y + 2, z := x + 3 \), respectively. There is a dependence arrow from event \( e \) to event \( f \) iff \( e \neq f \) and the variable assigned to in \( e \) occurs in the assigned expression at the right-hand side of \( f \). 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 \( [e_1, \ldots, e_n] =_{df} \{e_1, \ldots, e_n\} \) (for uniformity we sometimes also write \( [\ ] \) for \( \text{skip} \)). 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. □

The above operations on traces are lifted pointwise to programs by setting \( P \circ Q =_{df} \{p \circ q | p \in P \land q \in Q\} \). It is straightforward from the definitions that the liftings of \( \cdot \), \( \parallel \) and \( [\ ] \) are commutative and that \( [\ ] \subseteq \cdot \subseteq \parallel \subseteq * \), where for operators \( \cdot, \circ \in \{*, \parallel, [\ ]\} \) the formula \( \cdot \subseteq \circ \) abbreviates \( \forall P, Q : P \cdot Q \subseteq P \circ Q \). We can now also explain informally why the exchange law

\[
(P * R) ; (Q * S) \subseteq (P ; Q) * (R ; S)
\]

mentioned in the introduction is valid. In both programs, dependence arrows from \( Q \) to \( P \) and from \( S \) to \( R \) need to be excluded. However, in the program on the left-hand side all events from the \( P * R \) part have to occur before all those of the \( Q * S \) part, so that, in addition, dependence arrows from \( Q \) to \( R \) and from \( S \) to \( Q \) need to be excluded. Therefore the left program may have fewer traces than the right one. This is illustrated by the diagram

\[
P \xrightarrow{R} Q \quad \subseteq \quad P \xrightarrow{R} Q
\]
In the remainder of this paper we shall mostly concentrate on the more interesting operators $\ast$ and ;.

Another essential operator is union which again is $\subseteq$-isotone and distributes through arbitrary unions. However, it is not 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 false. Use of union in such a recursion enables non-trivial fixpoints, as will be seen in Section 8.

3. Aggregation and Independence

To derive interesting laws about our operators in a general and concise way, we take a more abstract view of systems, such as programs, their parts and their interactions. The main concepts we study are aggregation—how systems are built from their parts—and (in)dependence—how systems and their parts interact.

Definition 3.1. An aggregation algebra is a structure $(A, +)$ formed by a set $A$ and a binary operator $+ : A \times A \rightarrow A$.

We interpret $p + q$ as the system that is formed or aggregated from the parts $p$ and $q$. For instance, $A$ may be the set of traces and $+$ trace union. For the time being, the algebra $(A, +)$ is assumed to be absolutely free, that is, $+$ need not satisfy any laws. Later we will assume aggregation algebras that are (commutative) semigroups or monoids.

Definition 3.2. An independence relation is a binary relation $R$ on $A$ that is bilinear in the following sense:

\[
R(p + q, r) \iff R(p, r) \land R(q, r),
\]

\[
R(p, q + r) \iff R(p, q) \land R(p, r).
\]

A system $p$ is independent of a system $q$ if $R(p, q)$ holds.

The linearity conditions say that a combined system is independent of another one if and only if both its parts are.

Example 3.3.

1. Our first example of an independence relation is $\not\leftarrow$ as given in Def. 2.2.

2. Consider the aggregation algebra $(\mathcal{P}(A), \cup)$, where $\cup$ is set union. Then, for all $X, Y \subseteq A$, the relation defined by $R(X, Y)$ if and only if $X$ and $Y$ are disjoint is an independence relation. This holds since $(X \cup Y) \cap Z = \emptyset$ if and only if $X \cap Z = \emptyset$ and $Y \cap Z = \emptyset$.

3. Consider the set $(G, \cup)$ of digraphs under (disjoint) union. Then, for all digraphs $g_1, g_2 \in G$, the relation defined by $R(g_1, g_2)$ if and only if there is no arrow with source in $g_1$ and target in $g_2$ is an independence relation. The same facts hold for digraphs with respect to arrows from $g_2$ to $g_1$ and for undirected graphs with respect to adjacency.

4. Consider subspaces of some vector space with $+$ being the span. Then orthogonality is an independence relation.

5. Let $t_1$ and $t_2$ be subtrees of a tree $t$. Let them be related by $R$ if their roots are not on a single path from the root of $t$ to its leaves. Let $+$ correspond to forming the least subtree of $t$ that has both $t_1$ and $t_2$ as subtrees. Then $R$ is not a dependence relation in the above sense, because a tree $t_3$ which is related by $R$ to both $t_1$ and $t_2$ can be “captured” as a subtree of $t_1 + t_2$.

Examples 1–4 show that some natural notions of dependence are covered by the above definition, whereas Example 5 shows that some other natural notions, such as disjointness of subtrees in a tree, are not. □
Lemma 3.4. Let \((A, +)\) be an aggregation algebra and let \(R\) be an independence relation. Then

1. \(R(p + q + r, s) \iff R(p + (q + r), s)\).
2. \(R(p, (q + r) + s) \iff R(p, q + (r + s))\).
3. \(R(p + q, r) \iff R(q + p, r)\).
4. \(R(p, q + r) \iff R(p, r + q)\).
5. \(R(p + p, q) \iff R(p, q)\).
6. \(R(p, q + q) \iff R(p, q)\).
7. \(R(p + q, r) \land R(p, q) \iff R(q, r) \land R(p, q + r)\).

Proof. By bilinearity and the fact that conjunction is associative, commutative and idempotent. \(\square\)

We now consider two independence relations \(R\) and \(S\).

Lemma 3.5. Let \((A, +)\) be an aggregation algebra. Let \(R\) and \(S\) be independence relations that satisfy \(R \subseteq S\).

1. \(R(p + q, r) \land S(p, q) \implies S(p, q + r) \land R(q, r)\).
2. \(R(p, q + r) \land S(q, r) \implies S(p + q, r) \land R(p, q)\).

Proof. We only prove Part 1; Part 2 is similar.

\[
R(p + q, r) \land S(p, q) \iff R(p, r) \land R(q, r) \land S(p, q) \\
\implies S(p, r) \land R(q, r) \land S(p, q) \\
\iff R(q, r) \land S(p, q + r).
\]

\(\square\)

Next we prove a property that will imply the crucial inequational exchange law mentioned in the introduction. We write \(S^\sim\) for the relational converse of \(S\).

Proposition 3.6. Let \((A, +)\) be an aggregation algebra (absolutely free). Let \(R\) and \(S\) be two independence relations. Let \(R \subseteq S\) and \(S = S^\sim\) (\(S\) is symmetric). Then

\[
R(p + q, r + s) \land S(p, q) \land S(r, s) \iff R(p, r) \land R(q, s) \land S(p + r, q + s).
\]

Proof.

\[
R(p + q, r + s) \land S(p, q) \land S(r, s) \\
\iff R(p, r) \land R(q, r) \land R(p, s) \land R(q, s) \land S(p, q) \land S(r, s) \\
\implies R(p, r) \land S(q, r) \land S(p, s) \land R(q, s) \land S(p, q) \land S(r, s) \\
\implies R(p, r) \land R(q, s) \land S(r, q) \land S(p + r, s) \land S(p, q) \\
\implies R(p, r) \land R(q, s) \land S(p + r, q) \land S(p + r, s) \\
\iff R(p, r) \land R(q, s) \land S(p + r, q + s).
\]

\(\square\)

The proofs in this section are only intended to give a flavour of the approach. In fact, they have all been automated, hence formally verified, with Prover9.
4. Algebraisation of the Calculus

This section further pursues the idea of interpreting independence arrows as algebraic operators. Formally, the algebraisation is achieved by lifting the aggregation algebra to powersets.

Definition 4.1. For an aggregation algebra \((A, +)\) and an independence relation \(R\), we define an operator \(\circledast_R\) of \(R\)-composition (or complex product w.r.t. \(R\)) of type \(P(A) \times P(A) \to P(A)\) for all \(a, b \subseteq A\) by

\[
a \circledast_R b = \{ p + q \mid p \in a \wedge q \in b \wedge R(p, q) \}.
\]

Example 4.2.

1. In Section 2 we have \(\circledast_-\circledast=\circledast\).

2. Let \(A = \Sigma^*\) be the set of strings over the alphabet \(\Sigma\). For all \(a, b \in \Sigma^*\) let \(a + b\) be string concatenation and let \(R\) be the identity relation. Let \(B, C \subseteq \Sigma^*\) be sets of strings. Then \(A \circledast_R B\) is the usual complex product of regular language theory.

Now, in order to obtain more interesting results, we assume a semigroup or monoidal structure on the aggregation algebra. In some cases, we also consider independence relations that are not only bilinear, but also bistrict, that is, they satisfy

\[
R(p, 0) \quad \text{and} \quad R(0, p),
\]

whenever the aggregation algebra has a unit \(0\) that plays the rôle of the empty system. The conditions say that the empty system depends on nothing and nothing depends on it. These conditions seem rather natural.

Call an equational law involving a function linear if every variable in it occurs exactly once on each side of the equation. Such laws are inherited by the pointwise extension (see e.g. [28]). Typical examples of such laws are associativity and commutativity. This entails the following result.

Proposition 4.3.

1. Let \((A, +)\) be a semigroup and \(R\) be bilinear. Then \((P(A), \circledast_R)\) is a semigroup.

2. Let \((A, +, 0)\) be a monoid, and \(R\) be bilinear and bistrict. Then \((P(A), \circledast_R, \{0\})\) is a monoid.

5. Semirings and Quantales

In powerset algebras, next to the pointwise extensions of basic aggregation algebras, we have all the set theoretic operations available. As already mentioned in Section 2 the most interesting one for us is set union, since it allows modelling nondeterminacy. This is reflected in the following definition.

Definition 5.1.

1. An idempotent semiring is a structure \((A, +, \cdot, 0, 1)\) such that \((A, +, 0)\) is a commutative monoid with idempotent addition, that is, \(a + a = a\) for all \(a \in A\), \((A, \cdot, 1)\) is a monoid, multiplication distributes over addition, that is, for all \(a, b, c \in A\),

\[
a \cdot (b + c) = a \cdot b + a \cdot c \quad \text{and} \quad (a + b) \cdot c = a \cdot c + b \cdot c,
\]

and \(0\) is a left and right annihilator for multiplication, that is, for all \(a \in A\),

\[
a \cdot 0 = 0 = 0 \cdot a.
\]
2. Every idempotent semiring is partially ordered by

\[ a \leq b \iff a + b = b. \]

Then + and are isotope w.r.t. \( \leq \) and 0 is the least element. Moreover, \( a + b \) is the supremum of \( a, b \in A \).

3. A idempotent semiring is called a quantale \[34\] or standard Kleene algebra \[10\] if \( \leq \) induces a complete lattice and multiplication distributes over arbitrary suprema. The infimum and the supremum of a subset \( B \subseteq A \) are denoted by \( \sqcap B \) and \( \sqcup B \), respectively. Their binary variants are \( a \sqcap b \) and \( a \sqcup b \) (the latter coinciding with \( a + b \)).

Quantales have been used in many contexts other than that of program semantics (cf. the c-semirings of \[11\] or the general reference \[35\]). They have the advantage that the general fixpoint calculus is available there. A number of our proofs in Section 10 need the principle of fixpoint fusion which is a second-order principle; in the first-order setting of conventional Kleene and omega algebra (see Section 8) only special cases of it, like the induction and coinduction rules, can be added as axioms. Moreover, in every quantale left and right residuals w.r.t. multiplication can be defined by the Galois connections

\[ x \leq a/b \iff x \cdot b \leq a \]

\[ x \leq b/a \iff b \cdot x \leq a. \]

Let again \( PR(EV) \) denote the set of all programs over the event set \( EV \) (cf. Definition 2.1). The following fact is immediate from the observations in Section 2.

**Lemma 5.2.** \((PR(EV), \cup, \text{false}, *, \text{skip})\) and \((PR(EV), \cup, \text{false}, ;, \text{skip})\) are quantales. In each of them \( \top = \mathcal{P}(EV) \) is the most general program over \( EV \).

**Proposition 5.3.** Let \((A, +, 0)\) be a monoid and \( R \) be bilinear and bistrict. Then \((\mathcal{P}(A), \cup, R, \emptyset, \{0\})\) is an idempotent semiring.

This follows again from standard results about pointwise extension mentioned in Section 4 (cf. \[28\]).

6. Concurrent Algebras

The results of the previous section can readily be generalised to more than one independence relation. Here, we consider only the case of two such relations, \( S \) and \( T \), which are defined over one single aggregation algebra.

**Definition 6.1.**

1. A bisemigroup is a structure \((A, *, ;)\) such that \((A, *)\) and \((A, ;)\) are semigroups. A bimonoid is a structure \((A, *, ;, 1)\) such that \((A, *, 1)\) and \((A, ;, 1)\) are monoids.

2. An idempotent bisemiring is a structure \((A, +, *, ;, 0, 1)\) such that \((A, +, *, 0, 1)\) and \((A, +, ;, 0, 1)\) are idempotent semirings.

We could define bimonoids etc. more generally with two different units \(1_\ast\) and \(1_\cdot\), but in the cases we consider the operators share one single unit.

The following statement is immediate from the results of the previous section.

**Proposition 6.2.** Let \((A, +)\) be a semigroup and let \( R \) and \( S \) be bilinear. Then \((\mathcal{P}(A), \cup, R, S, 0)\) is an idempotent bisemiring.

In the above statement the independences \( R \) and \( S \) were unrelated. We now consider the situation where one of them is contained in the other, as in Section 2. This allows us to lift the statements of Lemma 3.5 and the exchange law (Proposition 3.6) to the powerset level.
Lemma 6.3.

1. Let \((A, +)\) be an aggregation algebra and let \(R\) and \(S\) be independence relations on \(A\). Then \(R \subseteq S\) implies \(a \circ R b \subseteq a \circ S b\).

2. Let \((A, +)\) be a commutative semigroup and let \(R\) be a symmetric independence relation on \(A\). Then \(a \circ R b = b \circ R a\).

Proof. The proof of the first statement is entirely trivial. We display the proof of the second statement to show the role of commutativity.

\[
p \in a \circ R b \iff \exists q, r : (p = q + r \land q \in a \land r \in b \land R(q, r))
\]

\[
\iff \exists q, r : (p = r + q \land q \in a \land r \in b \land R(r, q))
\]

\[
\iff p \in b \circ R a.
\]

\[
\square
\]

Proposition 6.4. Let \((A, +)\) be a semigroup and let \(R\) and \(S\) be bilinear independence relations such that \(R \subseteq S\). Then

1. \((a \circ S b) \circ R c \subseteq a \circ (b \circ R c)\),
2. \(a \circ (b \circ S c) \subseteq (a \circ R b) \circ S c\).

Proof. We only prove the first inequality.

\[
p \in (a \circ S b) \circ R c \iff \exists q, r, s : (p = q + r + s \land q \in a \land r \in b \land s \in c \land R(q + r, s) \land S(q, r))
\]

\[
\iff \exists q, r, s : (p = q + (r + s) \land q \in a \land r \in b \land s \in c \land S(q, r + s) \land R(r, s))
\]

\[
\iff p \in a \circ (b \circ R c).
\]

The second step uses associativity of +, Lemma 3.5.1 and bilinearity. \(\square\)

Proposition 6.5. Let \((A, +)\) be a commutative semigroup. Let \(R\) and \(S\) be bilinear independence relations such that \(R \subseteq S\) and \(S\) is symmetric. Then the following exchange law holds:

\[(a \circ S b) \circ (c \circ S d) \subseteq (a \circ R c) \circ (b \circ R d).\]

Proof.

\[
p \in (a \circ S b) \circ (c \circ S d) \iff \exists q, r, s, t : (p = (q + r) + (s + t) \land q \in a \land r \in b \land s \in c \land t \in d
\]

\[
\land R(q + r, s + t) \land S(q, r) \land S(s, t)
\]

\[
\iff \exists q, r, s, t : (p = (q + s) + (r + t) \land q \in a \land r \in b \land s \in c \land t \in d
\]

\[
\land R(q, s) \land R(r, t) \land S(q + s, r + t)
\]

\[
\iff b \in (a \circ R c) \circ (b \circ R d).
\]

The second step uses associativity and commutativity of the aggregation algebra and Proposition 3.6. \(\square\)

These results motivate the following definitions, abstracting \(\circ\) to \(;\) and \(\circ\) to \(*\).

Definition 6.6.
1. An ordered semigroup is a structure \((A, \cdot, \leq)\) such that \((A, \cdot)\) is a semigroup, \(A\) is partially ordered by \(\leq\) and \(\cdot\) is isotone in both arguments. An ordered monoid is a structure \((A, \cdot, 1, \leq)\) such that \((A, \cdot, \leq)\) is an ordered semigroup and \((A, \cdot, 1)\) is a monoid.

2. An ordered bisemigroup is a structure \((A, \ast, \cdot, \leq)\) such that \((A, \ast, \cdot, \leq)\) and \((A, \cdot, \leq)\) are ordered semigroups. An ordered bimonoid is defined analogously.

3. A concurrent semigroup is an ordered bisemigroup \((A, \ast, \cdot, \leq)\) that satisfies
\[
\begin{align*}
  a \cdot b &\leq a \ast b, \\
  a \ast b &= b \ast a, \\
  (a \ast b) \cdot c &\leq a \ast (b \cdot c), \\
  a \cdot (b \ast c) &\leq (a \cdot b) \ast c, \\
  (a \ast b) \cdot (c \ast d) &\leq (a \cdot c) \ast (b \cdot d).
\end{align*}
\]

4. A concurrent monoid is an ordered bimonoid \((A, \ast, \cdot, 1, \leq)\) that satisfies
\[
\begin{align*}
  a \cdot b &= b \ast a, \\
  (a \ast b) \cdot (c \ast d) &\leq (a \cdot c) \ast (b \cdot d).
\end{align*}
\]

5. A concurrent semiring is an idempotent bisemiring \((A, +, \ast, \cdot, 0, 1)\) such that \((A, \ast, \cdot, \leq, 1)\) is a concurrent monoid, where \(\leq\) is the natural semiring order.

6. A concurrent semiring \((A, +, \ast, \cdot, 0, 1)\) is called a concurrent quantale if \((A, +, \ast, 0, 1)\) and \((A, +, \cdot, 0, 1)\) are quantales.

**Lemma 6.7.** The above axioms for concurrent semigroups and concurrent semirings are irredundant.

**Proof.** We have used Mace4 to find models in which all but one of the axioms are true and the remaining axiom is false, for each combination. □

The unit 1 allows us to replace the two concurrent bimonoid axioms by the single one
\[
(a \ast b) \cdot (c \ast d) \leq (b \cdot c) \ast (a \cdot d),
\]
which has its free variables in a different order than (7). Moreover, every concurrent bimonoid is a concurrent semigroup, as can be shown by automated theorem proving:

**Lemma 6.8.** The concurrent monoid axioms entail the identities
\[
\begin{align*}
  a \cdot b &\leq a \ast b, \\
  (a \ast b) \cdot c &\leq a \ast (b \cdot c), \\
  a \cdot (b \ast c) &\leq (a \cdot b) \ast c.
\end{align*}
\]

Moreover, Mace4 yields a two-element counterexample showing that these two laws do not imply the exchange axiom (7).

The development so far can be summed up in the following theorem.

**Theorem 6.9.** Let \((A, +)\) be a commutative semigroup and let \(R\) and \(S\) be bilinear independence relations with \(S \subseteq R\). Then \((P(A), R, S, 0)\) is a concurrent bimonoid.

This theorem shows that the entire structure of concurrent algebras can be obtained from the very general assumption of a (commutative) monoidal aggregation algebra and two (strict and) bilinear independence relations.
7. Generalised Sequential and Concurrent Composition

We now check that independence relations for generalised variants of sequential and concurrent composition operators from Section 2 satisfy the strictness and bilinearity conditions.

For these particular operators, we assume a distributive lattice \((A, +, \sqcap, 0)\) with least element 0 as the underlying aggregation algebra. This is compatible with all assumptions in previous statements. We also use a strict and additive operator \(f : A \rightarrow A\), that is, it satisfies

\[
f(0) = 0 \quad \text{and} \quad f(p + q) = f(p) + f(q).
\]

Such an operator arises, for instance, as the preimage operator over a relational structure, defined as

\[
f(p) = \{a \mid \exists b \in p : R(a, b)\} \quad \text{and} \quad f(p) = \{a \mid \exists b \in p : R^+(a, b)\},
\]

where \(R^+\) denotes the transitive closure of a relation \(R\). In that concrete setting, \(p\) is a set.

In our original definitions \(R\) would be \(\rightarrow\) and \(f\) the following function \(\text{dep}\) that yields the set of events on which \(tp\)-events depend directly or indirectly.

**Definition 7.1.** For a trace \(tp\), we define the set

\[
\text{dep}(tp) =_{df} \{f \mid \exists e \in tp : f \rightarrow^+ e\}.
\]

We then consider the following operators:

- **fine-grain concurrent composition** \(ab\) with \((*)(p, q) \Leftrightarrow p \sqcap q = 0;\)
- **weak sequential composition** \(a; b\) with \((.;)(p, q) \Leftrightarrow (*)(p, q) \land f(p) \sqcap q = 0;\)
- **disjoint parallel composition** \(a||b\) with \((||)(p, q) \Leftrightarrow (;)(p, q) \land p \sqcap f(q) = 0;\)
- **alternation** \(a\mathbb{0}b\) with \((\mathbb{0})(p, q) \Leftrightarrow p = 0 \lor q = 0.\)

In contrast to Section 2, we use prefix notation here for the relations to emphasise the connection to Section 6.

**Lemma 7.2.**

1. \((\mathbb{0}) \subseteq (||) \subseteq (;) \subseteq (*).\)
2. \((||) = (||)^-, (||) = (||)^-, (.;) \neq (.;)^-, (\ast) = (\ast)^-\).
3. \((\mathbb{0}), (||), (;)\) and \((\ast)\) are bilinear.
4. \((\mathbb{0}), (||), (;)\) and \((\ast)\) are bistrict.

**Proof.** The proofs of (1), (2) and (4) are trivial, so we only consider case (3).

- **Fine-grain concurrent composition.**
  \[
  (\ast)(p + q, r) \Leftrightarrow (p + q) \sqcap r = 0 \Leftrightarrow p \sqcap r = 0 \land q \sqcap r = 0 \Leftrightarrow (\ast)(p, r) \land (\ast)(q, r).\]

The second linearity condition is similar.

- **Weak sequential composition.**
  \[
  (;)(p + q, r) \Leftrightarrow (\ast)(p + q, r) \land f(p + q) \sqcap r = 0
  \Leftrightarrow (\ast)(p, r) \land (\ast)(q, r) \land (f(p) + f(q)) \sqcap r = 0
  \Leftrightarrow (\ast)(p, r) \land (\ast)(q, r) \land f(p) \sqcap r = 0 \land f(q) \sqcap r = 0
  \Leftrightarrow (.;)(p, r) \land (.;)(q, r).
  \]

The second linearity condition is again similar.
Definition 8.1.

1. A Kleene algebra $[26]$ is a structure $(A, +, \cdot, *, 0, 1)$ such that $(A, +, \cdot, 0, 1)$ is an idempotent semiring and the star operator $*$ satisfies the unfold and induction laws

\[
1 + a \cdot a^* \leq a^*, \quad 1 + a^* \cdot a \leq a^*,
\]

\[
c + a \cdot b \leq b \Rightarrow a^* \cdot c \leq b, \quad c + b \cdot a \leq b \Rightarrow c \cdot a^* \leq b.
\]

The star here should not be confused with the separation operator $\tau$ above.

2. 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.

3. An omega algebra $[9]$ is a structure $(A, +, \cdot, *, \omega, 0, 1)$ such that $(A, +, \cdot, *, 0, 1)$ is a Kleene algebra and the omega operator $\omega$ satisfies the unfold and coinduction laws

\[
a^\omega \leq a \cdot a^\omega,
\]

\[
b \leq c + a \cdot b \Rightarrow b \leq a^\omega + a^* \cdot c.
\]

The axioms of Kleene and omega algebras entail many useful laws. As examples we mention

\[
1 \leq a^*, \quad a \leq a^*, \quad a^* \cdot a = (a^*)^* = a^*, \quad (a + b)^* = a^* \cdot (b \cdot a^*)^*, \quad (a + b)^\omega = a^\omega + a^* \cdot b \cdot (a + b)^\omega.
\]

KA

\[
1 = \top, \quad (a \cdot b)^\omega = a \cdot (b \cdot a)^\omega, \quad (a + b)^\omega = a^\omega + a^* \cdot b \cdot (a + b)^\omega.
\]

OA

It is well known that in a quantale $A$, the finite iteration $a^*$ exists for all $a \in A$ and is given by $a^* = \mu x. 1 + a \cdot x$, where $\mu$ denotes the least fixpoint operator. Since in a quantale the defining function for star is continuous, Kleene’s fixpoint theorem shows that $a^* = \bigsqcup_{n \in \mathbb{N}} a^n$. If the complete lattice $(A, \leq)$ in a quantale $A$ is completely distributive, i.e., if $+$ distributes over arbitrary infima, then also the infinite iteration $a^\omega$ exists for all $a \in A$ and is given by $a^\omega = \nu x. a \cdot x$, where $\nu$ denote the greatest fixpoint operator.

We now define concurrent versions of these types of algebras.

Definition 8.2.

1. A bi-Kleene algebra is a structure $(A, +, *, ;, \otimes, 0, 1)$ such that $(A, +, *, 0, 1)$ and $(A, +, ;, \otimes, 0, 1)$ are Kleene algebras.

2. A concurrent Kleene algebra (CKA) is a bi-Kleene algebra $(A, +, *, ;, \otimes, 0, 1)$ over a concurrent bimonoid $(A, *, ;, 1)$.

3. Bi-omega algebras and concurrent omega algebras are defined analogously.
The above discussion entails the following result.

**Theorem 8.3.** Let \((A,\cdot,1)\) be a commutative monoid and let \(R\) and \(S\) be bilinear and bistrict independence relations with \(S \subseteq R\). Then the structure \(\langle P(A)\cup{\emptyset},\cup,\emptyset,\emptyset,0,1\rangle\) with \(a^{\bigcirc} = df \mu x.1 + a\overline{1}\) for \(T \in \{R,S\}\) is a concurrent Kleene algebra. An analogous property holds for omega iteration.

**Corollary 8.4.** Let \(A\) be a bounded distributive lattice and let \(*\) and \(:\) be defined as in Section 7. Then with \(a^{\bigcirc} = df \mu x.1 + a\cdot x\) and \(a^{\bigcirc} = df \mu x.1 + a\cdot x\) the structure \(\langle P(A)\cup{\emptyset},\cup,*,\bigcirc,\emptyset,\{0\}\rangle\) is a concurrent Kleene algebra.

We now explain the behaviour of iteration in our program quantales. For a program \(P\), the program \(P^{\bigcirc}\), denoted by \(P^\omega\) in [20], consists of all sequential compositions of finitely many traces in \(P\). The program \(P^{\bigcirc}\) 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 a primitive action \(a\); the iterated program just needs to supply sufficiently many (e.g., countably many) events that stand for occurrences of \(a\) and can then use a fresh one of these in each round of iteration.

**Example 8.5.** With the notation of Example 2.3 let \(P = df P_z \cup P_y \cup P_z\). We first look at the powers of \(P\) w.r.t. \(*\) composition:

\[
P^2 = P \cdot P = [ax,ay] \cup [ax,az] \cup [ay,az],
\]

\[
P^3 = P \cdot P \cdot P = [ax,ay,az].
\]

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 \(P^4 \neq P^1 \cdot P = \emptyset\), and hence strictness of \(*\) w.r.t. \(\emptyset\) implies \(P^n = \emptyset\) for all \(n \geq 4\). Therefore \(P^{\bigcirc}\) consists of all traces with at most three events from \(\{ax,ay,az\}\) (the empty trace is in \(P^{\bigcirc}\), too, since by definition \(\text{skip}\) 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 10.

It turns out that for the powers of \(P\) w.r.t. the operator \(:\), we obtain exactly the same expressions, since for every program \(Q = [e] \cup [f]\) with \(e \neq f\) we have

\[
Q : Q = ([e] \cup [f]) : ([e] \cup [f]) = [e] : [e] \cup [e] \cup [f] \cup [f] : [e] \cup [f] : [f] = [e,f] = Q \cdot Q,
\]

provided \(e \not\leftrightarrow f\) or \(f \not\leftrightarrow e\), i.e., provided the trace \([e,f]\) is consistent with the dependence relation. Only if there were a cyclic dependence \(e \leftrightarrow f \leftrightarrow e\) we would have \(Q : Q = \emptyset\), whereas still \(Q \cdot Q = [e,f]\).

Since \(PR(EV)\) is a power set lattice, it is completely distributive. Hence it forms an concurrent omega algebra. 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 (cf. [39]).

### 9. Hoare Calculus

Essential tools for reasoning about programs are the Hoare calculus and its variants for the concurrent setting. We now show how to treat the Hoare calculus algebraically in our setting. In [20], Hoare triples relating programs are defined by \(P \{Q\} R \Rightarrow df P; Q \subseteq R\). Hence such a triple expresses that the program \(Q\) is guaranteed to extend every trace in the “pre-history” \(P\) to a trace in \(R\).

Again, it is beneficial to abstract from the concrete case of programs.

**Definition 9.1.** Given an ordered monoid \((A,\leq,\cdot,1)\) we define, for elements \(a,b,c \in A\), the **Hoare triple** \(a \{b\} c\) by

\[
a \{b\} c \iff df a \cdot b \leq c.
\]
We show that this very general definition entails all the familiar properties of Hoare triples.

**Lemma 9.2.** Assume an ordered monoid \((A,\leq,\cdot,1)\).

1. \(a\{1\}c \iff a \leq c\); in particular, \(a\{1\}a \iff \text{TRUE}\). (skip)
2. \((\forall a,c : a\{b\}c \Rightarrow a\{b'\}c) \iff b' \leq b\). (antitony)
3. \((\forall a,c : a\{b\}c \iff a\{b'\}c) \iff b = b'\). (extensionality)
4. \(a\{b \cdot b'\}c \iff \exists d : a\{b\}d \land d\{b'\}c\). (composition)
5. \(a \leq d \land d\{b\}e \land e \leq c \Rightarrow a\{b\}e\). (weakening)
6. \(a\{0\}c \iff \text{TRUE},\) (failure)
7. \(a\{b + b'\}c \iff a\{b\}c \land a\{b'\}c\). (choice)

If \((A,\cdot,1)\) is the multiplicative reduct of an idempotent semiring \((A,+,0,\cdot,1)\) and the order used in the definition of Hoare triples is the natural semiring order, we also have

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

**Proof.**

1. Immediate from the definitions and neutrality of \(1\).
2. \((\Leftarrow)\) follows directly from isotony of composition. For \((\Rightarrow)\) set \(a = 1\) and \(c = b'\), and expand the definition.
3. Immediate from Part 2 and antisymmetry of \(\leq\).
4. \((\Leftarrow)\) By the definitions, isotony of \(\cdot\) and transitivity of \(\leq\),
\[
a\{b\}d \land d\{b'\}c \iff a \cdot b \leq d \land d \cdot b' \leq c \Rightarrow a \cdot b \cdot b' \leq c \iff a\{b \cdot b'\}c.
\]
\((\Rightarrow)\) Choose \(d = a \cdot b\).
5. By isotony and the assumptions, \(a \cdot b \leq d \cdot b \leq e \leq c\).
6. Immediate from the definitions and the annihilation property of \(0\).
7. By the definitions, distributivity and the definition of the supremum,
\[
a\{b + b'\}c \iff a \cdot (b + b') \leq c \iff a \cdot b + a \cdot b' \leq c
\leq a \cdot b \leq c \land a \cdot b' \leq c \iff a\{b\}c \land a\{b'\}c.
\]
8. The implication \((\Leftarrow)\) of the first equivalence follows from Part 2 and \(b \leq b^+\). For \((\Rightarrow)\) we have, using the definitions, the second star induction rule in (10) and idempotence of \(+\),
\[
a\{b^+\}a \iff a \cdot b \cdot b^* \leq a \iff a \cdot b^+ \leq a \iff a \cdot b \leq a \iff a\{b\}a.
\]
The second equivalence follows from \(b^* = 1 + b^+\) and the skip and choice rules. \(\square\)
Lemma 9.2 can be expressed more concisely in relational notation. For \( b \in A \) the relation \( \{ b \} \subseteq A \times A \) between precondition elements \( a \) and postcondition elements \( c \) is defined by
\[
\forall a, c : a \{ b \} c \iff a \cdot b \leq c.
\]

Then the above properties rewrite into

1. \( \{1\} = \leq. \)
2. \( \{b\} \subseteq \{b'\} \iff b' \leq b. \)
3. \( \{b\} = \{b'\} \iff b = b'. \)
4. \( \{b \cdot b'\} = \{b\} \circ \{b'\} \) where \( \circ \) means relational composition.
5. \( \leq \circ \{b\} \circ \leq \subseteq \{b\}. \)
6. \( \{0\} = \top \) where \( \top \) 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 9.3.** Assume again an ordered monoid \((A, \leq, \cdot, 1)\). Then
\[
(\forall a, d, c : a \{ b \} d \land d \{ b' \} c \Rightarrow a \{ e \} c) \iff e \leq b \cdot b'.
\]

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

**Lemma 9.4.** Let \( A = (A, +, 0, *, ;) \) be a concurrent semigroup and \( a, a', b, b', c, c', d \in A \) with \( a \{ b \} c \) interpreted as \( a ; b \leq c \).

1. \( a \{ b \} c \land a' \{ b' \} c' \Rightarrow (a \ast a') \{ b \ast b' \} (c \ast c'). \) (concurrency)
2. \( a \{ b \} c \Rightarrow (d \ast a) \{ b \} (d \ast c). \) (frame rule)

**Proof.**

1. \( a \{ b \} c \land a' \{ b' \} c' \)
\[
\iff \quad \text{[definition]}
\]
\[
a ; b \leq c \land a' ; b' \leq c'
\]
\[
\Rightarrow \quad \text{[isotony of \( \cdot \)]}
\]
\[
(a ; b) \ast (a' ; b') \leq c \ast c'
\]
\[
\Rightarrow \quad \text{[exchange (7)]}
\]
\[
(a \ast a') ; (b \ast b') \leq c \ast c'
\]
\[
\iff \quad \text{[definition]}
\]
\[
(a \ast a') \{ b \ast b' \} (c \ast c')
\]

2. \( a \{ b \} c \)
\[
\iff \quad \text{[definition]}
\]
\[
a ; b \leq c
\]
\[
\Rightarrow \quad \text{[isotony of \( \ast \)]}
\]
\[
d \ast (a ; b) \leq d \ast c
\]
⇒ \{ \text{by Lemma 6.8.5} \}
(d * a) ; b \leq d * c
⇔ \{ \text{definition} \}
(d * a) \{b\} (d * c).

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. For
instance, 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.

10. Invariants

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

Definition 10.1. 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 $\text{skip} = \mathcal{P}(\emptyset) = \{\emptyset\}$. The term “invariant”
expresses that a program often 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 10.2. Consider again the event set $EV$ from Example 2.3. 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 $R = d \mathcal{P}(E)$. The fact that we want $P$ to be executed only in such environments is
expressed by forming the parallel composition $P \ast R$.

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 13, invariants will be used to express properties of the environment on which a program wants to
rely (whence the identifier $R$).

Power invariants satisfy many useful laws. To state them, we want to define a function that maps
a program to the smallest power invariant containing it. Let $|P| = \bigcup P$ denote the set of all events
occurring in traces of a program $P$; when convenient, $|P|$ can also be considered as a trace.

It is straightforward to check that $|$ distributes through arbitrary unions. Hence it has an upper adjoint
$F$, defined by the Galois connection

$|P| \subseteq X \Leftrightarrow P \subseteq F(X)$.

This entails $F(X) = \mathcal{P}(X)$ and $|\mathcal{P}(X)| = X$. Moreover, as adjoints of a Galois connection, $\mathcal{P}(\_)$ and $\_|$ are
$\subseteq$-isotone. Setting $X = |P|$ we obtain $P \subseteq \mathcal{P}(|P|)$. Finally, for $X, Y \subseteq EV$ we have $\mathcal{P}(X) \subseteq \mathcal{P}(Y) \Leftrightarrow X \subseteq Y$. 16
Motivated by the above remarks, we now define \( \text{INV}(P) = \{ \mathcal{P}(|P|) \} \). Then \( \text{INV}(P) \) is the most general program that can be formed from the events of \( P \). As a composition of isotone functions, \( \text{INV} \) is isotone, too.

We now prepare for our abstract notion of invariant. An invariant is a program \( R \) with \( R = \text{INV}(R) \). In particular, every invariant in our concrete quantale of programs is a power invariant. In general concurrent semirings we will replace \( \text{INV} \) by a suitable abstract operator the properties of which will be discussed below. By definition, invariants are fixpoints of an isotone function and hence form a complete lattice under the inclusion order.

The operator \( \nabla \) from [20] and \( \text{INV} \) are interrelated. To this end we set \( \text{SINGLES}(P) = \{ \{ e \} | e \in P \} \). Then

\[
\text{INV}(\text{SINGLES}(Q)) = Q \nabla Q , \quad Q \nabla R = \text{INV}(\text{SINGLES}(Q \cup R)) .
\]

We shall use \( \text{INV} \), since it leads to simpler and more intuitive formulations.

We give a few useful properties of \( \text{INV} \).

**Theorem 10.3.** Let \( P \) and \( Q \) be programs.

1. \( \text{INV}(P) \) is the smallest invariant containing \( P \).
2. \( \text{INV}(\text{INV}(P)) = \text{INV}(P) \); hence \( \text{INV}(P) \) is an invariant.
3. \( \text{INV} \) is a closure operator.
4. \( \text{skip} \subseteq \text{INV}(P) \).
5. \( \text{INV}(P * Q) \subseteq \text{INV}(P \cup Q) \).
6. \( \text{INV}(P) * \text{INV}(P) \subseteq \text{INV}(P) \).

**Proof.**

1. We have already seen above that \( P \subseteq \text{INV}(P) \). Let \( S \) be another invariant with \( P \subseteq S \). Then, by isotony of \( \text{INV} \) and the definition of invariants, \( \text{INV}(P) \subseteq \text{INV}(S) = S \).
2. Since, as remarked above, \( \mathcal{P}(\{ e \}) = X \), we have

\[
\text{INV}(\text{INV}(P)) = \mathcal{P}(\{ \mathcal{P}(|P|) \}) = \mathcal{P}(|P|) = \text{INV}(P) .
\]
3. By Part 1 we have \( P \subseteq \text{INV}(P) \). By the Galois connection \( \text{INV} \) is isotone and by Part 2 it is idempotent.
4. Immediate from the definition of \( \text{INV} \).
5. By the definition of \( * \) we have \( |P * Q| \subseteq |P \cup Q| \) and the property follows by isotony of \( \mathcal{P} \).
6. From the definitions it is straightforward to check that \( |P * Q| \subseteq |P| \cup |Q| \). Hence

\[
\text{INV}(P) * \text{INV}(P) \subseteq \text{INV}(\text{INV}(P) * \text{INV}(P)) = \mathcal{P}(\text{INV}(P) * \text{INV}(P)) \subseteq \mathcal{P}(\text{INV}(P) \cup \text{INV}(P)) = \text{INV}(\text{INV}(P))
\]

by Parts 1 and 2.

Since \( \text{INV} \) is a closure operator we have the following (cf. [5]).

**Corollary 10.4.** For set \( \mathcal{R} \) of power invariants, \( \bigcap \mathcal{R} \) and \( \text{INV}(\bigcup \mathcal{R}) \) are the meet and join of \( \mathcal{R} \) in the complete lattice of invariants, respectively.

We now abstract again from the concrete case of programs. It turns out that the properties in Theorem [10.3] and [10.3] largely suffice for characterising invariants.
Definition 10.5. An invariant in a concurrent bimonoid $A$ is an element $r \in A$ satisfying $1 \leq r$ and $r \ast r \leq r$. In a concurrent semiring these two axioms can equivalently be combined into $1 + r \ast r \leq r$. The set of all invariants of $A$ is denoted by $I(A)$.

We now give a number of algebraic properties of invariants that are useful in proving the soundness of the rely/guarantee-calculus in Section 13.

Theorem 10.6. Assume a concurrent bimonoid $A$, an $r \in I(A)$ and arbitrary $a, b \in A$.

1. $a \leq r \ast a$ and $a \leq a \ast r$.
2. $r ; r \leq r$.
3. $r \ast r = r ; r$.
4. $r ; (a \ast b) \leq (r ; a) \ast (r ; b)$ and $(a \ast b) ; r \leq (a ; r) \ast (b ; r)$.
5. $r ; a ; r \leq r \ast a$.
6. If $A$ is a CKA then $r \in I(A) \iff r = r \circledast$.
7. If $A$ is a CKA then the least invariant comprising $a$ is $a \circledast$.

Proof.

1. By neutrality of 1 and isotony of $\ast$ we have $a = 1 \ast a \leq r \ast a$. The proof of the second inequality is symmetric.
2. This is immediate from $a ; b \leq a \ast b$ and transitivity of $\leq$.
3. By Part 1 we have $r \leq r \ast r$; the converse equation holds by definition and Part 2 respectively.
4. $r ; (a \ast b) \leq \{ \text{by Lemma 6.8.6} \}$
   $$ (r ; a) \ast b \leq \{ \text{by Part 1 and isotony} \}$$
   $$(r ; a) \ast (r ; b).$$
   The proof of the second law is symmetric.
5. $r ; a ; r \leq \{ \text{by Part 1} \}$
   $$r \ast a \ast r = \{ \text{commutativity of $\ast$} \}$$
   $$r \ast r \ast a \leq \{ \text{definition of invariants} \}$$
   $$r \ast a .$$
6. $(\Rightarrow)$ By the definition of invariants we have $1 + r \ast r \leq r$. Hence star induction (10) shows $r \circledast \leq r$. The converse inequality $r \leq r \circledast$ holds by (KA).
   $(\Leftarrow)$ follows from (9).
7. By (KA), $a \leq a \circledast$. Moreover, $a \circledast$ is an invariant by Part 6 and (KA) again. Finally, if $r$ is an invariant with $a \leq r$ then $a \circledast \leq r \circledast = r$ by isotony of $\circledast$ and Part 6.

Next we discuss the lattice structure of the set $I(A)$ of invariants.
Theorem 10.7. Assume again a CKA A.

1. If A is a complete lattice, then so is \( I(A), \leq \). Its least and greatest elements are 1 and \( \top \), respectively.

2. For \( r, r' \in I(A) \) we have \( r \leq r' \iff r \ast r' = r' \). This means that \( \leq \) coincides with the natural order induced by the associative, commutative and idempotent operator \( \ast \) on \( I(A) \).

3. If \( r, r' \in I(A) \) have an infimum \( r \cap r' \) in A then this coincides with the infimum of \( r \) and \( r' \) in \( I(A) \).

4. \( r \ast r' \) is the supremum of \( r \) and \( r' \) in \( I(A) \). In particular, \( r \leq r'' \land r' \leq r'' \iff r \ast r' \leq r'' \).

5. Invariants are downward closed: \( r \ast r' \leq r'' \Rightarrow r \leq r'' \).

6. If A is a complete lattice then \( I(A) \) is even closed under arbitrary infima, i.e., for a subset \( U \subseteq I(A) \), the infimum \( \bigcap U \) taken in A coincides with the infimum of \( U \) in \( I(A) \).

Proof.

1. By Theorem \([10.6][6]\) the invariants are exactly the fixpoints of the \( \oplus \) operation. Since this operation is isotone, Tarski’s theorem shows the completeness claim. Leastness of 1 in \( I(A) \) is an axiom. Since \( \top \) is the greatest element, we have \( 1 \leq \top \) and \( \top \cdot \top \leq \top \) and hence \( \top \in I(A) \).

2. First, \( r \leq r' \Rightarrow r \ast r' \leq r' \ast r' \ast r' = r' \) by isotony and Theorem \([10.6][6]\). Second, by Theorem \([10.6][6]\) \( r \leq r \ast r' \) and hence \( r \ast r' = r' \) implies \( r \leq r' \).

3. First, \( 1 \leq r \) and \( 1 \leq r' \) imply \( 1 \leq r \cap r' \). Second, by isotony of \( \ast \) and Theorem \([10.6][6]\) \( (r \cap r') \ast (r \cap r') \leq r \ast r = r \). Likewise, \( (r \cap r') \ast (r \cap r') \leq r' \). Hence \( (r \cap r') \ast (r \cap r') \leq r \cap r' \). This shows that \( r \cap r' \) is in \( I(A) \) and therefore also the infimum of \( r \) and \( r' \) in \( I(A) \).

4. First, \( 1 = 1 \ast 1 \leq r \ast r' \) and \( (r \ast r') \ast (r \ast r') = r \ast r \ast r \ast r' \ast r' \leq r \ast r' \) show that \( r \ast r' \in I(A) \) as well. The supremum property is a well known fact about the natural order and hence follows from Part \([2]\).

The second assertion is straightforward from that and Part \([3]\).

5. Immediate from Part \([3]\).

6. By standard Kleene algebra, the operation \( \oplus \) is a closure operation. Hence, as shown e.g. in \([5]\) its set of fixpoints \( I(A) \) is closed under arbitrary infima. \( \square \)

Next we state two laws about iteration.

Lemma 10.8. Assume a CKA A and let \( r \in I(A) \) be an invariant and \( a \in A \) be arbitrary.

1. \( (r \ast a) \oplus \leq r \ast a \oplus \).

2. \( r \ast a \oplus = r \ast (r \ast a) \oplus \).

Proof.

1. We calculate

\[
(r \ast a) \oplus \leq r \ast a \oplus
\]

\[
\iff \quad \{ \text{star induction \([10]\)} \}
\]

\[
1 + (r \ast a) \ast (r \ast a) = r \ast a \oplus
\]

\[
\iff \quad \{ \text{join} \}
\]

\[
1 \leq r \ast a \oplus \land (r \ast a) \ast (r \ast a) \oplus \leq r \ast a \oplus .
\]

19
The first conjunct holds by \( 1 \leq r \) and \( 1 \leq a^\oplus \). For the second one we have, by \(*\)-idempotence of \( r \), the definition of star, isotony and associativity and commutativity of \(*\),
\[
r * a^\ominus = (r * r) * a^\ominus \geq (r * r) * (a * a^\ominus) = (r * a) * (r * a^\ominus) .
\]

2. By Part 1 isotony of \( * \) and idempotence of \( r \) we have
\[
r * (r * a)^\ominus \leq r * r * a^\ominus = r * a^\ominus .
\]

For the reverse inequation we first conclude \( a \leq r * a \) from Theorem 10.6.1 and then use isotony of \( ^\ominus \) and \(*\).

The above view of invariants is too special for some circumstances. Therefore we define a more liberal notion of invariants based on the fact that \( \mathbf{INV} \) is a closure and take Parts 4 and 5 of Theorem 10.3 as the characteristics of abstract invariants, since these properties suffice to prove the results about the rely/guarantee calculus in Section 14 we are after.

**Definition 10.9.** A concurrent semiring with invariants is a structure \((A,+,0,*,;\;\;\;1,t)\) such that \((A,+,0,*;\;\;\;1)\) is a concurrent semiring and \( t : A \rightarrow A \) is a closure operator that satisfies, for all \( a,b \in A \),
\[
1 \leq t a , \quad t (a * b) \leq t (a + b) .
\]

A closure invariant is an element \( a \in A \) with \( t a = a \).

**Lemma 10.10.** By the definition \( t a =_{df} a^\ominus \) every CKA becomes a concurrent semiring with invariants.

**Proof.** By standard Kleene algebra, \( ^\ominus \) is a closure operator with \( 1 \leq a^\ominus \). The remaining axiom is shown by star induction (10), \( a,b \leq a + b \) and isotony as follows:
\[
(a * b)^\ominus \leq (a + b)^\ominus \iff 1 + a * b * (a + b)^\ominus \leq (a + b)^* \iff 1 \leq (a + b)^\ominus \land (a + b) * (a + b)^\ominus \leq (a + b)^\ominus \iff \text{TRUE} .
\]

Again it is clear that the closure invariants form a complete lattice with properties analogous to those of Corollary 10.4 Moreover, one has the usual Galois connection for closures (cf. [12]):
\[
a \leq t b \iff t a \leq t b .
\]

With this definition we can give a uniform abstract proof of idempotence of operators on invariants.

**Theorem 10.11.** Let \( A \) be a concurrent semiring with invariants and \( \circ \) be an isotone binary operator on \( A \) that has \( 1 \) as neutral element and satisfies \( \forall a,b : t (a \circ b) \subseteq t (a + b) \). Then, for closure invariant \( r \), we have \( r \circ r = r \).

**Proof.** We first show \( r \circ r \leq r \). By extensivity of \( t \), the assumption and \( r + r = r \) as well as invariance of \( r \) we have \( r \circ r \subseteq t (r \circ r) \subseteq t r = r \). The converse inclusion is shown by \( r = r \circ 1 \leq r \circ r \), using neutrality of \( 1 \), the axiom \( 1 \leq t a \) and isotony of \( \circ \).

**Example 10.12.** Consider a concurrent semiring \( A \) with invariants. Setting \( \circ = ; \) we obtain by (3) and isotony of \( t \) that \( t (a ; b) \leq t (a + b) \). Since, in turn, \( t (a + b) \leq t (a + b) \) by Def. 10.9 Theorem 10.11 shows \( r ; r = r \) for all closure invariants \( r \).
11. Characterising Dependence

Invariants are of central importance for the rely/guarantee calculus in Sections 13 and 14. Their most fundamental property is the star distribution rule mentioned already in the introduction. We will now characterise the dependence relations for which this rule and another related one are valid.

**Theorem 11.1.** Let $R = \mathcal{P}(E)$ be a power invariant in $\mathcal{P}(\mathcal{E}V)$.

1. If $\rightarrow$ is acyclic and $e \in \mathcal{E}V$ then
   $$R \ast [e] \subseteq R; [e]; R,$$
   where $[e]$ is again the single-event program $\{\{e\}\}$ (cf. Section 3).

2. If $\rightarrow$ is transitive then for all $P,Q \in \mathcal{P}(\mathcal{E}V)$ we have
   $$R \ast (P; Q) \subseteq (R \ast P); (R \ast Q).$$

This means that the two properties of Theorem 11.1 hold if $\rightarrow$ is a strict-order.

To prove Theorem 11.1, we first show an auxiliary lemma about the dependence relation. To formulate it, we need an additional notion.

**Definition 11.2.** Remember the function $\text{dep}$ from Def. 7.1 that, for a trace $tp$ yields the set of events on which $tp$-events depend directly or indirectly. Given traces $tp$, $tr$ with $tp \cap tr = \emptyset$, we define

$$tr' = \{ df \, tr \cap \text{dep}(tp) \}, \quad tr'' = \{ df \, tr - \text{dep}(tp) \},$$

and call the pair $(tr', tr'')$ the dependence split of $tr$ w.r.t $tp$. Then $tr' \cup tr'' = tr$.

**Lemma 11.3.** Consider arbitrary traces $tp$ and $tq$.

1. The function $\text{dep}$ is $\subseteq$-isotone and hence subdistributive over intersection, i.e., it satisfies $\text{dep}(tp \cap tq) \subseteq \text{dep}(tp) \cap \text{dep}(tq)$.

2. $\text{dep}(\text{dep}(tp)) \subseteq \text{dep}(tp)$.

Let now $tp$ and $tr$ be traces with $tp \cap tr = \emptyset$, and let $(tr', tr'')$ be the dependence split of $tr$ w.r.t $tp$.

3. $\text{dep}(tr') \subseteq \text{dep}(tr) \cap \text{dep}(tp)$.

4. $tr'' \cap \text{dep}(tp) = \emptyset$.

5. For arbitrary trace $tq$ we have $tq \cap \text{dep}(tp) = \emptyset \Rightarrow tq \cap \text{dep}(tr') = \emptyset$.

6. $tp \cap \text{dep}(tp) = \emptyset \Rightarrow tp \cap \text{dep}(tr') = \emptyset$.

7. $tr'' \cap \text{dep}(tr') = \emptyset$.

8. Assume that $\rightarrow$ is acyclic and $tp = \{e\}$ for some event $e \in \mathcal{E}V$. Then $\{e\} \cap \text{dep}(tr') = \emptyset$ and hence

$$\{ tr \ast [e] = \{ tr' \}; [e]; \{ tr'' \}.$$

**Proof.**

1. As a general property, $\subseteq$-isotony is equivalent to subdistributivity over intersection.

2. Immediate from transitivity of $\rightarrow^+$.

3. By Parts 1 and 2

$$\text{dep}(tr') = \text{dep}(tr \cap \text{dep}(tp)) \subseteq \text{dep}(tr) \cap \text{dep}(\text{dep}(tp)) \subseteq \text{dep}(tr) \cap \text{dep}(tp).$$
4. Immediate from the definition of \( tr'' \) and Boolean algebra.

5. By Part 3 and the assumption about \( tq \),

\[
\begin{align*}
\text{tq} \cap \text{dep}(tr') & \subseteq \text{tq} \cap \text{dep}(tr) \cap \text{dep}(tp) = \emptyset .
\end{align*}
\]

6. Immediate from Parts 3 and 5.

7. Immediate from Parts 4 and 5.

8. This follows from the equivalence

\[
\{ e \} \cap \text{dep}(\{ e \}) = \emptyset \iff e \not\in \text{dep}(\{ e \}) \iff \neg (e \rightarrow^+ e ) .
\]

\[\Box\]

**Proof of Theorem 11.1**

We first note that power invariants \( R = \mathcal{P}(E) \) satisfy a stronger form of downward closure than the one stated in Theorem 10.7.5, namely \( tr \in R \land tr' \subseteq tr \Rightarrow tr' \in R \). In particular, the components of any dependence split of \( tr \) are in \( R \) again.

1. If \( e \in E \) then \( R * [e] = \emptyset \) and the claim holds trivially. Hence we calculate, assuming \( e \not\in E \):

\[
\begin{align*}
R * [e] & = \begin{cases}
\text{definition of * } & \\
\bigcup_{tr \in R} \{ tr * \{ e \} \} & \\
\subseteq \begin{cases}
\text{by Lemma 11.3.8 and downward closure of } R & \\
\bigcup_{tr' \in R} \bigcup_{tr'' \in R} \{ tr' ; \{ e \} ; tr'' \} & \\
\text{definition of ; } & \\
R ; [ e ] \in R . &
\end{cases}
\end{cases}
\end{align*}
\]

2. We show the property for singleton programs \( P = \{ tp \} \), \( Q = \{ tq \} \) with traces \( tp, tq \); then a similar calculation as for Part 3 extends it to arbitrary programs \( P, Q \). The property holds trivially if \( R * (P ; Q) = \emptyset \). Therefore assume \( R * (P ; Q) \neq \emptyset \) and consider an arbitrary trace \( tr \in R \) with \( \{ tr \} * (P ; Q) \neq \emptyset \). This implies that \( tp, tq, tr \) are pairwise disjoint and \( P ; Q \neq \emptyset \), hence \( \text{dep}(tp) \cap tq = \emptyset \). Moreover, \( ts = \text{df}_{\text{tr}} (tp ; tq) = tr \cup tp \cup tq \).

Let now \( (tr', tr'') \) be the dependence split of \( tr \) w.r.t. \( tp \). We show that then \( ts = (tr' * tp) ; (tr'' * tq) \) and hence \( ts \in (R * P) ; (R * Q) \).

(a) By Lemma 11.3.7 \( \text{dep}(tr') \cap tr'' = \emptyset \).

(b) By Lemma 11.3.5 \( \text{dep}(tr') \cap tq = \emptyset \).

(c) By Lemma 11.3.4 \( \text{dep}(tp) \cap tr'' = \emptyset \).

Now, by definition of \( tr', tr'', \) associativity and commutativity of union and (a),(b),(c) as well as \( \text{dep}(tp) \cap tq = \emptyset \) we have

\[
\begin{align*}
ts & = tr \cup tp \cup tq = tr' \cup tr'' \cup tp \cup tq = tr' \cup tp \cup tr'' \cup tq = (tr' * tp) ; (tr'' * tq) .
\end{align*}
\]

\[\Box\]

Next we want to see that in a sense also the reverse implication of Theorem 11.1 holds. To formulate this we need a further notion.
**Definition 11.4.** We call $\rightarrow$ weakly acyclic if for all events $e,f$,
\[ e \rightarrow f \rightarrow^+ e \Rightarrow f = e \, , \]
and weakly transitive if
\[ e \rightarrow f \rightarrow g \Rightarrow (e = g \lor e \rightarrow g) \, . \]

Weak acyclicity means that $\rightarrow$ may at most have immediate self-loops (which cannot be “detected” by the $;$ operator, since it is defined in terms of distinct events only).

**Theorem 11.5.** Let $[e]$ be again the single-event program $\{e\}$.
\[ \begin{align*}
1. & \text{ If } R \ast [e] \subseteq R ; [e] ; R \text{ is valid for all power invariants } R \text{ and events } e, \text{ then } \rightarrow \text{ is weakly acyclic.} \\
2. & \text{ If } R \ast (P ; Q) \subseteq (R \ast P) ; (R \ast Q) \text{ is valid for all power invariants } R \text{ and programs } P,Q \text{ then } \rightarrow \text{ is weakly transitive.}
\end{align*} \]

**Proof of Part 2.** Assume events $e,f,g$ with $f \rightarrow g$ and $g \rightarrow e$ but $f \not\rightarrow e$. This implies $f \neq g$ and $g \neq e$. Assume now $e \neq f$ and set $P =_{df} [e]$, $Q =_{df} [f]$ and $R =_{df} [\emptyset] \cup [g]$. Then $P ; Q = [e,f]$ and $R \ast (P ; Q) = [e,f] \cup [g,e,f]$. Moreover, $R \ast P = [e] \cup [g,e]$ and $R \ast Q = [f] \cup [g,f]$, hence $(R \ast P) ; (R \ast Q) = [e,f]$ contradicting the assumed property. Therefore we must have $e \leftarrow f$. \hfill $\blacksquare$

We abstract this as follows.

**Definition 11.6.** A concurrent semiring $A$ with invariants is $*$-distributive if all closure invariants $r$ and all $a,b \in A$ satisfy
\[ r \ast (a ; b) \leq (r \ast a) ; (r \ast b) \, . \]

We still have to prove Part [1] of Theorem [11.5]. Rather than doing this directly, we investigate a slightly more general property which is equivalent to an interesting property of traces that are more general than single-event ones.

**Definition 11.7.** A trace $tp$ is convex if for all events $e,f \in tp$ and arbitrary event $g$ we have
\[ e \rightarrow^+ g \rightarrow^+ f \Rightarrow g \in tp \, . \]

A convex trace can be considered as “closed” under dependence. Remember again the function dep from Def. [7.1]. Then we have

**Lemma 11.8.** Let $tp$ be a trace and assume that $R \ast \{tp\} \subseteq R ; \{tp\} ; R$ holds for all power invariants $R$.
\[ \begin{align*}
1. & \text{ Dependence between a trace and any event outside occurs at most in one direction, i.e., for any event } g \notin tp \text{ we have } \\
& tp \cap \operatorname{dep} \{g\} = \emptyset \lor \{g\} \cap \operatorname{dep}(tp) = \emptyset . \\
2. & \text{ As a consequence, } tp \text{ is convex.}
\end{align*} \]

**Proof.**
\[ \begin{align*}
1. & \text{ Set } R =_{df} \mathcal{P} \{\{f\}\} \text{. By assumption, the trace } tr = \{f\} \in R \text{ can be split as } tr = tr' ; tr'' \text{ such that } \\
& tr \ast tp = tr' ; tr'' \ast ; tp. \\
& \text{Case 1: } tr' = \{f\} \text{ and } tr'' = \emptyset. \text{ Hence } tr \ast tp = \{f\} ; tp. \text{ This implies } \{f\} \cap \operatorname{dep}(tp) = \emptyset. \\
& \text{Case 2: } tr' = \emptyset \text{ and } tr'' = \{f\}. \text{ Hence } tr \ast tp = tp ; \{f\}. \text{ This implies } tp \cap \operatorname{dep} \{\{f\}\} = \emptyset. \\
2. & \text{ Suppose } g \notin tp. \text{ The premise } e \rightarrow^+ g \text{ implies } e \in tp \cap \operatorname{dep} \{g\} \text{ while } g \rightarrow^+ f \text{ implies } g \in \{g\} \cap \operatorname{dep}(tp). \text{ In particular, both sets are non-empty, contradicting Part [1].} \hfill \blacksquare
\end{align*} \]
The case of singleton traces is covered as follows:

**Lemma 11.9.** All traces \( \{e\} \) are convex iff \( \rightarrow^+ \) is weakly acyclic.

**Proof.** (\( \Rightarrow \)) Assume \( e \rightarrow^+ f \rightarrow^+ e \). Then, by the assumed convexity of \( \{e\} \), we get \( f \in \{e\} \), i.e., \( f = e \).

(\( \Leftarrow \)) Assume \( g \rightarrow^+ f \rightarrow^+ h \) for \( g, h \in \{e\} \), i.e., \( e \rightarrow^+ f \rightarrow^+ e \). Then, by the assumed weak acyclicity, we obtain \( f = e \), i.e., \( f \in \{e\} \).

We now want to show that also the reverse of Lemma 11.8 holds.

**Lemma 11.10.** Let \( tp \) be convex. Then for all power invariants \( R \) the formula \( R * \{tp\} \subseteq R ; \{tp\} ; R \) is valid.

**Proof.** Consider some \( tr \in R \). We need to show \( \{tr\} * \{tp\} \subseteq R ; \{tp\} ; R \). The claim holds vacuously if \( tp \cap tr \neq \emptyset \). Hence assume that \( tp \cap tr = \emptyset \) and set

\[
tr' =_{df} tr \cap \text{dep}(tp), \quad tr'' =_{df} tr - \text{dep}(tp).
\]

In particular, \( tp \cap tr' = \emptyset \). From Lemma 6.3 of [22] we know

\[
tr'' \cap \text{dep}(tp) = tr'' \cap \text{dep}(tr') = \emptyset.
\]

If we can show that also \( tp \cap \text{dep}(tr') = \emptyset \) we have \( \{tr\} * \{tp\} = \{tr'\} ; \{tp\} ; \{tr''\} \) and are done. Therefore, suppose \( e \in tp \cap \text{dep}(tr') \), say \( e \rightarrow^+ g \) for some \( g \in tr' \). By definition of \( tr' \) there is an \( f \in tp \) with \( g \rightarrow^+ f \).

Since \( tp \) is assumed to be convex, this implies \( g \in tp \), a contradiction to \( g \in tr' \) and \( tp \cap tr' = \emptyset \). \( \square \)

Next, we consider general programs.

**Definition 11.11.** A program is convex if all its traces are.

**Lemma 11.12.** \( P \) is convex iff it satisfies for all power invariants \( R \)

\[
R * P \subseteq R ; P ; R.
\]

**Proof.** (\( \Rightarrow \)) Immediate from the definition and Lemma 11.10.

(\( \Leftarrow \)) Consider traces \( tp \in P \) and \( tr \in R \). We need to show \( \{tr\} * \{tp\} \subseteq R ; \{tp\} ; R \). The claim holds vacuously if \( tp \cap tr \neq \emptyset \). Hence let \( tp \cap tr = \emptyset \). By the assumption, there are traces \( tp' \in P \) and \( tr', tr'' \in tr \) with \( tp' \cap tr' = tp' \cap tr'' = tr' \cap tr'' = \emptyset \) and \( tr' \not\sim tp' \land tp' \not\sim tr'' \) such that \( tp \cup tr = tr' \cup tp' \cup tr'' \).

But, by disjointness, this implies \( tp' = tp \) and we are done.

These results motivate the following abstraction.

**Definition 11.13.** An element \( a \) of a concurrent semiring with invariants is called convex iff for all invariants \( r \) we have \( r * a \leq r ; a ; r \).

By \( b ; c \leq b * c \), commutativity of \( * \) and idempotence of invariants (Theorem 10.11) this inequation strengthens to an equality. This means that convex elements behave like “atoms” w.r.t. sequentialisation. Convexity will be important for one of the rules presented in the next section.

12. Rely/Guarantee Algebras

As before, we abstract the results of the previous section into general algebraic terms. The terminology stems from the applications in the following section.

**Definition 12.1.** A rely/guarantee semiring is a pair \( (A,I) \) such that \( A \) is a concurrent semiring with invariants and \( I \subseteq \ell(A) \) is a sublattice of closure invariants with \( 1 \in I \) and \( r * r' \in I \). In particular, for all \( r, r' \in I \) their meet \( r \cap r' \in I \) is assumed to exist. Moreover, all \( r \in I \) and \( a, b \in A \) have to satisfy \( r * (a ; b) \leq (r * a) ; (r * b) \).

A rely/guarantee CKA (quantale) is a rely/guarantee semiring that is a CKA (quantale).
The restriction that $I$ be a sublattice of $I(A)$ is motivated by the rely/guarantee-calculus in Section 13 below. Using Mace4 it can be shown that the axiomatisation is redundant.

Together with the exchange law (7), *-idempotence of $r$ and commutativity of * the definition implies
\[ r \ast (b \circ c) = (r \ast b) \circ (r \ast c) \]
\[(\ast\text{-distributivity)}\]
for all invariants $r \in I$ and operators $o \in \{\ast, ;\}$.

Using Theorem 11.1 we can prove

Lemma 12.2. Let $I = \{P(E) \mid E \subseteq EV\}$ be the set of all power invariants over $EV$. Then $(PR(EV), I)$ is a rely-guarantee semiring.

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

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

Example 12.3. Assume an event set $EV$ with three different events $e, f, g \in EV$ and dependences $e \rightarrow g \rightarrow f$. Set $P = \{e, f\}$. Then $P \ast P = \emptyset$ and hence $P^i = \emptyset$ for all $i > 1$. This means that the invariant $R = \{P^* = \text{skip} \cup P = \emptyset \cup [e, f]\}$ is not downward closed. Indeed, $\ast$-distributivity does not hold for it: we have $R \ast [r] = [r] \cup [e, f, g]$, but $R \ast [g] = [g]$.

The property of $\ast$-distributivity implies further iteration laws.

Lemma 12.4. Assume a rely/guarantee quantale $(A, I)$, an invariant $r \in I$ an arbitrary $a \in A$, and $\circ \in \{\ast, ;\}$.

1. $r \ast a^\circ = (r \ast a)^\circ \circ r = r \circ (r \ast a)^\circ$.
2. $(r \ast a)^+ = r \ast a^+$, where $a^+ = a \circ a^\circ$.

For the proof we use the following fusion rule for least fixpoints that is valid in quantales (cf. [2]). Let $f, g, h : A \rightarrow A$ be isotope functions. Then
\[ \forall x : f(g(x)) = h(f(x)) \]
\[ f(pg) = \mu h \]  \hspace{1cm} (12)

Proof of Lemma 12.4

1. For the first equation we use the fusion law (12) with the functions $f(x) = a \ast r \ast x$, $g(x) = 1 + a \circ x$ and $h(x) = a \ast r + (r \ast a) \circ x$. First, by the quantale assumptions, $f$ is strict and continuous. Second,
\[ f(g(x)) = \left\{ \begin{array}{l} \text{definitions} \\ r \ast (1 + a \circ x) \end{array} \right. \]
\[ = \left\{ \begin{array}{l} \text{distributivity of $\ast$ over +} \\ r \ast 1 + r \ast (a \circ x) \end{array} \right. \]
\[ = \left\{ \begin{array}{l} \text{neutrality of 1 and $\ast$-distributivity} \\ r + (r \ast a) \circ (r \ast x) \end{array} \right. \]
\[ = \left\{ \begin{array}{l} \text{definitions} \\ h(f(x)) \end{array} \right. . \]

For the equation $r \ast a^\ast = r \circ (r \ast a)^\ast$ we choose symmetrically $g'(x) = 1 + x \circ a$ and $h'(x) = x + (x \circ (r \ast a)).$

2. Analogously, with $g(x) = a + a \circ x$ and $h(x) = a \ast r + (r \ast a) \circ x$. \hfill \Box
13. Jones’s Rely/Guarantee-Calculus

In [25] Jones has presented a calculus that considers properties of the environment on which a program wants to rely and the ones it does, in turn, guarantee for the environment. We now provide an abstract algebraic treatment of this calculus.

The original motivation for discussing invariants was that they should allow guaranteeing that a program only uses events from a given admissible set. To this end we base our treatment on a concurrent bimonoid with invariants and define a guarantee relation, slightly more liberally than [20], by

\[ a \text{ guar } b \iff t a \leq t b , \]

meaning that \( a \) guarantees the closure invariant of \( b \). Since \( t \) as a closure is extensive, isotone and idempotent, the right hand side is equivalent to \( a \leq t b \). If \( b \) is an invariant, i.e., \( b = t b \), we obtain by (11)

\[ a \text{ guar } b \iff \iota a \leq \iota b , \]

Example 13.1. With the notation \( P_u = \text{df } [au] \) for \( u \in \{x, y, z\} \) of Example 2.3 we have \( P_u \text{ guar } G_u \) where \( G_u = \text{df } P_u \cup \text{skip } = [au] \cup [\] .

We have the following properties.

Theorem 13.2. Assume a rely/guarantee semiring \((A, I)\).

1. \( 1 \text{ guar } g \).

2. If \( g, g' \) are closure invariants and \( \circ \) is again an isotone binary operator satisfying \( \forall a, b : t (a \circ b) \leq t (a + b) \) then

\[ b \text{ guar } g \land b' \text{ guar } g' \Rightarrow (b \circ b') \text{ guar } (g + g') . \]

3. If \( A \) is a rely/guarantee CKA then for \( \circ \in \{*, ;\} \) we have \( a \text{ guar } g \iff a \circ \text{ guar } g \).

4. For the concrete case of programs, \([e] \text{ guar } G \iff e \in |G| .\)

Proof.

1. Immediate from the axioms and the above remark on guar.

2. \[ b \text{ guar } g \land b' \text{ guar } g' \]

\[ \iff \text{ above remark on guar } \]

\[ t b \leq g \land t b' \leq g' \]

\[ \Rightarrow \text{ isotony of + } \]

\[ t b + t b' \leq g + g' \]

\[ \Rightarrow \text{ subdistributivity of } t \]

\[ t (b + b') \leq g + g' \]

\[ \Rightarrow \text{ assumption about } \circ \]

\[ t (b \circ b') \leq g + g' \]

\[ \iff \text{ extensivity of } t \]

\[ t (b \circ b') \leq t (g + g') \]

\[ \iff \text{ definition } \]

\[ (b \circ b') \text{ guar } (g + g') . \]

3. Using the assumption, invariance of \( g \) and star induction, we calculate

\[ a \leq g \Rightarrow a \circ g \leq g \circ g = g \Rightarrow 1 + a \circ g \leq g \Rightarrow a^{\circ} \leq g . \]

The reverse implication follows by \( a \leq a^{\circ} . \)
4. By the definitions and the Galois connection for $| |$,

$$[e] \text{guar } G \iff \text{INV}([e]) \subseteq \text{INV}(G) \iff [e] \subseteq \text{INV}(G) \iff e \in |G|.$$  

Using the guarantee relation, Jones quintuples can be defined, as in [20], by

$$a \ r \{b\} \ s \ y \  \iff \ a \{r \,* \, b\} \ s \wedge b \ \text{guar } g \ ,$$

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 13.3.** Consider a rely/guarantee semiring $(A, I)$. For invariants $r, r', g, g' \in I$ and arbitrary $a, a', b, b', c, c' \in A$, 

$$a \ r \{b\} \ c \ g \wedge a' r' \{b'\} \ c' g' \wedge \ g \ \text{guar } r \wedge \ g \ \text{guar } r' \Rightarrow (a \cap a') \ (r \cap r') \{b \ast b'\} \ (c \cap c') (g \ast g') .$$

**Proof.** The guarantee part is covered by Theorem 13.2.2. For the remainder we note that the assumptions 

$$b' \ \text{guar } g' \ \text{guar } r$$

and $b \ \text{guar } g \ \text{guar } r'$ imply, by transitivity of guarantee, that $b' \ \text{guar } r \wedge b \ \text{guar } r'$, and calculate 

$$a \ r \{b\} \ c \ g \wedge a' r' \{b'\} \ c' g' \wedge \ g \ \text{guar } r \wedge \ g \ \text{guar } r' \Rightarrow (a \cap a') \ (r \cap r') \{b \ast b'\} \ (c \cap c') (g \ast g') .$$

Note that $r \cap r'$ and $g \ast g'$ are again invariants by Lemma 10.7.3 and 10.7.4. 

For sequential composition one has 

**Theorem 13.4.** Assume a rely/guarantee semiring $(A, I)$. Then for invariants $r, r', g, g' \in I$ and arbitrary 

$$a \ r \{b\} \ c \ g \wedge c \ r' \{b'\} \ c' g' \Rightarrow a \ (r \cap r') \{b ; b'\} \ c' \ (g \ast g')$$

**Proof.** The guarantee part is again covered by Theorem 13.2.2. Specialising $b, b', c, e$ in Lemma 9.3 to 

$$r \ast b, c, (r' \ast b'), (c; (r \ast r') \ast (b; b'))$$

respectively, we obtain that the weakest condition implying the remainder of the claim is 

$$(r \cap r') \ast (b ; b') \leq (r \ast b) ; (r' \ast b') .$$

Since Theorem 10.6.3 and the assumption on $I$ imply $r \cap r' \in I$ again, this follows by $\ast$-distributivity and isometry of $\ast$ and $;$. 

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

**Theorem 13.5.** Assume a rely/guarantee semiring $(A, I)$. Then for invariants $r, g \in I$ and arbitrary $s \in A$, 

1. $a \ r \{1\} \ s \ y \  \iff \ a \{r\} \ s$. 

27
We will show that \( a \{ b + b' \} s g \iff a \{ b \} s g \land a \{ b' \} s g. \)

3. Assume power invariants \( R = \mathcal{P}(E), G = \mathcal{P}(F) \) for \( E, F \subseteq E \) and let \( e \notin E \) and let \( \rightarrow \) be acyclic. Then \( P R \{ [e] \} S G \iff P \{ R; [e] ; R \} S \land [e] guar G. \)

**Proof.**

1. The guarantee part 1 \( guar \ g \) holds by the definition of invariants. For the remainder of the claim we have by the definition and neutrality of 1,
\[
a; (r + 1) \leq s \iff a; r \leq s \iff a \{ r \} s.
\]

2. By the definitions, distributivity and lattice algebra we have
\[
a \{ b + b' \} s g \iff a; (r \ast (b + b')) \leq s \land b + b' \leq g \iff a; (r \ast b) + a; (r \ast b') \leq s \land b \leq g \land b' \leq g \iff a \{ b \} s g \land a \{ b' \} s g.
\]

3. This is immediate from Theorem 13.11.

Finally we give rely/guarantee rules for iteration.

**Theorem 13.6.** Assume a rely/guarantee CKA \((A, I)\) and let \( \circ \) be finite iteration w.r.t. \( \circ \in \{*, ; \} \). Then for invariants \( r, g \in I \) and arbitrary elements \( a, b \in A \),
\[
a \{ r \} a \land a \{ r \} b g \iff a \{ r \} b g.
\]

**Proof.** The first law is immediate from Lemma 12.4.2 Lemma 9.2.8 and Theorem 13.3. The second one follows from the first one by \( b^* = 1 + b^+ \) and the choice and skip rules.

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

**Example 13.7.** We consider again the programs \( P_u = [au] \) and invariants \( G_u = P_u \cup \text{skip} \) \((u \in \{x, y\})\) from Example 13.1. Moreover, we assume an event \( \text{av} \) with \( v \neq x, y, ax \not\rightarrow \text{av} \) and \( ay \rightarrow \text{av} \) and set \( P_v = =df \{ av \}. \)

We will show that
\[
P_v \text{skip} \{ P_x \ast P_y \} [av, ax, ay] (G_x \ast G_y)
\]
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
\[
(a) \ P_x \text{guar} G_x \text{guar} R_y , \quad (b) \ P_y \text{guar} G_y \text{guar} R_x .
\]

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

From the definition of Hoare triples we calculate
\[
P_v \{ R_x \} ([av] \cup [av, ay]) \quad ([av] \cup [av, ay]) \{ P_x \} S_x \quad S_x \{ R_x \} S_x .
\]

since \([av, ax, ay] \ast [ay] = 0\). Combining the three clauses by Lemma 9.2.4 we obtain
\[
P_v \{ R_x ; P_x ; R_x \} S_x .
\]

By Theorem 13.3 we obtain \( P_v \ R_y \{ P_y \} S_x G_x \) and, similarly, \( P_v \ R_x \{ P_y \} S_y G_y \). Now the claim follows from the clauses (a),(b),(c),(d) and Theorem 13.3.

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.
14. A Simplified Rely/Guarantee-Calculus

For certain purposes, the following type of quadruples with an invariant $r$ works just as well as the Jones quintuples:

$$a r \{ b \} s \Leftrightarrow a \{ r \} s .$$

If information about the events of a program $b$ is needed (the rôle of $g$ in the original quintuples of the Jones calculus is, to a certain extent, to carry this information), one can use the smallest invariant containing $b$.

Note that the quadruples can be retrieved as special cases of quintuples:

$$a r \{ b \} s \Leftrightarrow a r \{ b \} s b .$$

We give the simplified versions of the original rely/guarantee-properties; the proofs result in a straightforward way from the ones above by the above embedding. Throughout this section we assume a rely/guarantee semiring $(A, I)$.

For concurrent composition we obtain

**Theorem 14.1.** For invariants $r, r'$,

$$a r \{ b \} s \land a' r' \{ b' \} s' \land b' \leq r \land b \leq r' \Rightarrow \left( (a \sqcap a') (r \sqcap r') \{ b \sqcup b' \} (s \sqcap s') .\right.$$   

Next we give rules for 1, union and convex programs.

**Theorem 14.3.**

1. $a r \{ 1 \} s \Leftrightarrow a \{ r \} s$.

2. $a r \{ b + b' \} s \Leftrightarrow a r \{ b \} s \land a r \{ b' \} s$.

3. If $b$ is convex then $a r \{ b \} s \Leftrightarrow a \{ r ; b ; r \} s$.

Part 3 has only been given for concrete single-event programs in [22]; therefore we give a quick proof for the abstract form here:

$$a r \{ b \} s \Leftrightarrow a ; (r * b) \subseteq s \Leftrightarrow a ; (r ; b ; r) \subseteq s \Leftrightarrow a \{ r ; b ; r \} s .$$

15. Event-Based Algebras

The definition of a concurrent semiring does not mention the dependence relation anymore. However, in the next section, when we establish a sufficient condition for protectedness, we shall need it, even at the level of single events. Therefore we now give algebraic characterisations of traces and events.

Throughout this section we assume a concurrent semiring $A$ with $1 \neq 0$. A subatom is an element $a$ such that $b \leq a \Rightarrow b = 0 \lor b = a$. A subatom different from $0$ is called an atom.
Definition 15.1. An element \( t \in A \) is called a \textit{trace} if it is a subatom and join-prime, i.e., if
\[
\forall a \in A : a \leq t \Rightarrow a = 0 \lor a = t ,
\]
\[
\forall T \subseteq A : T \neq \emptyset \land t \leq \sqcup T \Rightarrow \exists a \in T : t \leq a .
\]
The set of all traces is denoted by \( TR(A) \). For \( b \) in \( A \), the set of traces of \( b \) is
\[
TR(b) =_{df} \{ a \in TR(A) | a \leq b \} .
\]

By this definition, 0 is a trace. The traces different from 0 would be called atoms in lattice theory (cf. [5]). Admitting also 0 as a trace saves a number of case distinctions. It is immediate that every trace \( a \) is +-irreducible, i.e.,
\[
a = b + c \Rightarrow a = b \lor a = c .
\]
Moreover, if \( a \) is a trace and \( b \leq a \) then \( b \) is a trace, too. In particular, if \( a \ast b \) is a trace then by (3) also \( a ; b \) is a trace.

In our concrete model the abstract traces different from 0 correspond to singleton programs.

Definition 15.2. In a concurrent bimomoid \( A \) we define a relation \( \sqsubseteq \) by
\[
a \sqsubseteq b \iff_{df} \exists c : b = a \ast c .
\]

To investigate its properties we need

Definition 15.3. A subset \( E \subseteq A \) is \textit{well behaved} if the following conditions hold (for \( a, b, c \in E \)):
(a) \( 1 \in E \).
(b) \( E \ast E \subseteq E \).
(c) \( \ast \) is cancellative on \( E \), i.e., \( a \ast b \neq 0 \land a \ast b = a \ast c \Rightarrow b = c \).
(d) \( 1 \) is \( \ast \)-irreducible in \( E \), i.e., \( 1 = a \ast b \Rightarrow a = 1 \lor b = 1 \).

Lemma 15.4.

1. \( \sqsubseteq \) is a preorder, i.e., reflexive and transitive.

Assume now that \( E \subseteq A \) is well behaved. Then we have the following additional properties.
2. \( \sqsubseteq \) is antisymmetric on \( E \).
3. \( 1 \) is the \( \sqsubseteq \)-least element of \( E \).
4. If \( 0 \in E \) then it is the \( \sqsubseteq \)-greatest element of \( E \).

Proof.

1. Reflexivity follows by choosing \( c = 1 \) in the definition of \( \sqsubseteq \).
   For transitivity assume \( a \sqsubseteq b \) and \( b \sqsubseteq c \), say \( b = a \ast d \) and \( c = b \ast e \). Then \( c = (a \ast d) \ast e = a \ast (d \ast e) \).

2. Assume \( a \sqsubseteq b \) and \( b \sqsubseteq a \). If \( a = 0 \) then \( b = 0 \) follows from the definition of \( a \sqsubseteq b \), since 0 is an annihilator for \( \ast \). Otherwise suppose \( b = a \ast c \) and \( a = b \ast d \). Then \( a \ast 1 = a = b \ast d = a \ast c \ast d \), hence \( 1 = c \ast d \) by cancellativity. Now, irreducibility of 1 implies \( c = 1 \lor d = 1 \) and hence \( c = 1 = d \), showing \( a = b \).

3. and (4) are straightforward from the definition of \( \sqsubseteq \), neutrality of 1 and annihilation of 0. \( \square \)
In our concrete model, the set $E$ of singleton programs is well behaved and the relation $\sqsubseteq$ is isomorphic to the subset relation on concrete traces.

Assume now that $E$ is well behaved and hence $\sqsubseteq$ is a partial order on $E$. The supremum of a subset $D \subseteq E$ w.r.t. $\sqsubseteq$, if existent, is denoted by $\biguplus D$.

**Lemma 15.5.** If $0 \in D \subseteq E$ then $0 = \biguplus D$.

This is immediate from the definition of $\sqsubseteq$ and suprema.

**Definition 15.6.** Assume that $E$ is well behaved. Then $e \in E$ is called an $E$-event if it is subatomic and join-prime w.r.t. $\sqsubseteq$, i.e., if

\[
\forall d \in E : d \sqsubseteq e \Rightarrow d = 1 \lor d = e ,
\forall D \subseteq E : D \neq \emptyset \land \biguplus D \text{ exists} \Rightarrow (t \sqsubseteq \biguplus D \Rightarrow \exists d \in D : t \sqsubseteq d) .
\]

By this definition, 1 is an $E$-event, as is 0 if $0 \in E$. The $E$-events different from 0, 1 are atoms w.r.t. $\sqsubseteq$ in $E$. Clearly, every $E$-event $a$ is $\ast$-irreducible in $E$:

\[
a = b \ast c \Rightarrow b = a \lor c = a .
\]

To put things into perspective, we note that the order $\sqsubseteq$ corresponds to the well-known divisibility order on the natural numbers and $E$-events play the same rôle as the prime numbers.

**Definition 15.7.** A concurrent semiring $A$ is event-based if the following properties hold:

(a) 1 is a trace.

(b) Every element is the supremum of its traces, i.e., for all $a \in A$ we have $a = \bigcup TR(a)$.

(c) The set $TR(A)$ of traces is well behaved. By $EV(A)$ we denote the set of $TR(A)$-events and call them the events of $A$. The set of events of trace $t$ is

\[EV(t) = \text{df} \{ e \in EV(A) | e \sqsubseteq t \} .\]

(d) The set $TR(A)$ of traces is a complete lattice w.r.t. $\sqsubseteq$ and every trace is the supremum of its events, i.e., for all $t \in TR(A)$ we have $t = \biguplus EV(t)$.

(e) For all events $e$ we have $e \ast e = 0$ and hence $e ; e = 0$.

For an arbitrary $a \in A$ we then set $EV(a) = \text{df} \bigcup_{t \in TR(a)} EV(t)$.

Hence our concrete model of programs forms an event-based concurrent semiring. Event-based concurrent semirings are quite similar to the feature algebras developed in [24] for the description of product families.

The definition of an event-based concurrent semiring $A$ immediately yields

**Lemma 15.8.**

1. $EV(0) = EV(A)$.

2. $EV(1) = \{ 1 \}$.

3. For traces $a, b$ with $a \ast b \neq 0$ we have $EV(a \ast b) = EV(a) \cup EV(b)$ and hence $a \ast b = \biguplus \{ a, b \}$.

31
16. Abstract Dependence and Protection

In this section we define an abstract counterpart to the dependence relation and use it to give an intuitive sufficient criterion for protectedness. This requires an abstract formulation of the dependence relation.

**Definition 16.1.** We call element \( a \) **sequentially independent** of element \( b \), in signs \( a \not\leftarrow b \), if \( a \ast b \leq a \cdot b \).

The following properties are shown by straightforward calculation and, in the last case, by Theorem 10.11:

**Lemma 16.2.**

1. \( 0 \not\leftarrow a \) and \( a \not\leftarrow 0 \).
2. \( 1 \not\leftarrow a \) and \( a \not\leftarrow 1 \).
3. \( a \not\leftarrow c \land b \not\leftarrow c \Rightarrow (a + b) \not\leftarrow c \).
4. \( a \not\leftarrow b \land a \not\leftarrow c \Rightarrow a \not\leftarrow (b + c) \).
5. If \( r \) is an invariant then \( r \not\leftarrow r \).

Part 5 shows that for general programs this notion behaves in an unexpected way. However, in our concrete model it works fine for singleton programs:

\[
\{tp\} \not\leftarrow \{tq\} \iff \forall e \in tp, f \in tq : \neg(e \leftarrow f).
\]

In particular, \([e] \not\leftarrow [f] \iff \neg(e \leftarrow f)\). This motivates the following

**Definition 16.3.** In an event-based concurrent semiring we define the dependence relation between events \( e,f \) by

\[
e \rightarrow f \iff \neg(f \not\leftarrow e) \iff f ; e \neq e \ast f.
\]

We denote the converse of \( \rightarrow \) by \( \leftarrow \). We say that the algebra respects dependence if \( e \leftarrow f \Rightarrow e ; f = 0 \).

**Lemma 16.4.** Consider traces \( tp,tq \) of an event-based concurrent semiring that respects dependence.

1. If \( e \rightarrow f \) for some \( e \in EV(tp) \) and \( f \in EV(tq) \) then \( tp ; tq = 0 \).
2. If \( tp \ast tq \neq 0 \) then

\[
\text{tp} \not\leftarrow \text{tq} \iff \forall e \in EV(tp), f \in EV(tq) : e \not\leftarrow f.
\]

**Proof.**

1. By additivity of \( ; \) we have \( tp ; tq = \{u \cdot v \mid u \in EV(tp), v \in EV(tq)\} \) and the claim follows from Lemma 15.5.
2. (\( \Leftarrow \)) Immediate from event-basedness and additivity of \( \ast \) and \( ; \).

(\( \Rightarrow \)) By Part 1 we have \( e ; f \neq 0 \) for all \( e \in EV(tp) \) and \( f \in EV(tq) \). Since \( TR(A) \) is assumed to be well behaved, also \( e \ast f \) is a trace, and from \( e ; f \leq e \ast f \) it follows that \( e ; f = e \ast f \). \( \Box \)

With these prerequisites it is now possible to completely replay the proof of Theorem 11.1 in the abstract setting of event-based concurrent semirings; we omit the details.
17. 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 [4], CCS [31], CSP [17], mcRL2 [15] and the π-calculus [39].

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

<table>
<thead>
<tr>
<th>CKA operator</th>
<th>corresponding operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>non-deterministic choice in CSP</td>
</tr>
<tr>
<td>*</td>
<td>parallel composition</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>;</td>
<td>sequential composition ; in CSP and · in ACP</td>
</tr>
<tr>
<td>[]</td>
<td>choice + in CCS and internal choice □ in CSP</td>
</tr>
<tr>
<td>1</td>
<td>SKIP in CSP</td>
</tr>
<tr>
<td>0</td>
<td>this is the miracle and cannot be represented in any implementable calculus</td>
</tr>
</tbody>
</table>

However, there are a number of laws which show the inaccuracy of this table. For instance, in CSP we have SKIP ⊔ P = P, whereas CKA satisfies 1 ⊔ P = P. A similarly different behaviour arises in CCS, ACP and the π-calculus concerning distributivity of composition over choice.

As the discussion after Theorem 11.1 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 [14, 13, 37] 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 [40] 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 [7] present the notion of trioids, which are algebras offering the operators of choice, sequential and concurrent composition. However, there are no interrelating laws and no iteration. Chothia and Kleijn07 [8] 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) [36]. The main differences are the following. SKAs 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 [27]. 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 [50] quickly shows that this is not a consequence of the CKA axioms but holds only for the particular model.

18. 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 π-calculus in [20]. 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 CONCUR 09 and RelMiCS/AKA 09.

References

[29] S. Mac Lane: Categories for the working mathematician (2nd ed.). Springer 1998
Appendix A. Axiom Systems

For ease of reference we summarise the most important algebraic structures employed in the paper.

1. An idempotent semiring is a structure \((A, +, \cdot, 0, 1)\) such that \((A, +, 0)\) is a commutative monoid with idempotent addition, that is, \(a + a = a\) for all \(a \in A\), \((A, \cdot, 1)\) is a monoid, multiplication distributes over addition, that is, for all \(a, b, c \in A\),
\[
    a \cdot (b + c) = a \cdot b + a \cdot c \quad \text{and} \quad (a + b) \cdot c = a \cdot c + b \cdot c,
\]
and 0 is a left and right annihilator for multiplication, that is, for all \(a \in A\),
\[
    a \cdot 0 = 0 \cdot a = 0.
\]

2. Every idempotent semiring is partially ordered by
\[
    a \leq b \iff a + b = b.
\]
Then + and \(\cdot\) are isotone w.r.t. \(\leq\) and 0 is the least element. Moreover, \(a + b\) is the supremum of \(a, b \in A\).

3. A idempotent semiring is called a quantale \([34]\) or standard Kleene algebra \([10]\) if \(\leq\) induces a complete lattice and multiplication distributes over arbitrary suprema. The infimum and the supremum of a subset \(B \subseteq A\) are denoted by \(\sqcap B\) and \(\sqcup B\), respectively. Their binary variants are \(a \sqcap b\) and \(a \sqcup b\) (the latter coinciding with \(a + b\)).

4. An ordered monoid is a structure \((A, \cdot, 1, \leq)\) such that \((A, \cdot, 1)\) is a monoid, \(A\) is partially ordered by \(\leq\) and \(\cdot\) is isotone in both arguments.

5. A concurrent monoid is a structure \((A, *, ;, 1, \leq)\) such that \((A, *, \leq)\) and \((A, ;, \leq)\) are ordered monoids and the following axioms hold:
\[
    a * b = b * a, \quad (a * b) ; (c * d) \leq (a ; c) * (b ; d).
\]

6. A concurrent semiring is a structure \((A, +, *, ;, 0, 1)\) such that \((A, +, *, 0, 1)\) and \((A, +, ;, 0, 1)\) are idempotent semirings and \((A, *, ;, \leq)\) is a concurrent semigroup, where \(\leq\) is the natural semiring order.

7. A concurrent semiring \((A, +, *, ;, 0, 1)\) is called a concurrent quantale if \((A, +, *, 0, 1)\) and \((A, +, ;, 0, 1)\) are quantales.

8. A Kleene algebra \([26]\) is a structure \((A, +, *, *, 0, 1)\) such that \((A, +, *, 0, 1)\) is an idempotent semiring and the star operator \(*\) satisfies the unfold and induction laws
\[
    1 + a \cdot a^* \leq a^*, \quad 1 + a^* \cdot a \leq a^*, \quad (A.1)
\]
\[
    c + a \cdot b \leq b \Rightarrow a^* \cdot c \leq b, \quad c + b \cdot a \leq b \Rightarrow c \cdot a^* \leq b. \quad (A.2)
\]
9. A concurrent Kleene algebra (CKA) is a structure \((A, +, *, ;, \bigcirc, 0, 1)\) such that \((A, +, *, ;, 0, 1)\) is a concurrent semiring and \((A, +, *, \bigcirc, 0, 1)\) and \((A, +, *, 0, 1)\) are Kleene algebras.

10. An invariant in a concurrent semiring \(A\) is an element \(r\) satisfying \(1 \leq r\) and \(r * r \leq r\), equivalently, \(1 + r * r \leq r\). The set of all invariants of \(A\) is denoted by \(I(A)\).

11. A concurrent semiring with invariants is a structure \((A, +, 0, *, ;, 1, t)\) such that \((A, +, 0, *, ;, 1)\) is a concurrent semiring and \(t : A \rightarrow A\) is a closure operator that satisfies, for all \(a, b \in A\),

\[
1 \leq t a , \quad t (a * b) \leq t (a + b) .
\]

A closure invariant is an element \(a \in A\) with \(t a = a\).

12. A concurrent semiring \(A\) with invariants is \(*\)-distributive if all closure invariants \(r\) and all \(a, b \in A\) satisfy

\[
r * (a ; b) \leq (r * a) ; (r * b) .
\]

13. A rely/guarantee semiring is a pair \((A, I)\) such that \(A\) is a concurrent semiring with invariants and \(I \subseteq t(A)\) is a sublattice of closure invariants with \(1 \in I\) and \(r * r' \in I\). In particular, for all \(r, r' \in I\) their meet \(r \cap r' \in I\) is assumed to exist. Moreover, all \(r \in I\) and \(a, b \in A\) have to satisfy \(r * (a ; b) \leq (r * a) ; (r * b)\).

A rely/guarantee CKA (quantale) is a rely/guarantee semiring that is a CKA (quantale).

Appendix B. Input Files for Automated Theorem Proving

Appendix B.1. Introduction

Prover9 is an automated theorem proving system (ATP system). It is complete for first-order logic with equality: Given a valid statement in that language, it should in principle return with a proof, though, in practice, it will often fail due to time and memory resources. Given a statement that is not valid, it will usually run forever, and terminate with failure only on the most trivial inputs.

Prover9 is based on the superposition calculus \([6]\). It uses sophisticated term orderings and redundancy elimination mechanisms to control the search space, and employs rewriting techniques to make equational reasoning more efficient. We use the tool as a black box with its default settings; its internals are of no relevance to us.

Prover9 is complemented by Mace4, a counterexample generator which by and large accepts the same input syntax. The combination of these tools support a game of proof and refutation which is essential for axiomatic developments and theory engineering (cf. \([1]\)).

Our main reasons for choosing Prover9 for verifying proofs in concurrent Kleene algebras are its readable input and output syntax, its good performance on algebraic structures \([5]\), and its integration with Mace4. Some other provers, in particular Vampire \([8]\) and E \([7]\) might have yielded comparable success rates. Experiments show that state of the art off-the-shelf ATP systems are able to automate calculational proofs that eminent mathematicians would have found worth publishing some decades ago, for instance in the field of Tarski’s relation algebras \([3, 4]\).

The main purpose of this appendix is to document to which extent the proofs in this paper can be automated. Statements that are not prima facie calculational are not considered. Our experiments show that all first-order calculational proofs using concurrent Kleene algebras could be automated, including the correctness proofs of the rules of the Hoare calculus, and most of the rules of the rely/guarantee calculus. But while the ATP system performed consistently well on these algebraic proofs, an automation of the proofs based on predicate logic in the first sections of this report was much less successful, in particular when specifications used sets or contained some existential quantifiers. The fact that a large number of variables and existential quantifiers can have a negative effect on proof search is not too surprising. The contrast between the performance in predicate-logical and algebraic reasoning nicely demonstrates the benefits of the algebraic approach.

All proofs were carried out with a 2.4 GHz Intel Core 2 Duo laptop under Mac OS X.
Appendix B.2. Aggregation and Independence

We first prove Lemma 3.4. The input file for Prover9 is as follows.

```
op(500, infix, "+").
formulas(assumptions). % aggregation algebra
  % x+(y+z)=(x+y)+z.
  % x+0=x.
  % 1+0=x.
% independence relation
  R(x+y,z) <-> R(x,z) & R(y,z).
  R(x,y+z) <-> R(x,y) & R(x,z).
% R(0,x).
% R(x,0).
end_of_list.
```

The first block declares that + is an infix operator. The second block contains the assumptions. In its first part, the aggregation algebra is defined as a monoid. Note that all axioms have been commented out; hence we work with the absolutely free aggregation algebra. In the second part of the assumptions block, the independence relation $R$ is defined. The strictness axioms have again been commented out.

The third block of the file contains the lemma to be verified — by calling Prover9 — or refuted — by calling Mace4. Prover9 can verify more than one equation or positive goal at a time, but not more than one non-equational or non-positive goal. All goals but the first one have therefore again been commented out.

By calling Prover9, the statement in Lemma 3.4.1 is automatically verified, and the following output proof is generated.

```
% Proof 1 at 1.67 (+ 0.07) seconds.
% Length of proof is 23.
% Level of proof is 9.
% Maximum clause weight is 14.
% Given clauses 674.
1 R(x + y,z) <-> R(x,z) & R(y,z) # label(non_clause). [assumption].
3 R((w + x) + y,z) <-> R(w + (x + y),z) # label(non_clause) # label(goal). [goal].
4 -R(x + y,z) | R(x,z). [clausify(1)].
5 -R(x + y,z) | R(y,z). [clausify(1)].
6 R(x + y,z) | -R(x,z) | -R(y,z). [clausify(1)].
12 R((c1 + c2) + c3,c4) | R(c1 + (c2 + c3),c4). [deny(3)].
13 -R((c1 + c2) + c3,c4) | -R(c1 + (c2 + c3),c4). [deny(3)].
24 R(c1 + (c2 + c3),c4) | R(c3,c4). [resolve(12,a,5,a)].
25 R(c1 + (c2 + c3),c4) | R(c1 + c2,c4). [resolve(12,a,4,a)].
148 R(c3,c4) | R(c2 + c3,c4). [resolve(24,a,5,a)].
180 R(c3,c4). [resolve(148,b,5,a),merge(b)].
191 R(x + c3,c4) | -R(x,c4). [resolve(180,a,6,c)].
223 R(c1 + c2,c4) | R(c2 + c3,c4). [resolve(25,a,5,a)].
224 R(c1 + c2,c4) | R(c1,c4). [resolve(25,a,4,a)].
1124 R(c1,c4). [resolve(224,a,4,a),merge(b)].
1144 R(c1 + c4) | -R(x,c4). [resolve(1124,a,6,b)].
17189 R(c1 + c2,c4) | R(c2,c4). [resolve(223,b,4,a)].
17184 R(c2,c4). [resolve(17159,a,5,a),merge(b)].
17185 R(c1 + c2,c4). [resolve(17184,a,1144,b)].
17188 R(c2 + c3,c4). [resolve(17184,a,191,b)].
```

37
This proof uses the rules of the superposition calculus to derive the empty clause $\neg F$ from the negated goal in line (12) and (13). It is not intended to be readable by humans. The other proofs for Lemma 3.4 can also be generated almost instantaneously by Prover9.

The input file for Lemma 3.5 is as follows.

Both goals could be verified instantaneously. The input file for Lemma 3.6 is as follows.

Again, the goal could be verified instantaneously.

Appendix B.3. Algebraisation of the Calculus

Next, we attempted to prove Proposition 4.3. Since Prover9 cannot express set comprehension, we used set extensionality, encoding the epsilon relation and set equality as binary predicates, to prove associativity of the complex product which is encoded as ; below.
However, we did not succeed to prove any of the goals within reasonable time (for the second goal, extensionality can be commented out). In general, it turned out that these mixed algebraic and predicate-logical proofs are rather difficult for Prover9. One reason certainly are the intricate unifications modulo associativity and commutativity that lead to an explosion of the search space.

But still, the proof took about one minute and it has 41 steps. The proof of the second monoid law was similar.

Appendix B.4. Diods and Quantales

The difficulties with mixed algebraic and predicate logical reasoning persisted when attempting to prove Proposition 5.3. We encoded set union as

\[ \text{in}(x, \text{union}(y, z)) \leftrightarrow (\text{in}(x, y) \lor \text{in}(x, z)) \]
and proved properties such as

\[
\text{seq}(x,y) \rightarrow \text{seq}(\text{union}(x,z),\text{union}(y,z)). \quad \% \text{ cong}
\]
\[
\text{seq}(\text{union}(x,x),x). \quad \% \text{ I}
\]
\[
\text{seq}(\text{union}(x,y),\text{union}(y,x)). \quad \% \text{ C}
\]
\[
\text{seq}(\text{union}(\text{union}(x,y),z),\text{union}(x,\text{union}(y,z))). \quad \% \text{ A}
\]

\[\text{end_of_list.}\]

We could not prove \(A\) in one go within reasonable time. We could prove

\[\text{all a } ((\text{el}(a,\text{union}(\text{union}(x,y),z))) \leftrightarrow (\text{el}(a,\text{union}(x,\text{union}(y,z))))).\]

and then

\[\text{seq}(\text{union}(\text{union}(x,y),z),\text{union}(x,\text{union}(y,z)))) \rightarrow \text{all a } ((\text{el}(a,\text{union}(\text{union}(x,y),z))) \rightarrow (\text{el}(a,\text{union}(x,\text{union}(y,z))))).\]

Due to these complications, we did not further pursue the automated verification of Proposition 5.3, the manual proof of which is entirely routine, and turned to the algebraic level where automated theorem proving is much more successful.

A general lesson to be learned is that the algebraic or calculational approach is particularly suitable for automated reasoning, and often superior to encodings in predicate logic.

\section*{Appendix B.5. Concurrent Algebras}

Disregarding again the mixed proofs of Proposition 6.2 to Proposition 6.5, we proved Lemma 6.7 automatically, using the following input file for \texttt{Prover9}.

\begin{verbatim}
formulas(goals).
 seq(x,y) -> seq(union(x,z),union(y,z)). \% cong
 seq(union(x,x),x). \% I
 seq(union(x,y),union(y,x)). \% C
 seq(union(union(x,y),z),union(x,union(y,z))). \% A
 end_of_list.

We could not prove A in one go within reasonable time. We could prove

\[\text{all a } ((\text{el}(a,\text{union}(\text{union}(x,y),z))) \leftrightarrow (\text{el}(a,\text{union}(x,\text{union}(y,z))))).\]

and then

\[\text{seq}(\text{union}(\text{union}(x,y),z),\text{union}(x,\text{union}(y,z)))) \rightarrow \text{all a } ((\text{el}(a,\text{union}(\text{union}(x,y),z))) \rightarrow (\text{el}(a,\text{union}(x,\text{union}(y,z))))).\]

Due to these complications, we did not further pursue the automated verification of Proposition 5.3, the manual proof of which is entirely routine, and turned to the algebraic level where automated theorem proving is much more successful.

A general lesson to be learned is that the algebraic or calculational approach is particularly suitable for automated reasoning, and often superior to encodings in predicate logic.

$$\text{Appendix B.5. Concurrent Algebras}$$

Disregarding again the mixed proofs of Proposition 6.2 to Proposition 6.5, we proved Lemma 6.7 automatically, using the following input file for \texttt{Prover9}.

\begin{verbatim}
op(450, infix, "*").
op(450, infix, ";").
formulas(assumptions). \% concurrent semigroup
 x*(y*z)=(x*y)*z.
 x*y=x*y.
 x<=x.
 x<y & y<x <-> x<y.
 x<y & y<=z -> x<y.
 x<y -> z;x<=z;y.
 x<y -> x*z<=y*z.
 x<y -> y*z<=x*z.
 x;y<=x;y.
 x;(y;z)<=x<=(x;y)*z.
 (x+y);z<=x*(y;z).
 \% (w*x);(y*z)<= (w;y)*(x;z).
end_of_list.

formulas(goals). \% redundancy
 \% x;(y*z)<= (x;y)*z. \% 4-element counterexample
 \% (x+y);z<=x*(y;z). \% 4-element counterexample
 \% (w*x);(y*z)<= (w;y)*(x;z). \% 2-element counterexample
end_of_list.

To prove redundancy, one axiom needs to be commented out in the assumptions list and all but that formula in the goals list. The axiom in the goal is then irredundant with respect to the other axioms in the assumption list if there is a counterexample; a model that makes the axioms in the assumption list true and the goal axiom false. We invoked \texttt{Mace4} to find this counterexample. The comments after the goal axioms indicate the size of the counterexample found. The proofs for concurrent semirings were similar.

40
We now pass from bisemigroups to bimonoids and prove Lemma 6.8 from the following input file. We can show that \((w * x); (y * z) \leq (w;y) * (x;z)\) implies \(x;(y*z) \leq (x;y)*z\) and \((x*y);z \leq x*(y;z)\), but not vice versa (2-element model).

Appendix B.6. Generalised Sequential and Concurrent Composition

We then proved Lemma 7.2 using an axiomatisation for bounded distributive lattices. It is worth noting that we now used the symbol \(\ast\) for lattice meet. This is not optimal, but with Prover9 the available infix symbols are constraint by what can be found on a keyboard and the prover's predefined syntax.
op(500, infix, "+"). % join
op(500, infix, "+"). % meet

formulas(assumptions). % distributive lattice with zero
x+y=y+x.
x+(y+z)=(x+y)+z.
x+x=x.
x+0=x.
x*(y+z)=(x*y)+(x*z).
x*(y+z)=(x*y)*x+z).
f(x+y)=f(x)+f(y). % strict additive operator
f(0)=0.

D(x,y) <-> x*y=0. % fine-grain concurrent composition
E(x,y) <-> D(x,y) & f(x)*y=0. % weak sequential composition
F(x,y) <-> E(x,y) & x*f(y)=0. % disjoint composition
G(x,y) <-> (x=0 | y=0). % alternation

Some of the bilinearity proofs ran for more than a minute.

Appendix B.7. Hoare Calculus

We then proved Lemma 9.2. For the statements based on ordered monoids we used the following input file. Encoding Hoare triples by predicates, we could specify our goals in declarative style similar to logic programming.

op(450, infix, ";").

formulas(assumptions). % ordered monoid
x;y;z=(x;y)*z.
1;x=x.
x;1=x.
x<y.

42
\[ x \leq y \implies y \leq x \iff x = y. \]
\[ x \leq y \implies y \leq z \implies x \leq z. \]
\[ x \leq y \implies z; x \leq z; y. \]
\[ x \leq y \implies x; z \leq y; z. \]

\% Hoare triple
\[ H(x,y,z) \iff x; y \leq z. \]
\end_of_list.

\textbf{formulas(goals).}
\% \[ H(x,1,y) \iff x \leq y. \]
\% \[ H(x,1,x). \]
\% all a (all b (H(a,x,b) \implies H(a,y,b))) \iff y \leq x.
\% all a (all b (H(a,x,b) \implies H(a,y,b))) \iff x = y.
\% all a (all b (H(a,x,y,b) \iff \exists d (H(a,x,d) \& H(d,y,b))))
\% all a (all b (all c all d(a \leq b \& H(b,x,c) \& c \leq d \implies H(a,x,d))))
\end_of_list.

The remaining statements in Lemma 9.2 could be proved from the following input file, commenting out the star axioms for the first two statements. Here again, the star operation is overloaded and used for the Kleene star.

\textbf{op(500, infix, "+").}
\textbf{op(400, postfix, "*").}
\textbf{op(450, infix, ";").}
\textbf{formulas(assumptions). % dioid/Kleene algebra}
\[ x+(y+z)=(x+y)+z. \]
\[ x+y=y+x. \]
\[ x+x=x. \]
\[ x+0=x. \]
\[ x;(y;z)=(x;y);z. \]
\[ 1;x=x. \]
\[ x;1=x. \]
\[ x:(y;z)=x;y+x;z. \]
\[ (x+y);z=x;z+y;z. \]
\[ x+0=0. \]
\[ 0;x=0. \]
\[ x+y \iff x+y=y. \]
\[ 1x;xx \iff x=x. \]
\[ 1x;x=x. \]
\[ z+x;y \leq y \implies z;x \leq y. \]
\[ z+y;x \leq z; x \leq y. \]
\% Hoare triple
\[ H(x,y,z) \iff x;y \leq z. \]
\end_of_list.

\textbf{formulas(goals).}
\% \[ H(x,0,y). \]
\% \[ H(u,x+y,z) \iff H(w,x,z) \& H(u,y,z). \]
\% \[ H(x,y,z) \iff H(x,y; y; x). \]
\% \[ H(x,y,x) \iff H(x,y; y; z). \]
\% \[ H(x,y,z) \iff H(x,y; y; x). \]
\end_of_list.

To reduce running times we splitted the last two equivalences into implications. Lemma 9.3 could also be proved instantaneously using the assumption file for ordered monoids from Lemma 9.2.

\textbf{op(450, infix, ";").}
\textbf{formulas(assumptions). % ordered monoid}
\% same as for Lemma 9.2
\end_of_list.

43
formulas(goals).
  (all a (all b (all c (H(a,x,c) & H(c,y,b) -> H(a,z,b))))) <-> z<= x;y.
end_of_list.

And here is the input file for Lemma 9.4 for concurrent semigroups.

```plaintext
op(450, infix, "*").
op(450, infix, ";").
formulas(assumptions). % concurrent semigroup
  x*(y*z)=(x*y)*z.
x*y=x*z.
x<=x.
x<y & y<x <-> x=y.
x<y -> z;x<=z;y.
x<y -> x;z<=y;z.
x<=y -> x*z<=y*z.
x;0=x.
x;1=x.
x;0=0.
end_of_list.

formulas(goals).
  % all a (all b (all c (all d (H(a,x,b) & H(c,y,d) -> H(a*c,x*y,b*d)))).
  % all d(H(x,y,z) -> H(d*x,y,d*z)).
end_of_list.
```

Appendix B.8. Invariants

We next gave an automated proof of Theorem 10.6. We first proved 10.6.6 for arbitrary Kleene algebras, namely that \( x \) is an invariant if and only if \( x = x^* \). This proof took about 240s.

```plaintext
op(500, infix, "+").
op(490, infix, ";").
op(480, postfix, "*").
formulas(assumptions). %Kleene algebra
  x+(y+z)=(x+y)+z.
x+y=y+x.
x+0=x.
x+x=x.
x;1=x.
1;x=1.
0;x=0.
end_of_list.

formulas(goals).
  1+x;x<=x <-> x*=x.
end_of_list.
```

We then proved a variant of Theorem 10.6 using \( x = x^* \) as the characterisation of invariants. The assumption list is also our reference input file for concurrent semirings and concurrent Kleene algebras.
Now to Theorem 10.7.1. We provided an alternative proof to that in the paper, which uses results about
Kleene algebras that have already been obtained [2]. We have already proved automatically that $x^*x^* = x^*$
holds in KA, hence invariants are idempotent. Of course, by definition, they are associative and commutative
as well, when $*$ is used for multiplication. We have also already proved automatically, for all Kleene algebras,
the following Church-Rosser property:

$$yx \leq xy \Rightarrow (x + y)^* = x^*y^*.$$ 

Finally, we have shown that $(x + y)^* = (x^*y^*)^*$. This implies that

$$(x + y)^* = (x^*y^*)^* = x^*y^*,$$

holds when multiplication is commutative, hence invariants are closed under multiplication $*$. This proves
that the set of invariants forms a sub-semilattice of a concurrent Kleene algebra. Since this is straightforward,
we didn’t automate the proof. If the semilattice is complete it is automatically a complete lattice.

We then proved Theorem 10.7.2:
The properties of Theorem 10.7.3 and Theorem 10.7.4 follow from the above proofs. We therefore proved 10.7.5 next from the same assumptions:

\begin{verbatim}
formulas(goals).
\% si(x)<=si(y) -> si(z)*si(x)=si(y).
\end_of_list.
\end{verbatim}

10.7.6 is beyond first-order logic.

For Lemma 10.8 we first proved an auxiliary lemma which does not appear in the paper, but which is a variant of Lemma 12.4.2:

\begin{verbatim}
(x+y)*=(x*y)*;x*.
\end_of_list.
\end{verbatim}

Using this, we could prove, for arbitrary commutative Kleene algebras, that

\begin{equation}
(x*y)^+ = (x^*y)^+ = (x+y)^*y = x^*y^*y = x^*y^+.
\end{equation}

\begin{verbatim}
formulas(goals).
\% lemma 12.4.2
(x*y)*;(x*y)=1+(x*y*);y.
\end_of_list.
\end{verbatim}

As a corollary we obtained Lemma 10.8.1, namely that

\begin{equation}
(x^*y)^* = 1 + x^*y^*y.
\end{equation}

Using this we could then easily prove Lemma 10.8.2 for arbitrary commutative Kleene algebras:
For Lemma 10.10 there is again nothing to prove since \((xy)^* \leq (x + y)^*\) holds in every Kleene algebra, which has already been automatically verified.

Next we proved Theorem 10.11 for an additive semilattice, a new operation \(\;\) and a closure invariant, where the axiom for \(\ast\) is ignored. The proof was instantaneous.

Appendix B.9. Rely/Guarantee Algebras

We first showed that the \(\ast\)-distributivity axiom from Definition 12.1 is irreduntant in CKA.

Mace4 found a 5-element counterexample. We then verified the equational variant of that law in concurrent semirings.
We then considered Lemma 12.4, the proof of which is based on quantales and fixpoint fusion. We could only prove inequalities, but failed with the equations. Perhaps they are beyond concurrent Kleene algebras, but we also couldn’t find finite counterexamples.

Appendix B.10. Jones’s Rely/Guarantee Calculus

We then proved Theorem 13.2 except for case 4, which is not algebraic.
We then proved Theorem 13.5.2. Interestingly, properties of invariants are not needed in that proof; hence the theorem holds already in concurrent Kleene algebras.

Here is the output file generated by Prover9:

% Proof 1 at 19.85 (+ 0.90) seconds.
% Length of proof is 52.
% Level of proof is 9.
% Maximum clause weight is 20.
% Given clauses 4798.

1 x <= y <-> x + y = y # label(non_clause). [assumption].
2 x + y <= z <-> x <= z & y <= z # label(non_clause). [assumption].
3 J(v,w,x,y,z) <-> v ; (w * x) <= y & x <= i(z) # label(non_clause). [assumption].
4 J(v,w,x1 + x2,y,z) <-> J(v,w,x1,y,z) & J(v,w,x2,y,z) # label(goal). [goal].

49
We then considered Theorem 13.6., but failed again. This is not too surprising since the manual proof relies on fixpoint fusion, which is beyond first order logic.

References


