A Discrete Geometric Model of Concurrent Program Execution

Bernhard Möller[1], Tony Hoare[2], Martin E. Müller[1], and Georg Struth[3]

[1] Institut für Informatik, Universität Augsburg, Germany
[2] Microsoft Research, Cambridge, UK
[3] Department of Computer Science, The University of Sheffield, UK

Abstract. A trace of the execution of a concurrent object-oriented program can be displayed in two-dimensions as a diagram of a non-metric finite geometry. The actions of a programs are represented by points, its objects and threads by vertical lines, its transactions by horizontal lines, its communications and resource sharing by sloping arrows, and its partial traces by rectangular figures.
We prove informally that the geometry satisfies the laws of Concurrent Kleene Algebra (CKA); these describe and justify the interleaved implementation of multithreaded programs on computer systems with a lesser number of concurrent processors. More familiar forms of semantics (e.g., verification-oriented and operational) can be derived from CKA.
Programs are represented as sets of all their possible traces of execution, and non-determinism is introduced as union of these sets. The geometry is extended to multiple levels of abstraction and granularity; a method call at a higher level can be modelled by a specification of the method body, which is implemented at a lower level.
The final section describes how the axioms and definitions of the geometry have been encoded in the interactive proof tool Isabelle, and reports on progress towards automatic checking of the proofs in the paper.

Keywords: Concurrent Kleene Algebra, Laws of Programming, Trace Algebra, Semantic Models, Refinement, Unifying Theories

1 Introduction

The intent of this paper is to make a modest but seminal contribution towards an ambitious long-term goal. The goal is to provide a secure conceptual foundation for the design, implementation and effective use of future program debugging tools. They will assist in unit testing, component integration, and evolution of concurrent and distributed systems software on an enterprise scale. Such tools will provide differential analysis of changed code, generation of effective test cases, run-time detection of errors, and assistance in their location, diagnosis and correction. The errors will include generic errors defined by the programming language (e.g., overflows), violation of properties explicitly defined as assertions

or assumptions in the program, as well as violations of behavioural design patterns originally laid down by the system architect. The tools will communicate with the programming teams by displaying a navigable trace of events leading up to the suspected anomalies – a technology known as "time-travel debugging".

Our modest contribution is to formalise a discrete geometry governing diagrams of program behaviour. The diagrams will include actions of the program that are relevant to an anomaly, as well as communications and other causal dependencies between the actions.

We provide an example of the application of the geometry to a concurrent object-oriented program. The set of all possible traces of execution of a particular program is a mathematical formalisation (model) of its meaning. Technically, it is known as a denotational semantics. We prove that this semantics satisfies the star-free laws of a Concurrent Kleene Algebra (CKA); this gives an algebraic semantics that justifies program transformation rules applied in optimisation. From the algebraic semantics it is possible to derive other familiar and widely applied forms of semantics (e.g., operational and verification-oriented). We offer this as evidence of the potential applicability of geometry to current and future programming practice.

Further evidence is provided by quoting the many sources of ideas that have been amalgamated into our theories. Our geometric foundation is inspired by graphical research tools developed and applied to the analysis of relaxed memory models, [26,1]. The pattern of horizontal and vertical lines in our diagrams is taken from Message Sequence Charts (MSC) [11] which are widely used to plan and record the architecture of a large-scale computer application. Our concept of a transaction matches the transition of a Petri Net, [30]. Our assertion language for specification of traces is Concurrent Separation Logic [9,29], widely used by seekers of proofs for concurrent programs. Finally, our motivation and methodology are those of past and current research into Unifying Theories of Programming [20,22].

Summary

In section 2 the primitive concepts of our geometry are enumerated as points, lines and figures, drawn on a two-dimensional surface. The vertical dimension represents time, the horizontal one space. Actions of a program are represented by points, objects by vertical lines, and transactions by horizontal lines. Points occur only at the intersection of a vertical with a horizontal line. Arrows are defined as segments of lines between two neighbouring points on a line. A figure contains a subset of points, and its perimeter is the set of arrows which connect its internal points to points in its external environment.

A figure (called a tracelet) is a trace of execution of some component of a structured program. It may be decomposed into two disjoint but neighbouring subsets p and q in two ways: one of them $(p; q)$ represents sequential composition, and the other $(p|q)$ represents concurrent composition. The arrows between p and q form the common part of the perimeter that separates them. A tracelet containing a single transaction cannot be further decomposed.

Section 3 introduces the concept of a tracelet as a figure representing the execution of some nested component of the program structure. Typical components are $(p; q)$ or $(p|q)$, standing for sequential or concurrent composition of subordinate components p and q. The actions of the original (bracketed) tracelet may then be split disjointly into separate tracelets for p and for q, which therefore share no actions. The arrows between them form a shared part of the perimeter of both of them. A line that passes through all the shared arrows can be drawn horizontally in the case of sequentiality or vertically in the case of concurrency. The splitting process may be continued until every tracelet contains only a single transaction, which cannot be further decomposed. The empty tracelet represents execution of a null command of the program, which of course does nothing.

Section 4 defines a pre-ordering relation $p \leq q$ between tracelets. It means that p is a possibly more interleaved version of q. If the converse relation also holds, the two tracelets are regarded as equal. From the definition of the ordering we prove informally all the laws of CKA whose variables range over singleton tracelets. They are as follows:

1. The operators ; and | are both associative, and both have the null command as unit.
2. Both operators are monotonic, for example $p < q$ implies $p; r \leq q; r$ and $r; p \leq r; q$.
3. Finally, an "interchange" law expresses a characteristic property of interleaving: $(p|q); (p'|q') \leq (p; p')|(q; q')$.

In an example proof we use a combination of all these laws to derive a fully interleaved version of an example tracelet.

Section 5 defines a program as the family of all its possible executions. The family is therefore downward closed, in that it contains all the more interleaved versions of any tracelet that it contains. A non-deterministic choice between programs is simply the set union of their two families. This disjunction has all the usual algebraic properties: associativity, commutativity, and idempotence; in addition, both ; and | distribute through it. The unit of disjunction is the empty family of traces, denoting a program which has no executions. This is the fate of a program containing a syntax error or a type error, or other errors which the language definition requires to be detected at compile time. Section 6 gives a simpler (more abstract) model of CKA. It abstracts from the intricate network of internal actions and arrows of a tracelet, and defines the two composition operators solely in terms of the perimeters of the operands. The common part of their perimeters is removed, and the rest forms the perimeter of the result of the composition. The function which maps a tracelet to its perimeter is a homomorphism w.r.t. ; and |, and therefore preserves all the star-free algebraic properties of the CKA. For some purposes, this perimeter model is an oversimplification, because it fails to model the phenomenon of deadlock resulting from a cyclic chain of causation. Cyclicity is a programming error that halts a group of threads, when each of them is waiting for occurrence of actions of other members of the cycle. This problem is solved by a second model, which retains the internal causal connectivity between the arrows of the perimeter. This model enables absence of deadlock to be proved, or at least detected. Section 7 reports

the early steps towards a formalisation of the geometric model in Isabelle. So far it provides the concepts and mechanical proofs of most concepts of the previous section. It gives a summary of the remaining steps towards a complete formalisation.

2 Primitive Concepts

We model a concurrent computer program as the set of all its possible executions on any computer system that offers an implementation of its programming language. Each execution is modelled by a discrete geometric diagram called a trace, which is drawn on a two-dimensional surface. The horizontal axis represents spatial distribution of locations in the memory of the computer system. The vertical axis represents the interval of time during which the program is executed.

The primitive components of our discrete geometry include analogues of the points, the lines and the figures familiar from Euclidean geometry. We have no concept of measurement of time or of distance in space. We maintain a distinction between horizontal and vertical coordinates; but whenever convenient, they are not drawn straight. Labels may be attached to a component: they describe its interpretation in the actual program execution.

A point represents a primitive action performed inside or in the immediate vicinity of the computer system during a single execution of the complete program. Every point is the unique member of the intersection of a horizontal and a vertical coordinate; all other such intersections are empty.

A *vertical line* is a non-empty sequence of points along a vertical coordinate that represent the sequential behaviour of an object stored at a particular location of the computer memory. This location number or name serves as a label unique to the line. Typical objects are threads or (possibly structured) variables. The topmost point of the line represents the primitive action of allocation of the object (or forking of a thread), and its bottommost point represents its disposal (or join of a thread). The intermediate points represent the temporal sequence of actions in which the object engages while it exists.

A *horizontal line* is a non-empty sequence of points along a horizontal coordinate whose actions appear to take place simultaneously as a single transaction. It is labelled by a reference to the basic command in the program which called for its execution. Apparent simultaneity will be ensured by disallowing any state of memory which records the performance of only some of the actions of a transaction, while omitting the rest. This follows the familiar definition of atomicity, without placing any constraint on how it is implemented.

A frequent type of transaction contains just two actions, one from the thread issuing the instruction that triggered the action, and the other from an object (usually owned by that thread) which performs the action required by the instruction. A transaction containing just a single action of a single object represents an autonomous behaviour of the object. Other transactions involve more than two objects. For example, a communication on a synchronised channel re-

quires simultaneous actions of six objects: two threads, an output port and an input port for the channel, and finally two variables which supply and receive the communicated value.

A pair of consecutive points on the same line is called an *arrow*. On a vertical line, the higher point is called the source, and the lower one is the target. On a horizontal line, an arrow may point either to the left or to the right. A vertical arrow is labelled by the value stored in its location of memory during the interval between its source action and its target action.

A subset of horizontal and vertical arrows represent *buffered communications* between threads. A horizontal communication arrow is labelled by the value of the message communicated. A vertical communication arrow conveys ownership of the object from one thread to another. It is convenient to draw communication arrows sloping at a slight angle from their nominal orientation.

A *tracelet* contains, surrounded by a rectangle, the subset of the points of a trace which occurred during execution of a single syntactic component of a structured program, i.e., a node in its abstract syntax tree (AST). This means that the complete trace is an execution of the root of the AST; and a typical leaf of the AST is a basic command of the program whose execution is a tracelet containing a single transaction. An *empty tracelet* (which we will call **1**) is an execution of the null command, which of course does nothing.

To summarise the basic concepts of our discrete geometry, we introduce names for infinite mathematical universes, containing all conceivable instances of the primitive concepts of our geometry. Let Pt be the set of all conceivable points; let Vert be the universe of all pairs of points that might feature as the tail or the head of an arrow in a vertical line. Let Hor be the set of all pairs of points that might feature as tail and head of a horizontal arrow. Let Comm be the set of all communication arrows (often drawn diagonally); they are also either in Vert or in Hor. Define Dep = Vert + Hor, where + denotes the union of disjoint sets. Its pairs are called arrows or dependences, because it is impossible for the tail action of an arrow to be performed before its head action.

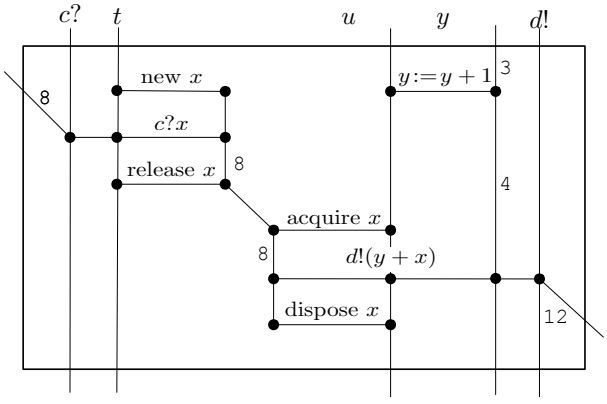

Fig. 1. A sample tracelet

Example 2.1 Fig. 1 shows a typical small tracelet. Its points are enclosed in a rectangular perimeter. There are six vertical lines, carrying the labels $c?, t, x, u, y$ and $d!$. Each label stands for the name or location of the object whose behaviour is recorded in the labelled line. All the vertical lines (except x, which is local to this tracelet) extend beyond the rectangle, both above it and below it. The lines t and u stand for threads, x and y are variables, $c?$ is the input port of a channel, and $d!$ is the output port of a different channel.

There are also seven horizontal lines. Two of them extend beyond the perimeter of the tracelet, one on the left and the other on the right. The three lines on the left each contain an action of the thread t, which issues the command for the transaction to occur. Similarly, the four lines on the right are executions of commands from the thread u. The other actions in each transaction are performed by objects (variables) owned by the threads: x is owned by t on the left and by u on the right.

The diagonal arrow in the middle of the diagram is a vertical arrow representing transfer of ownership of the variable x from the thread t to the thread u. The diagonal arrows entering and leaving the perimeter on the left and on the right are inputs and outputs of values on the buffered channels c and d, respectively.

The example shows a trace of the life history of the variable x. It begins with the allocation by its initial owner, the thread t. The next action is the allocation of an initial value to the new object. The value is acquired by input from channel c. The next two actions are a release of ownership by t, and its acquisition by the other thread u. This thread then outputs on channel d the value of the variable y, incremented by the current value of x. Finally, the variable x is disposed by its current owner. □

3 The Geometry of Tracelets

In this section, all tracelets will be subsets of the points of one single overall trace. Recall that each point is uniquely labelled by its coordinates. We can therefore identify a tracelet uniquely within its trace by the set of its points. All arrows that begin or end in a point of a tracelet are considered as part of that tracelet as well. For tracelets we use variables $p, q, r, \ldots$. The exterior $-p$ of p is defined as its relative complement $\mathsf{Pt} - p$, containing all points not in p.

Let $\times$ denote the Cartesian product operator between sets, i.e. the set of pairs (the relation) which contains all members of its first operand paired with all members of its second operand. By convention $\times$ binds tighter than union and intersection. The input arrows of p are $\mathit{input}(p) =_{df} -p \times p \cap \mathsf{Dep}$, and the output arrows are $\mathit{output}(p) =_{df} p \times -p \cap \mathsf{Dep}$. We define the *perimeter* of p as the set of arrows which have one end in p and the other end outside it; or more formally, $\mathit{perimeter}(p) = \mathit{input}(p) + \mathit{output}(p)$.

As mentioned in Example 2.1, a tracelet p is drawn as a rectangle which encloses all the points in p, and excludes all points in $-p$. That rectangle does

not pass through any of these points; it passes just once through each of the perimeter arrows.

Note (for interest) that the bounding rectangle is a closed curve that satisfies an analogue of the Jordan Curve theorem. Define a continuous line as a finite non-repeating sequence of arrows, in which the source or target of each arrow is also the target or source of one of its pair of neighbours within the sequence, or of its only neighbour in the case of endpoint of the chain. Every chain of arrows from one endpoint inside the rectangle to another endpoint outside it must cross at least one rectangle edge. This is proved by a simple induction on the length of the chain.

The perimeter of a rectangle is partitioned into its four edges. A horizontal edge does not contain any horizontal arrows, unless they are (sloping) communication arrows. Similarly, a vertical edge does not contain any vertical arrows unless they are transfers of ownership (also sloping). In drawing a perimeter, the top and bottom edges are horizontal and the left and right edges are vertical.

Each horizontal edge of the perimeter defines the state of part of the memory of the computer system at the relevant time coordinate. It is known in separation logic as a statelet. The top edge defines the initial state that is passed to the tracelet when it starts, and the bottom edge is passed as the final state on completion of execution.

The *content* of the memory at each horizontal edge is defined by the labels on the arrows that pass through the edge. It is defined in the standard way as a partial function which maps the location of each arrow crossing the edge (say $l_1, l_2, \ldots$) to the value (say $v_1, v_2, \ldots$) which labels that arrow. The function is written in the notation of separation logic. The infix binary operator $*$ stands for the disjoint union of the functions on either side of it. The function $(l \mapsto v)$ is a singleton function, whose whole domain is the singleton $\{l\}$ and which maps l to v. The value of the whole statelet is written in the form

$$(l_1 \mapsto v_1) * (l_2 \mapsto v_2) * \cdots$$

In separation logic, this formula is interpreted as an assertion that the value of l_1 is v_1, and the value of l_2 is v_2, etc.

The content of a vertical edge of a tracelet is defined similarly. But first, we must supply distinct names for all the messages that cross the edge. In the case of a communication channel, we use the channel name subscripted by the index of the message in the sequence of all messages passed on the channel, for example: $(c_4 \mapsto 12)$.

The specification of a tracelet contains the formula for all four edges of its perimeter. The formula for Fig. 1 is written on separate lines for each edge.

$(y \mapsto 3) * (c?, d!, t, u \mapsto \_)$	at the Top
$(d_{27} \mapsto 12)$	on the Right
$(y' \mapsto 4) * (c?, d!, t, u \mapsto \_)$	on the Bottom
$(c_9 \mapsto 8)$	on the Left

The first line states that the initial value of y is 3, and that the other named objects have been allocated. The second line says that (say) the 27th message

sent on channel d was 12. The third line gives the final value of y, and states that the other objects are still allocated. The fourth line states that channel c received the value 8 as the 9th message.

3.1 Sequential and Concurrent Composition

Our definition of the ; and | operators will be unconventional. Instead of defining how two tracelets can be composed to give the required result, we describe how the result can be decomposed to give the tracelets of its parts. It seems to be easier to learn first how to take something apart, and how to put it together later.

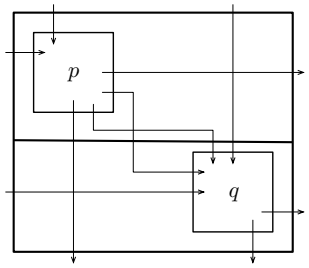

Fig. 2. Sequential composition

Consider a node of the program AST labeled by the operator of sequential composition. Let r be the tracelet for the considered node, and let p and q be the tracelets for its two immediate offspring in the corresponding AST. We describe this situation by the equation $r = p;q$. Now draw a horizontal coordinate internal to the rectangle for r, with all points in p above it, and all points in q below it. The diagram (see Fig. 2) makes it clear that the rectangle for p shares its top edge with r, and its bottom edge with q; similarly, the bottom edge of q is shared with that of r. The left and right edges of r are split into two disjoint parts, and the two top parts are assigned to p and the lower parts to q.

A defining feature of sequential composition is that an implementation can execute it by completing the execution of it first operand before starting execution of the second operand. This would be impossible if any action of the first operand were dependent on any action of the second operand. So the drawing of a horizontal edge is subject to the constraint that no arrow should point from its second operand to its first. That is assured by the fact that a horizontal edge contains only vertical and sloping horizontal arrows, and they all point downwards.

The practical consequence of this constraint is that is impossible to violate the atomicity of a transaction, except at one of its sloping arrows. Memory is represented by a horizontal edge; so any memory that records the result of the

action at one end of a non-sloping horizontal arrow must also record the action at the other end. Otherwise the constraint is violated.

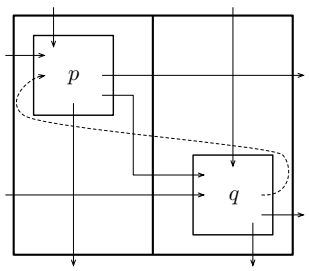

Fig. 3. Concurrent composition

A similar diagram can be drawn for concurrent composition (see Fig. 3), with a new vertical edge instead of a horizontal one. It leads to a similar pattern of sharing of left and right vertical edges, and a similar splitting of the top and bottom horizontal edges. Again, the vertical edge can contain only horizontal and sloping arrows pointing from left to right.

The practical consequence of this constraint is that no object can be owned by more than one thread at any one time. The only way that an object can be shared between two threads is by passing ownership between them by means a sloping vertical arrow. In a conventional view of sharing, ownership is passed between every pair of its actions. Such an object is represented geometrically by a vertical line, all of whose arrows are sloping.

If any of the constraints described above are violated, we simply say that the diagram for $p; q$ or for $p|q$ is undefined it is just not a tracelet. A composition is also undefined if the values which label any arrow in the edge differ in the two operands. Further reasons for the undefinedness of transactions that executed basic commands are given in the definition of these commands, which should be given in the definition of any particular programming language. Further pursuit of this topic is beyond the scope of this paper.

Summary To summarise and complement our decompositional definitions of the operators, we give a bottom-up formal presentation of some of the details. We start with a diagrammatic presentation. Figs. 2 and 3 show explicitly the pattern of arrows that cross the internal and external edges of a tracelet split horizontally or vertically. Each arrow of the figures represents a (maybe empty) set of arrows in a diagram. Arrow sets that must be empty are simply not shown. We use the convention that horizontal arrows leave their rectangle through the right edge, and enter it through the left edge.

The equations given below are derived by studying the figures. Let $T(p)$ be the set of arrows crossing the top edge of p, and let $B(p), L(p)$, and $R(p$ be

defined similarly as the bottom, left and right edges. Then Fig. 3 shows that

$$T(p \mid q) = T(p) + T(q) \ , \qquad B(p \mid q) = B(p) + B(q) \ .$$

The disjoint union is the separating conjunction that defines the initial and the final states of $p|q$: we have $B(p) \cap T(q) = \{\} = T(p) \cap B(q)$. There are no vertical arrows between p and q. This means that no state of memory is passed between them:

$$L(p \mid q) = (L(p) - R(q)) + (L(q) - R(p)) \ .$$

The horizontal inputs of p are taken either from the horizontal inputs of q or from the environment of $p \mid q$ (but not both); and similarly for the horizontal inputs of q. The equation for $R(p \mid q)$ is similar, with L and R interchanged.

Note the dashed curved arrow from $R(q)$ to $L(p)$. Since p is on the left of q, the arrow from p to q cannot be drawn as a straight line in two dimensions while observing the above convention. One could imagine that it was drawn on the back of the paper on which the diagram is drawn. Or one could maintain a uniform left-to-right direction of horizontal arrows by imagining the whole diagram drawn on the curved surface of a cylinder.

Fig. 2 shows the graph for sequential composition. It differs from Fig. 3 in two ways. Firstly, the curved arrow is removed, because it would violate our intended meaning of sequential composition. It would actually prevent an implementation of sequential composition from executing the whole of p before starting the execution of q. Secondly, a new internal arrow is introduced to stand for the transmission of the state of memory on termination of p and initiation of q. That is surely another part of our intention when using semicolon.

Derivation of the equations for sequential composition from this diagram is left as an exercise.

3.2 Quadrangulation

We now describe a process for splitting a complete trace or tracelet into all its component tracelets, so that it matches the AST of the program whose execution it represents. The splitting described above for $p;q$ or $p|q$ is repeated on p and on q, and then repeatedly on the smaller tracelets that result from earlier splittings. Once a tracelet has been split it cannot be split again as a whole — only its parts might be split further. Therefore no arrow can be split more than once by a horizontal or a vertical edge. By analogy with the familiar triangulation of figures in Euclidean geometry, we call the process *quadrangulation*. The process is *complete* when all splittable arrows have been split exactly once.

The completely quadrangulated tracelet is a tree which exactly matches the AST of its program. The points of each tracelet in it are the disjoint union of the points of each of its offspring. So any tracelet includes all points of any of its descendants, and is included among the points of all its ancestors. It is helpful to use the text of the program itself as a linear representation for the whole quadrangulated tracelet. Typical examples of such terms are $p \mid (q \mid r)$ and $(p \mid q) \, ; (p' \mid q')$, where $p, q, \ldots$ are variables standing for further descendant

tracelets, or (in the case of a leaf) the corresponding basic command of the program.

Example 3.1 Figs. 4 and 5 show the result of the first three steps in two different quadrangulations of the tracelet shown in Fig. 1. To avoid distraction, the labels that are irrelevant to our current purposes have been removed. The titles on the figures are the formulae that describe the quadrangulations. They use bracketing to indicate the order in which the splits were made.

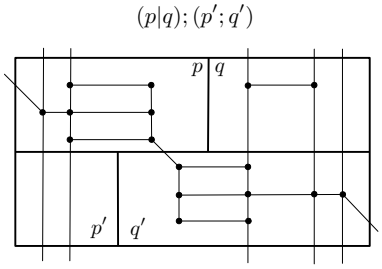

Fig. 4. Tracelet from Fig.1 split as $(p|q);(p';q')$

In Fig. 4 the first split is horizontal and the next two are vertical, whereas in Fig. 5 this order is reversed.

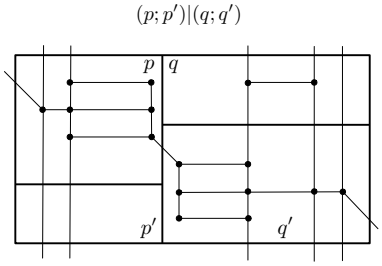

Fig. 5. Tracelet from Fig.1 split as $(p;p')|(q';q')$

Otherwise, the figures are very similar. All the points and arrows internal to each of the rectangles p, p', q, q' are identical on both figures, and all the internal arrows and splits within them are the same. The only difference is at the centre of the diagram, where the sloping communication arrow is split horizontally in Fig. 4, whereas it has been split vertically in Fig. 5. □

4 Algebra of Tracelets

In this section, we will continue to use the single word tracelet for a quadrangulated tracelet. Our algebra is a pre-order algebra, in the sense that it uses a pre-order relation $\leq$ (i.e., a reflexive and transitive relation), in place of the more usual equality symbol $=$ between the left and right hand sides of an equation. In an order algebra, an analogue of equality is re-introduced as an equivalence, again written as $=$, defined as the conjunction of $\leq$ and its converse. In our geometry, the ordering $p \leq q$ between tracelets p, q has an informally expressed meaning that p represents a more sequential execution than the one represented by q or equivalently that q is more concurrent than p.

To formalise this intuitive definition, we define $V(p)$ as the set of all sloping arrows crossing a vertical edge internal to p. Then

$$\begin{aligned} V(\mathbf{1}) &= \{\}\ , \\ V(p\,;q) &= V(p) + V(q) \\ V(p \mid q) &= V(p) + V(q) + (p \times q + q \times p) \cap \mathsf{Hor}\ . \end{aligned}$$

Similar equations are satisfied by $H(p)$, the set of all sloping arrows crossing horizontal edges in p:

$$\begin{aligned} H(\mathbf{1}) &= \{\} \\ H(p\,;q) &= H(p) + H(q) + p \times q \cap \mathsf{Vert} \\ H(p \mid q) &= H(p) + H(q) \end{aligned}$$

Every internal sloping arrow of p may be in $V(p)$ or $H(p)$, but never in both. If p is completely quadrangulated then the sets $V(p)$ and $H(q)$ are complements of each other relative to the set $\mathsf{Comm} \cap p \times p$ of all sloping arrows within p. Hence, if p and q are complete quadrangulations with identical underlying tracelets then by contraposition it follows that

$$V(p) \subseteq V(q) \iff H(q) \subseteq H(p)\ .$$

For an unsplit tracelet, V and H return $\{\}$.

We define the relation $p \leq q$ in two clauses. The first requires that p and q are entirely equal as tracelets; only their quadrangulations can differ. Hence the two tracelets have the same actions, and the same internal arrows, with the same orientations and the same labels. In particular, all the arrows not split by the quadrangulations match exactly in p and q. The second clause requires that $V(p)$ is contained in $V(q)$ and $H(q)$ is contained in $H(p)$. By the above remark, if $V(p)$ and $H(p)$ as well as $V(q)$ and $H(q)$ are relative complements (which holds, in particular, for complete quadrangulations p, q) then we may use either alternative at convenience. The definition allows a sloping arrow that crosses a horizontal edge in p to cross a vertical edge in q. Because set inclusion is a partial order, so is the relation $\leq$.

Example 4.1 Let r and r' be the quadrangulations in Figs. 4 and 5, respectively, and let a be the only diagonal arrow there. Then we have $V(r) = \{\}$ and

$H(r) = \{a\}$, whereas $V(r') = \{a\}$ and $H(r') = \{\}$. Since there is exacly the communication arrow a in both r and r', $V(r)$ and $H(r)$ as well as $V(r')$ and $H(r')$ are relative complements of each other. According to the remarks above and in Example 3.1 therefore $r \leq r'$. Below we will see that this is a special instance of a general law. □

From the definition we will now derive a set of algebraic laws governing sequential and concurrent composition; they are the basic laws of a Concurrent Kleene Algebra (CKA) [24]. For simplicity, we restrict ourselves here to complete quadrangulations. This allows us in each case to choose the simpler of the equations for V and H. There is a treatment of the general case which will be presented in a follow-up paper.

Theorem 4.2 (example) *$p\,;q \leq p \mid q$ and $q\,;p \leq p \mid q$.*

Proof. $V(p\,;q) = V(q\,;p) = V(p) + V(q) \subseteq V(p \mid q)$, by the definition of V. □

This theorem justifies the implementation of the concurrent composition by executing the operands in either order. However the justification is void in the case that the left hand side is undefined. The existence of dependences between one operand and the other will make one or both of the interleavings void.

Note that both interleavings of $p \mid q$ are below it in the ordering, but that $\mid$ is not itself commutative. Thus our model does not satisfy the standard definition of sequential consistency, that concurrency is a non-deterministic choice of all its possible interleavings. An asymmetric example of concurrency is the chaining operator $>>$ of CSP which allows communication only from left to right.

Theorem 4.3 (unit) *$p \mid \mathbf{1} = p = \mathbf{1} \mid p$ (and the same for $;$).*

Proof. $V(p \mid \mathbf{1}) = V(p) + V(\mathbf{1}) + p \times \{\} \cap \mathsf{Hor} \cap \mathsf{Comm} = V(p)$. The second and third terms on the rhs are both empty. In words: there are no points in $\mathbf{1}$, and therefore no arrow can cross its (invisible) perimeter. □

Theorem 4.4 (association) *$p \mid (q \mid r) = (p \mid q) \mid r$ (and the same for $;$).*

Proof. $H(rhs) = H(p \mid q) + H(r) = H(p) + H(q) + H(r) =$
$H(p) + H(q \mid r) = H(lhs)$.
The proof for $;$ is similar, using V instead of H. □

Theorem 4.5 (monotonicity) *If $p \leq q$ then $p\,;r \leq q\,;r$ (and the same for $\mid$).*

Proof. Assume $V(p) \subseteq V(q)$. Then, by monotonicity of $+$ and the hypothesis,

$$V(p\,;r) = V(p) + V(r) \subseteq V(q) + V(r) \ .$$

The proof for $\mid$ uses H instead of V. □

Theorem 4.6 (interchange) *$(p \mid q)\,;(p' \mid q') \leq (p\,;p') \mid (q\,;q')$.*

Proof. Let $K = V(p) + V(q) + V(p') + V(q')$. Then

$$V(lhs) = K + (p \times q + p' \times q') \cap \mathsf{Hor} \cap \mathsf{Comm} \subseteq K + (p + p') \times (q + q') \cap \mathsf{Hor} \cap \mathsf{Comm} = V(rhs) \ ,$$

because $\times$ distributes through $+$. □

Corollary 4.7 (frame) $(p \mid q) \,;\, p' \leq (p \,;\, p') \mid q$ *and* $p \,;\, (p' \mid q') \leq (p \,;\, p') \mid q'$.

Proof. For the first law substitute **1** for q'. By the unit law, the occurrences of **1** can be cancelled. The second law follows symmetrically. □

Note that Th. 4.2 follows by setting $p' = 1$ and substituting q for q' in the second law and by setting $p = 1$ and substituting p for p' in the first law.

The purpose of algebraic laws is to permit an implementation to replace the text of a submitted program by another text derived from it by algebraic reasoning. The hope is that the executed code will be better adapted to the structure and the detail of the capabilities of the executing hardware. Such transformations may be made by a compiler or by an instruction pipeline in the hardware of a computer chip.

For example, suppose the executing computer system has less processors than the number of threads initiated by the running program. In this case, concurrency has to be replaced by interleaving (time-sharing), in which an execution of several threads may be an interleaving of their separate sequential traces. In fact, repeated application of all the laws proved above can generate arbitrary interleaved executions of any pair (or group) of concurrent program.

This is demonstrated by an example of a fully algebraic proof. To avoid clutter, semicolons are omitted except when they are necessary to indicate how the interchange law is to be applied. Also, the use of monotonicity remains tacit.

$$\begin{array}{ll} & abcd \mid xyzw \\ = & \quad \{\!\![\text{ (assoc ;) }]\!\!\} \\ & (a \,;\, bcd) \mid (xy \,;\, zw) \\ \geq & \quad \{\!\![\text{ (interchange) }]\!\!\} \\ & (a \mid xy) \,;\, (bcd \mid zw) \\ \geq & \quad \{\!\![\text{ (assoc ;) }]\!\!\} \\ & a \mid (x \,;\, y) \,;\, (b \,;\, cd \mid zw) \\ \geq & \quad \{\!\![\text{ (frame) }]\!\!\} \\ & (a \mid x) \,;\, y \,;\, (b \mid zw) \,;\, cd \\ \geq & \quad \{\!\![\text{ (Theorem 4.2) }]\!\!\} \\ & axybzwcd \ . \end{array}$$

Interleaving is introduced by each step that uses the interchange law or its corollary. The position of the semicolon indicates a scheduling decision that the two semicolons on the rhs of the law will be reached simultaneously by both

threads, at exactly the moment when the lhs reaches its single semicolon. Different scheduling decisions would use different associations at each step, and thereby generate all possible different interleavings.

5 From Tracelets to Programs: Lifting

So far we have dealt with single tracelets. A *program* is identified by and with the set of all possible tracelets of its execution, which is what we will explore next. This section explains how all the operators defined on tracelets can be lifted to sets of tracelets in such a way that all the laws proven for operators on tracelets are preserved.

5.1 Elementwise Lifting

We do not consider arbitrary sets of tracelets. Rather, we adopt a downward closure condition which ensures that a relation $\leq$ between programs can be defined as simple set inclusion. A set P of tracelets is *downward closed* w.r.t. the pre-order $\leq$ if $p \in P$ and $p' \leq p$ imply $p' \in P$ as well. Downward closure codifies our intention that any program that can validly be executed concurrently can also be validly executed more sequentially.

If $\circ$ is a binary, possibly partial, operator on tracelets then its *elementwise lifting* to programs P, P' is defined as the downward closure of the set of all defined compositions between P and P', i.e., the set of all tracelets q such there are $p \in P$ and $p' \in P'$ with defined $p \circ p'$ and $q \leq p \circ p'$.

Since we do not only use equational laws but also inequational ones, we have to define a relation $\leq$ between programs if we want to lift laws to programs. While it is clear what equality means for sets, there are several ways to extend a pre-order like $\leq$ to sets. We choose the following definition: $P \leq P'$ holds iff every tracelet in P is below some tracelet in P'. For downward closed sets (and hence programs) $\leq$ coincides with inclusion $\subseteq$. This means that we can use ordinary union to introduce non-deterministic choice into our algebra of programs, and define it as set union. Furthermore, it means that an implementation can make an arbitrary choice from any non-deterministic variants allowed by the program under execution, giving our intended interpretation of non-determinism a demonic flavour.

Let T, T' be terms involving variables and operators on tracelets, and consider the inequational law $T \leq T'$. A sufficient condition for lifting this law from tracelets to programs is *linearity*, viz. that every variable occurs at most once on both sides of the law and that all variables in the left hand side T also occur in the right hand side T'. Examples are the frame and exchange laws. For equations a sufficient condition is *bilinearity*, meaning that both inequations that constitute an equation are linear. Examples are associativity, commutativity and neutrality; a counterexample is distributivity. The main result is as follows.

Theorem 5.1 *If a linear law $T \leq T'$ holds for tracelets then it also holds when all variables in T, T' are interpreted as variables for programs and the operators are interpreted as the elementwise liftings of the corresponding trace operators.*

A detailed proof for general pre-orders can be found in [25]. The technique is classical in mathematics; for related results see among others [14,19,15] (and also [7] for a survey).

We illustrate the gist of the proof for the case of the law $P\,;P' \leq P \,|\, P'$ lifted from Th. 4.2. Assume $r \in P\,;P'$. By the above definition there are $p \in P, p' \in P'$ such that $r \leq p\,;p'$. Since the frame law holds at the trace level, we have $p\,;p' \leq p \circ p'$. Moreover, $p \circ p'$ is in the set of all $\circ$-combinations of traces from P with traces from P' and hence also in its downward closure $P \,|\, P'$, so that we are done.

5.2 Errors, Recursion and Iteration

There are further useful consequences of our definition of programs. The set $\mathcal{P}$ of all programs forms a complete lattice w.r.t. the inclusion ordering; it has been called the *Hoare power domain* in the theory of denotational semantics (e.g. [31,27,8]).

The least element of $\mathcal{P}$ is the empty program $\emptyset$ which can also serve as an error element, modelling a completely faulty module without any sensible tracelet. A more detailed, elementwise, error handling is already contained in the definition of the elementwise lifting of operators: all erroneous, undefined combinations of tracelets are ruled out from the combination of the containing programs. This was already stated in Section 3.1.

The greatest element of $\mathcal{P}$ is the program U consisting of all tracelets. Infimum and supremum in $\mathcal{P}$ coincide with intersection and union, since downward closed sets are also closed under these operations.

Therefore we can define (unbounded) choice between a set $\mathcal{Q} \subseteq \mathcal{P}$ of programs as

$$[\!]\, \mathcal{Q} =_{df} \bigcup \mathcal{Q}$$

with binary choice as the special case

$$P \,[\!]\, P' =_{df} P \cup P' \ .$$

The lifted versions of monotonic tracelet operators are monotonic again (see [25]), but even distribute through arbitrary choices between programs.

Monotonicity of the lifted operators, together with completeness of the lattice of programs and the Tarski-Knaster fixed point theorem, guarantees that recursion equations have least and greatest solutions. More precisely, let $f : \mathcal{P} \to \mathcal{P}$ be a monotonic function. Then f has a least fixed point μf and a greatest fixed point νf, given by the following formulas:

$$\mu f = \bigcap \{P \,|\, f(P) \subseteq P\} \ , \qquad \nu f = \bigcup \{P \,|\, P \subseteq f(P)\} \ .$$

With our operator ; this can be used to define the Kleene star (see e.g. [10]), i.e., unbounded finite sequential iteration, of a program P as $P^* =_{df} \mu f_P$, where

$$f_P(X) =_{df} \mathsf{skip} \,[\!]\, (P\,;X) \ ,$$

where $\mathsf{skip} =_{df} \{\mathbf{1}\}$ is the idle program. Since f_P, by the above remark, distributes through arbitrary choices between programs, it is even continuous and Kleene's fixed point theorem tells us that $P^* = \mu f_P$ has the iterative representation

$$P^* = \bigcup\{f_P^i(\emptyset) \mid i \in \mathbb{N}\} \ , \tag{1}$$

which transforms into the well known representation of star, viz.

$$P^* = \bigcup\{P^i \mid i \in \mathbb{N}\}$$

with $P^0 =_{df} \mathsf{skip}$ and $P^{i+1} =_{df} P\,;P^i$.

Infinite iteration P^ω can be defined as the greatest fixed point νg_P where

$$g_P(X) =_{df} P\,;X \ .$$

Along the same lines, unbounded finite and infinite concurrent iteration of a program can be defined. For further forms of iteration we refer to [25].

We conclude this section with a brief description how pre-post-condition semantics can be integrated into our approach. As in [24] one can define, for programs P, P' and Q, the Hoare triple

$$P\ \{\!\{Q\}\!\}\ P' \iff_{df} P\,;Q \subseteq P' \ .$$

It expresses that, after any tracelet in "pre-history" P, execution of Q is guaranteed to yield an overall tracelet in P'. From this one can derive the standard properties of Hoare logic and separation logic; for further details we refer to [24,21].

6 Interfaces and Specifications

We now deal with *specifications* that abstract, to a certain extent, from the interior arrows of tracelets but preserve their interfaces, i.e., their perimeters. For this analysis the distinction between horizontal and vertical arrows is inessential; we only reason about the overall dependence relation Dep.

6.1 Two Types of Specifications

A first, quite radical, abstraction reduces a tracelet just to its perimeter that describes the interaction of the tracelet with its environment. It presents a pure black-box view of the tracelet.

This abstraction can be formalised as follows. The *input points* $in(p)$ of p are the end points of the input arrows to p, while the *output points* $out(p)$ of p are the starting points of the output arrows of p. Now the set of points of $perspec(p)$ is $in(p) \cup out(p)$, while its arrow set is given by $perimeter(p)$. This implies

$$perimeter(perspec(p)) \ = \ perimeter(p) \ . \tag{2}$$

A second, more refined, abstraction $connspec(p)$ of p records connections in the form of dependences between input and output points of p. It can be drawn as a tracelet containing only chains with at most three arrows, namely an input, an output and possibly an intermediate arrow. If present, the latter records the existence of a direct or indirect dependence between its source and target within p; however, the whole chain of intermediate internal points is omitted.

This abstraction allows an analysis which of the input arrows are actually useful in that they "contribute" to the outputs. Input arrows that are not connected to any output arrows could, together with the internal arrow chains emanating from them, be safely removed without affecting the observable behaviour of the tracelet. They will, inside p, lead to end points or, in the case of deadlock, to cycles of points that do not have outgoing arrows to points outside the cycles; therefore they cannot contribute to values in labels of output arrows from p.

The set of points of $connspec(p)$ is again $in(p) \cup out(p)$. The arrows of $connspec(p)$ are the input and output arrows of p plus a set of fresh arrows for each pair in $in(p) \times out(p) \cap \mathsf{Dep}_p^+$, where $\mathsf{Dep}_p =_{df} p \times p \cap \mathsf{Dep}$ is the local dependence relation for p. Using transitive rather than reflexive transitive closure ensures that a point e in $in(p) \cap out(p)$ does not receive an extra arrow (e, e) in $connspec(p)$. This takes care of singleton tracelets of the form $\dashv\!\!\rightarrow \bullet \dashv\!\!\rightarrow$ (where the brackets indicate the rectangle around the tracelet).

For tracelet p we have the decomposition

$$arrows(p) \;=\; perimeter(p) + \mathsf{Dep}_p \;,$$

where again + denotes disjoint union.

Both specification functions $s \in \{perspec, connspec\}$ are idempotent, i.e., satisfy $s(s(p)) = s(p)$.

6.2 Specification and (De-)Composition

To make such abstractions useful for the analysis of larger tracelets, they have to behave well w.r.t. composition or decomposition of tracelets. We will now show that this is indeed the case.

For this we use a generic (de)composition operator $\circ$ like in [25]. For tracelets p, p' with disjoint point sets,

$$p \circ p' \;=_{df}\; (p + p', arrows(p) \cup arrows(p')) \;.$$

Both operators | and ; from Section 3.1 can be seen as instances of $\circ$, since they administer the arrows involved in precisely that way.

Theorem 6.1 *For both specification functions $s \in \{perspec, connspec\}$ we have the homomorphic equation*

$$s(p \circ q) \;=\; s(s(p) \circ s(q)) \;.$$

The equation is homomorphic in the following sense. One can define a new operator $\circ'$ on specification tracelets r, t by $r \circ' t =_{df} s(r \circ t)$. Then, using idempotence of s, we have $s(p \circ q) = s(p) \circ' s(q)$.

We present the gist of the proof; full details can be found in the technical report [23]. Automated proofs of some parts are under way, see Section 7.

First we establish the behaviour of perimeter and local dependence on composed tracelets:

$$\left.\begin{array}{rcl} \mathit{perimeter}(p \circ p') & = & (\mathit{perimeter}(p) \cup \mathit{perimeter}(p')) - \mathit{intf}(p, p') \ , \\ \mathsf{Dep}_{p \circ p'} & = & \mathsf{Dep}_p \cup \mathsf{Dep}_{p'} \cup \mathit{intf}(p, p') \ , \end{array}\right\} \quad (3)$$

where $\mathit{intf}(p, p') =_{df} \mathit{arrows}(p) \cap \mathit{arrows}(p')$ is the interface between p and p'. Using (2) we obtain, moreover,

$$\mathit{intf}(\mathit{perspec}(p), \mathit{perspec}(p')) \ = \ \mathit{intf}(p, p') \ . \quad (4)$$

With the help of these properties easy calculations show that $s = \mathit{perspec}$ satisfies the homomorphic equation of Th. 6.1.

For the specification operator *connspec* it suffices to consider the local dependence relations of the tracelets on both sides of the homomorphic equations, since their perimeters coincide by the homomorphic property of *perspec* anyway. This also implies that the analogue of (4) holds for *connspec* as well:

$$\mathit{intf}(\mathit{connspec}(p), \mathit{connspec}(p')) \ = \ \mathit{intf}(p, p') \ .$$

For the local dependences we proceed in two steps. First, we have the following properties.

Lemma 6.2 *Set* $\hat{p} =_{df} \mathit{connspec}(p)$ *and likewise for* p'.

1. $\mathsf{Dep}_{\hat{p} \circ \hat{p}'} = \mathsf{Dep}_{\hat{p}} \cup \mathsf{Dep}_{\hat{p}'} \cup \mathit{intf}(p, p')$.
2. $\mathsf{Dep}_{\mathit{connspec}(\hat{p} \circ \hat{p}')} \subseteq \mathsf{Dep}_{\mathit{connspec}(p \circ p')}$.

The calculations are not too hard. However, showing the reverse inclusion

$$\mathsf{Dep}_{\mathit{connspec}(p \circ p')} \subseteq \mathsf{Dep}_{\mathit{connspec}(\hat{p} \circ \hat{p}')}$$

is much more laborious. Using the definitions this spells out to

$$(\mathsf{Dep}_p \cup \mathsf{Dep}_{p'} \cup \mathsf{C})^+ \cap \ \mathit{in} \times \mathit{out} \ \subseteq \ (\mathsf{Dep}_{\hat{p}} \cup \mathsf{Dep}_{\hat{p}'} \cup \mathsf{C})^+ \cap \ \mathit{in} \times \mathit{out} \ , \quad (5)$$

where $\mathit{in} =_{df} \mathit{in}(p) \cup \mathit{in}(p')$, $\mathit{out} =_{df} \mathit{out}(p) \cup \mathit{out}(p')$ and $\mathsf{C} =_{df} \mathit{intf}(p, p')$.

Let us first give an intuitive idea why (5) holds. Consider event-disjoint tracelets p, p' and events $e \in \mathit{in}(p), e' \in \mathit{out}(p')$ such that $(e, e') \in (\mathsf{Dep}_p \cup \mathsf{Dep}_{p'} \cup \mathsf{C})^+$. Consider an arbitrary path P from e to e' within $p + p'$. According to (3) we can group P into maximal pieces whose arrows are purely within Dep_p, purely within $\mathsf{Dep}_{p'}$ or consist only of "bridging" arrows in C. In Fig. 6.2, pieces of the first kind are indicated by dotted arrows, while interface and bridging arrows have solid lines.

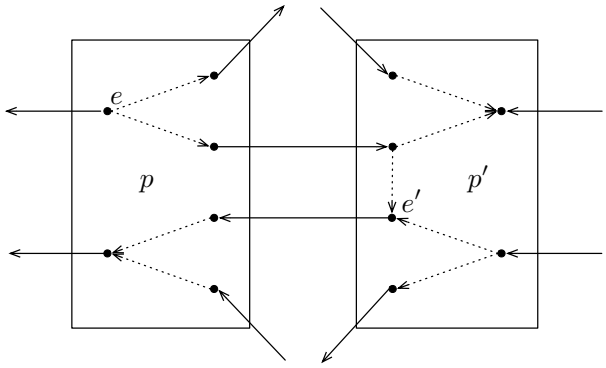

Fig. 6. Connection paths in a composition

The reason is that arrows from Dep_p cannot connect directly with those from $\mathsf{Dep}_{p'}$, because their end points lie in disjoint event sets. They can only connect via "bridges" in C. Now each of the maximal pieces within Dep_p or $\mathsf{Dep}_{p'}$ can be contracted to a single Dep_p^+ or $\mathsf{Dep}_{p'}^+$ edge, as is done by *connspec*. By maximality they have to start and end in events in $in(p) \cup out(p)$ or $in(p') \cup out(p')$, resp., which makes their contractions belong to $\mathsf{Dep}_{\tilde{p}}$ or $\mathsf{Dep}_{\tilde{p}'}$, resp. Therefore it does not matter if we contract a composition tracelet directly or first contract the maximal path pieces in its components and then contract the result further.

The formal proof uses regular algebra to good advantage; we denote relational composition by juxtaposition. We have to deal with the subexpression $(\mathsf{Dep}_p \cup \mathsf{Dep}_{p'} \cup \mathsf{C})^+$ occurring in the left hand side of (5), where we know from the definitions of $\mathsf{Dep}_p, \mathsf{Dep}_{p'}$ and $E \cap E' = \emptyset$ that $\mathsf{Dep}_p\mathsf{Dep}_{p'} = \emptyset = \mathsf{Dep}_{p'}\mathsf{Dep}_p$. We abstract a bit and show the following properties.

Lemma 6.3 *Consider relations R, S, T.*

1. $(R \cup S)^+ = R^+ \cup R^*(SR^*)^+$.
2. *If* $RS = \emptyset = SR$ *then* $(R \cup S)^+ = R^+ \cup S^+$ *and* $(R \cup S)^* = R^* \cup S^*$.
3. *If* $RS = \emptyset = SR$ *then* $(R \cup S \cup T)^+ = R^+ \cup S^+ \cup D(TD)^+$, *where* $D =_{df} R^* \cup S^*$.

For the expression occurring in the left hand side of (5) we obtain from Part 3

$$(\mathsf{Dep}_p \cup \mathsf{Dep}_{p'} \cup \mathsf{C})^+ = \mathsf{Dep}_p^+ \cup \mathsf{Dep}_{p'}'^+ \cup D(\mathsf{C}D)^+ , \tag{6}$$

where $D = \mathsf{Dep}_p^* \cup \mathsf{Dep}_{p'}'^*$. This is the formal counterpart of the above-mentioned path decomposition.

From this, further intensive use of regular algebra finally leads to a proof of (5), which establishes Th. 6.1 for $s =$ *connspec*.

7 Verification Tool Development

For practical uses of the geometric model in verifying concurrent programs, tool support is mandatory. This section outlines exemplarily how this can be achieved by formalising the CKAs exhibited in Section 4 together with the model of Section 6 in an interactive theorem prover. Isabelle/HOL [28] is used as an example.

We have already built mathematical components for variants of Kleene algebras, regular algebras and relation algebras in Isabelle [6,17,4,13,2], integrated some of them into verification components for sequential programs [16,5,18], local reasoning with separation logic [12] and the rely-guarantee calculus [3]. In all of them, an abstract algebraic layer has been linked via formal soundness proofs with concrete computational models, e.g. for the program store. The use of algebra makes the resulting components small and allows us to carry out large parts of the development by automated theorem proving. Here we follow the same approach. The underlying Isabelle theories can be found online[4].

7.1 Formalising CKA

A first step towards a verification component based on the geometric model consists in formalising CKA as an axiomatic type class in Isabelle.

class *cka* = *kleene-algebra* +
  fixes *pcomp* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** $\parallel$ *70*)
  assumes *pcomp-assoc*: $x \parallel (y \parallel z) = (x \parallel y) \parallel z$
  and *pcomp-comm*: $x \parallel y = y \parallel x$
  and *pcomp-oner* [*simp*]: $x \parallel 1 = x$
  and *pcomp-annir* [*simp*]: $x \parallel 0 = 0$
  and *pcomp-distribl*: $x \parallel (y + z) = x \parallel y + x \parallel z$
  and *interchange*: $(w \parallel x) \cdot (y \parallel z) \leq (w \cdot y) \parallel (x \cdot z)$

This class extends the operators and axioms of Kleene algebra by the concurrent composition operator and six further axioms. A concurrent iteration operator can be added along these lines. The extension brings all facts proved for Kleene algebras automatically into scope. It is easy, for instance, to derive the small interchange laws or the laws in Lemma 6.3 by automated theorem proving with Isabelle's Sledgehammer tool. Sledgehammer calls external automated theorem provers and SMT solvers and reconstructs their outputs by internally verified tools. This validates them relative to Isabelle's small trustworthy core.

7.2 Formalising Tracelets and Specifications

A second step is the formalisation of the tracelet model. We restrict our attention to the generic model from Sect. 6, which uses the dependency relation from Sect. 3. A refinement to models with several kinds of arrows is straightforward.

[4] `http://staffwww.dcs.shef.ac.uk/people/G.Struth/isa/GCKA/GCKA.thy`

type-synonym *'a graph* = *'a rel*

abbreviation *vertices g* ≡ *Field g*

definition *tracelets g* = *Pow* (*vertices g*)

definition *str τ g* ≡ *τ* ∈ *tracelets g*

definition *A τ g* = (*if str τ g then* ⌈*τ*⌉ ; *g* ∪ *g* ; ⌈*τ*⌉ *else undefined*)

A graph is formalised as a binary relation of type α, hence as the set of its arrows. Its set of points or vertices is thus the field of the relation; the union of its domain and range elements. Following Sect. 2, the set of tracelets of a graph is the power set of its vertex set. The subtracelet relation *str τ g* is defined next in the obvious way. It is generally used as a proviso on definitions and theorems (a tracelet type would have to depend on the graph). Finally, the set *A τ g* of arrows of τ in g consists of those arrows of g that have at least one point in τ, provided that τ is a subtracelet of g. It is undefined otherwise. In this definition, the function ⌈_⌉ lifts the set V to the subidentity relation $\{(v, v) \mid v \in V\}$ to support relation-algebraic reasoning..

Additional notions such as input and output arrows or vertices of tracelets can now be defined as partial functions relative to an underlying graph as well.

definition *iA τ g* = (*if str τ g then* ⌈−*τ*⌉ ; *A τ g else undefined*)

definition *oA τ g* = (*if str τ g then A τ g* ; ⌈−*τ*⌉ *else undefined*)

definition *iV τ g* = (*if str τ g then Range* (*iA τ g*) *else undefined*)

definition *oV τ g* = (*if str τ g then Domain* (*oA τ g*) *else undefined*)

The function − denotes set complementation. Various laws relating these vertices and arrows could then be derived easily be automated theorem proving. These enable automated proofs of some more intricate facts from Sect. 6. In addition, they allow us to define the perimeter, abbreviated as *ioA*, as *iA τ g* ∪ *oA τ g*, and the associated specification *perspec*, for which we write *S*.

definition *S τ g* = (*if str τ g then iV τ g* ∪ *oV τ g else undefined*)

By contrast to previous sections, *S τ g* is thus a set of vertices, and not a pair. In fact, the specifications from previous sections are neither subtraces of the underlying graph nor graphs themselves. This leads to complications in Isabelle's strongly typed setting. Instead, in the case of *perspec*, specifications are tracelets with respect to the perimeter of the underlying tracelet: *str* (*S τ g*) (*ioA τ g*) holds whenever τ is a tracelet in g. Obviously, *connspec* of a tracelet has the same vertex set as *perspec*, but is a tracelet with respect to a different set of

arrows. We therefore do not distinguish between the two in the above definition. Analogues of property (2) and idempotency of the specification function can be proved fully automatically for *perspec*, that is, $ioA\ (S\ \tau\ g)\ (ioA\ \tau\ g) = ioA\ \tau\ g$ and $S\ (S\ \tau\ g)\ (ioA\ \tau\ g) = S\ \tau\ g$, whenever τ is a subtracelet of g. For *connspec*, the arrow sets in formulas must be adapted. Additional facts can be found online.

Another essential ingredient of the graph model is the generic (de)composition operation from Sect. 6.2. We have formalised it as a partial function in Isabelle.

partial-function (*tailrec*) *tcomp* :: *'a set* ⇒ *'a set* ⇒ *'a graph* ⇒ *'a set* **where**
tcomp σ τ g = (*if str σ g* ∧ *str τ g* ∧ *σ* ∩ *τ* = {} *then σ* ∪ *τ else undefined*)

The sequential and concurrent compositions from Sect. 3.1 can be defined likewise. It is easy to prove the commutative monoidal properties of composition, subject to definedness. A variant of Theorem 6.1 for *perspec* is more tedious.

lemma *pS-tcomp*:
assumes *str σ g* **and** *str τ g* **and** *σ* ∩ *τ* = {}
shows *pS* (*tcomp* (*pS σ g*) (*pS τ g*) *g*) (*ioA σ g* ∪ *ioA τ g*) = *pS* (*tcomp σ τ g*) *g*

7.3 Further Formalisation Steps

The following steps are needed for completing the verification component. They follow the design of our previous verification components [5,18] closely.

Enriching the Model. Various kinds of edges, sequential and concurrent compositions, labels for programming concepts such as actions or transactions, and notions of memory location must be added to the extant tracelet model to obtain the full-fledged geometric model outlined in Sect. 2.

Tracelet and Powerset Algebra. The interchange laws for tracelets (Sect. 4) must be derived. The enriched tracelet model must be lifted to the powerset level and the CKA structure must be established at this level.

Formal Soundness Proof. An interpretation statement is needed to formalise soundness of the enriched tracelet model with respect to CKA within Isabelle's type class framework. All statements proved for CKA are then available within the model. This is important for verification condition generation with the Hoare logic outlined in Sect. 5.2 and for program refinement.

This completes the development of a verification component prototype based on the geometric tracelet model. Program verification is possible by using a shallow embedding of an appropriate programming syntax into the graph model. Alternatively, program syntax could be mapped into the model as usual.

One merit of the approach outlined is that the resulting verification component is correct by construction relative to Isabelle's small trustworthy core. Due to the link with CKA and the genericity of the tracelet formalisation used, the approach should also be robust to minor changes to the model, which has undergone a considerable evolution over time. Beyond a proof of concept and the formal verification of the results in this article, the component could therefore

serve as a reference that can be refined and modified easily by other researchers. Because of its simplicity and declarative nature it may also be useful as a template for implementing practical verification tools.

Acknowledgements We are grateful for valuable input from discussions with Jade Alglave, Peter OHearn, Peter Höfner, Matthew Parkinson, Stephan van Staden, Ian Wehrman, John Wickerson and Huibiao Zhu.

References

1. Alglave, J.: A formal hierarchy of weak memory models. Formal Methods in System Design 41(2), 178–210 (Jun 2012)
2. Armstrong, A., Foster, S., Struth, G., Weber, T.: Relation algebra. Archive of Formal Proofs (2014)
3. Armstrong, A., Gomes, V.B.F., Struth, G.: Algebraic principles for rely-guarantee style concurrency verification tools. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) FM 2014. LNCS, vol. 8442, pp. 78–93 (2014)
4. Armstrong, A., Gomes, V.B.F., Struth, G.: Kleene algebra with tests and demonic refinement algebras. Archive of Formal Proofs (2014)
5. Armstrong, A., Gomes, V.B.F., Struth, G.: Building program construction and verification tools from algebraic principles. Formal Aspects of Computing 28(2), 265–293 (2016)
6. Armstrong, A., Struth, G., Weber, T.: Kleene algebra. Archive of Formal Proofs (2013)
7. Brink, C.: Power structures. Algebra Universalis 30(2), 177–216 (1993)
8. Brink, C., Rewitzky, I.: A Paradigm for Program Semantics: Power Structures and Duality. CSLI Publications (2001)
9. Brookes, S.: A semantics for concurrent separation logic. Theoretical Computer Science 375 (2007)
10. Conway, J.H.: Regular Algebra and Finite Machines. Chapman and Hall (1971)
11. Damm, W., Harel, D.: Lscs - breathing life into message sequence charts. Formal Methods in System Design 19(1), 45–80 (2001)
12. Dongol, B., Gomes, V.B.F., Struth, G.: A program construction and verification tool for separation logic. In: Hinze, R., Voigtländer, J. (eds.) MPC 2015. LNCS, vol. 9129, pp. 137–158. Springer (2015)
13. Foster, S., Struth, G.: Regular algebras. Archive of Formal Proofs (2014)
14. Gautam, N.: The validity of equations of complex algebras. Arch. Math. Logik Grundl. Mat. 443, 117–124 (1957)
15. Goldblatt, R.: Varieties of complex algebras. Annals of Pure and Applied Logic 44, 173–242 (1989)
16. Gomes, V.B.F., Struth, G.: Program construction and verification components based on Kleene algebra. Archive of Formal Proofs (2016)
17. Gomes, V.B.F.G., Guttmann, W., Höfner, P., Struth, G., Weber, T.: Kleene algebra with domain. Archive of Formal Proofs (2016)
18. Gomes, V.B.F.G., Struth, G.: Modal Kleene algebra applied to program correctness. In: Fitzgerald, J.S., Heitmeyer, C.L., Gnesi, S., Philippou, A. (eds.) FM 2016: Formal Methods - 21st International Symposium, Limassol, Cyprus, November 9-11, 2016, Proceedings. Lecture Notes in Computer Science, vol. 9995, pp. 310–325 (2016)

19. Grätzer, G., Whitney, S.: Infinitary varieties of structures closed under the formation of complex structures. Colloquium Mathematicae 48, 1–5 (1984)
20. He, J., Hoare, C.: Unifying theories of programming. Prentice Hall (1998)
21. Hoare, C.A.R., Hussain, A., Möller, B., O'Hearn, P., Petersen, R.L., Struth, G.: On locality and the exchange law for concurrent processes. In: Katoen, J., König, B. (eds.) CONCUR 2011 - Concurrency Theory - 22nd International Conference, CONCUR 2011, Aachen, Germany, September 6-9, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6901, pp. 250–264. Springer (2011)
22. Hoare, C., Jifeng, H.: Unifying theories of programming (1998)
23. Hoare, T., Möller, B., Müller, M.: Tracelets and specifications. Tech. Rep. 2017-01, Dept of Informatics, University of Augsburg (2016)
24. Hoare, T., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene algebra and its foundations. J. Log. Algebr. Program. 80(6), 266–296 (2011)
25. Hoare, T., van Staden, S., Möller, B., Struth, G., Zhu, H.: Developments in concurrent Kleene algebra. J. Log. Algebr. Meth. Program. 85(4), 617–636 (2016)
26. Horn, A., Alglave, J.: Concurrent Kleene Algebra of partial strings. arXiv.org (Jul 2014)
27. Main, M.: A powerdomain primer — a tutorial for the bulletin of the EATCS 33. Tech. Rep. CU-CS-375-87 (1987). Paper 360, Univ. Colorado at Boulder, Dept of Computer Science (1987), `http://scholar.colorado.edu/csci_techreports/360`
28. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - A Proof Assistant for Higher-Order Logic, LNCS, vol. 2283. Springer (2002)
29. O'Hearn, P.W., Petersen, R.L., Villard, J., Hussain, A.: On the relation between concurrent separation logic and concurrent Kleene algebra. J. Log. Algebr. Meth. Program. 84(3), 285–302 (2015)
30. Petri, C.A.: Communication with automata. Tech. Rep. RADC TR 65-377, RADC, Research and Technology Division, New York (1966)
31. Winskel, G.: On powerdomains and modality. Theoretical Computer Science 36, 127–137 (1985)