Developments in Concurrent Kleene Algebra

Tony Hoare[1], Stephan van Staden[2], Bernhard Möller[3], Georg Struth[4]
Jules Villard[5], Huibiao Zhu[6], and Peter O'Hearn[7]

[1]Microsoft Research, Cambridge, United Kingdom
[2]ETH Zurich, Switzerland
[3]Institut für Informatik, Universität Augsburg, Germany
[4]Department of Computer Science, The University of Sheffield, United Kingdom
[5]Department of Computing, Imperial College London, United Kingdom
[6]Software Engineering Institute, East China Normal University, China
[7]Facebook, United Kingdom

Abstract. This report summarises recent progress in the research of its co-authors towards the construction of links between algebraic presentations of the principles of programming and the exploitation of concurrency in modern programming practice. The research concentrates on the construction of a realistic family of partial order models for Concurrent Kleene Algebra (aka, the Laws of Programming). The main elements of the model are objects and the events in which they engage. Further primitive concepts are traces, errors and failures, and transferrable ownership. In terms of these we can define other concepts which have proved useful in reasoning about concurrent programs, for example causal dependency and independence, sequentiality and concurrency, allocation and disposal, synchrony and asynchrony, sharing and locality, input and output.

1 Introduction

Concurrency has many manifestations in computer system architecture of the present day. It is provided in networks of distributed systems and mobile phones on a world-wide scale; and on a microscopic scale, it is implemented in the multicore hardware of single computer chips. In addition to these differences of scale, there are many essential differences in detail. As in other areas of basic scientific research, we will initially postpone consideration of these differences, and try to construct a mathematical model which captures the essence of concurrency at every scale and in all its variety.

Concurrency also has many manifestations in modern computer programming languages. It has been embedded into the structure of numerous new and experimental languages, and in languages for specialised applications, including hardware design. It is provided in more widely used languages by a choice of thread packages and concurrency libraries. Further variation is introduced by a useful range of published concurrency design patterns, from which a software

architect can select one that reconciles the needs of a particular application with the particular hardware available.

Concurrency is also a pervasive phenomenon of the real world in which we live. A general mathematical model of concurrency shares with the real world the concept of an object engaging together with other objects in events that occur at various points in space and at various instants in time. It also shares the principle of causality, which states that no event can occur before an event on which it causally depends, as well as a principle of separation, which states that separate objects occupy separate regions of space. It is these principles that guide definitions of sequential and concurrent composition of programs in a model of CKA. They provide evidence for a claim that CKA is an algebraic presentation of common-sense (non-metric) spatio-temporal reasoning, similar to that formalised in various ancient and modern modal logics.

1.1 Domain of Discourse

The construction of a model of a set of algebraic laws consists of three tasks. The first task is the definition of the domain of discourse (carrier set of the algebra). Any element of the domain is a value that may be attributed to any of the variables occurring in any of the laws. It is a mathematically defined structure, describing at some level of abstraction the relevant aspects of the real or conceptual world to which the algebra is applied. The second task is the interpretation of each operator of the algebra as a mathematical function, with its arguments and its result in the domain of discourse. Finally, there is the proof that the laws of the algebra are true for any attribution of values in the domain of discourse to all the variables in each of the equations.

It is instructive and convenient to introduce a series of three domains, each one including all the elements of the next. The most comprehensive domain is that of specifications, which describe properties and behaviour of a computer system while executing a program. They may be desirable properties of programs that are not yet written, or undesirable properties of a program that is still under test. Formally, a specification is just a set containing traces of all executions that satisfy the property. It may be expressed in any meaningful mathematical notation, including arbitrary set unions, and arbitrary intersections, and even complementation. A most important quality of a specification is that it should be comprehensible; and therefore it should be accompanied by an informal explanation that makes it so. That is a precept that we hope to follow in this presentation.

The second domain consists of programs. A program can be regarded as a precise description of the exact range of all its own possible behaviours when executed by computer. It is expressed in a highly restricted notation, namely its programming language. The language excludes negation and other operators that are incomputable in the sense of Turing and Church. As a result, a program text can be input directly by computer, and (after various mechanised transformations) it can be directly executed to cause the computer to produce just one of the behaviours specified. Inefficiency of implementation is another good reason

for omission from a programming language of the more general operators useful in a specification. For example, intersection is usually excluded, even though it is the most useful operator for assembling large sets of design requirements into a specification.

At the third level, a single trace, produced for example by a single program test, describes just one particular execution of a particular program at a particular time on a particular computer system or network. The execution itself is also modelled as a lower-level set, consisting of events that occurred during that execution, including events in the real world environment immediately surrounding the computer system. A trace is effectively a singleton set in the domain of programs, so it cannot be composed by union; but it can still be composed either sequentially or concurrently with another trace, and (in our partial order model) the result is still a single trace. The composition operators are easily proved to satisfy the laws of CKA.

There is close analogy between this classification of domains for concurrent programming and the classification of the standard number systems of arithmetic - reals, rationals, and integers. For example, the operators of a programming language share the same kind of elementary algebraic properties as arithmetic operators - distribution, association and commutation, with units and zeroes.

The analogy can be pursued further: our modelling methods are also similar to those used in the foundations of arithmetic. For example, the reals are defined as downward-closed sets of rationals (Dedekind cuts). The operators at each level are then defined in terms of the operators at the lower level. For example, an operator like addition on reals is defined as the set obtained by adding each rational from the first real operand to each rational from the second operand. This construction is known as 'lifting to sets', and we will use it to lift individual traces to the domain of programs and specifications that describe them.

The constructions at the foundations of arithmetic show that the operators of all the number systems obey the same algebraic laws, or nearly so. Our models are designed to do the same for the laws of programming, as expressed in a Concurrent Kleene Algebra.

1.2 Contracts and Counterexamples

Models play an essential role in the development of theories and the practical use of an algebra in mathematics. They provide evidence (a counterexample) for the invalidity of an inaccurately formulated conjecture, explaining why it can never be proved from a given set of algebraic laws. In pure logical and algebraic research, such evidence proves the independence of each axiom of the algebra from all the others. For purposes of counterexample generation, appropriate selection from a family of simple models can be more useful and more efficient than repeated use of a single realistic model that is more complicated. An experienced mathematician is familiar with a wide range of models, and selects the one most likely to serve current purposes. However, the research reported here seeks realism rather than simplicity of its model.

Discovery of counterexamples is also a primary role of models of programming. A counterexample consists in a trace of program execution which contains an error; it thereby demonstrates falsity of the conjecture that a program is correct. An automatic test case generator should obviously concentrate on finding such counterexamples. It should also indicate where the errors have been detected, and where they cannot have occurred. The information should be provided in a form that guides human judgement in diagnosing the error, tracing where it has occurred, and deciding whether it should be corrected or worked around.

The definition of what counts as an error, and of where it is to be attributed, can be formalised as a contract at the interface between one part of the program and another. For each of its participants, a contract has two sides. One side is a description of the obligations, which any of the other participants may expect to be fulfilled. An example is the post-condition of a method body, which every call of the method will rely on to be true afterwards. The other side is a description of the requirements which each participant may expect of the behaviour of all the other participants taken together. An example is the precondition of the method body. Every calling program is required to make this true before the call, and the method body may rely on this as an assumption.

In addition to violation of contracts, there are various kinds of generic error, which are universally erroneous, independent of the purposes of the program. Examples familiar to sequential programmers are undefined operations, overflows, and resource leakages. Concurrency has introduced into programming several new classes of generic error, for example, deadlock, livelock, and races (interference). To deal with these errors, we need new kinds of contract, formulated in terms of new concepts such as dependency, resource sharing, ownership, and ownership transfer. We also need to specify dynamic interactions (by synchronisation, input, or output) between a component of a concurrent program and its surrounding environment.

A full formal definition (semantics) of a programming language will specify the range of generic errors which programs in the language are liable to commit. The semantics itself may be regarded as a kind of contract between the implementer and the user of the language, and they will often allocate responsibility for errors that occur in a running program. For example, syntax errors and violation of type security are often avoided by compile-time checks, and the implementer undertakes to ensure that a program which contains any such errors will not be released for execution, even in a test.

Conversely, for certain intractable errors, the programmer must accept the responsibility to avoid them. In the case of a violation occurring at run time, the language definition may state explicitly that the implementer is freed of all responsibility for what does or does not happen afterwards. For example, in the case of deadlock, nothing more will happen. Or worse, the error may even make the program susceptible to malware attack, with totally unpredictable and usually unpleasant consequences.

The inclusion of contractual obligations in a model lends it an aspect of deontic logic, which has no place in the normal pursuit of pure scientific knowledge.

However, it plays a vital role in engineering applications of the discoveries of science.

1.3 Semantics

There are four well-known styles for formalising the definition of the meaning of a programming language. They are denotational, algebraic, operational and deductive (originally called axiomatic). They are all useful in defining a common understanding for the design and exploitation of various software engineering tools in an Integrated Development Environment (IDE), and for defining sound contracts between them.

A model of the laws of programming plays the role of a *denotational* semantics (due to Scott and Strachey) of a language which obeys the laws. The denotation of each program component is a mathematical structure, which describes program behaviour at a suitable level of abstraction. The first examples of such a model were mathematical functions, mapping an input value to an output value. Later examples included concurrent behaviour, modelled as sets of traces of events. We follow the later examples, and extend them to support the discovery and attribution of errors in a program. The denotations therefore provide a conceptual basis for the design and implementation of testing tools, including test case generators, test trace explorers, and error analysers.

The laws themselves present an *algebraic* semantics (advocated, for example by Bergstra and his colleagues) of the same abstract programming language. Algebra is useful in all forms of reasoning about programs, and the proofs are often relatively simple, both for man and for machine. The most obvious example is the use of algebra to validate the transformation of a program into one with the same meaning, but with more efficient executions. An algebraic semantics is therefore a good theoretical foundation for program translators, synthesisers and optimisers.

The rules of an *operational* semantics (due to Plotkin and Milner) show how to derive, from the text of a program plus its input data, the individual steps of just a single execution of the program. This is exactly what any implementation of the language has to do. The rules thereby provide a specification of the correctness of more efficient methods of implementation, for example, by means of interpreters written in the same or a different language, or compilers together with their low-level run-time support.

The *deductive* semantics (attributed to Hoare) gives proof rules for constructing a proof that a program is correct. Correctness means that no possible execution of the program contains an error. Some of the errors, like an overflow, a race condition or a deadlock, are generic errors. Others are violations of some part of a contract, for example an assertion, written in the program itself. A deductive semantics is most suitable as a theoretical basis for program verifiers, analysers and model checkers, whose function is to prove correctness of programs.

When the full range of tools, based on the four different formalisations of semantics, are assembled into an IDE, it is obviously important that they should be mutually consistent, and provably so. It is common for the tools to communicate

with each other by passing annotated programs between them. The programs are often expressed in a common verification-oriented intermediate language like Boogie designed and implemented by Leino. The semantics of this common language must obviously be rigorously formalised and understood by the designers of all the tools. As described above, the semantics needs to be formalised in different ways, to suit the purposes of different classes of tool. The mutual consistency of all the forms of semantics establishes confidence in the successful integration of the tools, by averting errors at the interface between them. Ideally, this proof can be presented and checked, even before the individual tools are written.

An easy way to prove consistency of two different formalisations is to prove one of them on an assumption of the validity of the other. For example, Hoare and Wehrman describe how the laws of the algebraic semantics can be derived rather simply from a graphical denotational model. Similarly, Hoare and van Staden show that the rules of the operational semantics, as well as the rules of the deductive semantics, can be derived from the same algebraic laws of programming. In combination, these proofs ensure that all the models of the laws satisfy the rules of all three of the other kinds of semantics.

In fact, most of the laws can themselves be derived in the other direction, either from the rules of the operational or from the rules of the deductive semantics, or even from both. For example, the principal law of concurrency (the exchange law) is derivable either from the deductive separation logic rules for concurrency formalised by O'Hearn, or from the Milner transition rules which define concurrency in CCS. This direction of derivation gives convincing evidence that our laws for concurrency are consistent with well-established understanding of the principles of concurrent programming. Similar mutual derivations are familiar from the study of propositional logic, where the rules of natural deduction are derived from Boolean algebra, and vice versa.

2 The Laws of Programming

The laws of programming are an amalgam of laws obtained from many sources: relational algebra (Tarski), regular languages (Kleene), process algebras (Brookes and Roscoe, Milner, Bergstra), action algebra (Pratt) and Concurrent Kleene Algebra (Hoare et al.). The pomset models of Gischer and others have also provided inspiration.

An earlier introduction to the laws for sequential programming is (Hoare et al., Laws of Programming). This was written for general computer scientists and professional software developers. It contains simple proofs that the laws are satisfied by a relational model of program execution. Unfortunately, the relational model does not extend easily to concurrency.

The purpose of this section is to list a comprehensive (but not complete) selection of the laws applicable to concurrent programming. The laws are motivated informally by describing their consequences and utility. The informal description of the operators gives the most general meaning of each of them,

when applied to programs and specifications. Several of them do not apply to traces.

The model described in section 3 offers a choice of definitions for many of the operators. Any combination of the choices will satisfy the laws. The choice is usually made by a programming language definition; but in principle, the choice could be left as a parameter of an individual test run of a program.

2.1 The Basic Operators

Basic Commands

1 (skip) does nothing, because there is nothing it has been asked to do.

$\top$ is a program whose behaviour is totally undetermined. For example, it might be under control of an undetected virus. Other names for this behaviour are abort (Dijkstra), CHAOS (in CSP) and havoc in Boogie.

$\bot$ is a program with no executions. For example, it might contain a type error, which the compiler is required to detect. As a result, the program is prevented from running.

Binary Operators

Sequential composition $p\,;\,q$ executes both p and q, where p can finish before q starts. It is associative with unit 1, and has $\bot$ as zero.

Concurrent composition $p \,|\, q$ executes both p and q, where p and q can start together, proceed together with mutual interactions, and finally they can finish together. The operator is associative and commutative with unit 1 , and has $\bot$ as zero.

Choice $(p \cup q)$ executes just one of p or q. The choice may be determined or influenced by the environment, or it may be left wholly indeterminate. The operator is associative, commutative and idempotent, with $\bot$ as unit.

Refinement

The refinement relation $p \Rightarrow q$ is reflexive and transitive, i.e., a pre-order. It means that p is comparable to q in some relevant respect. For example, p may have less traces, so its behaviour is more deterministic than q, and therefore easier to predict and control. The three operators listed above are covariant (also called monotone or isotone) in both arguments with respect to this ordering. The ordering has $\bot$ as bottom, $\top$ as top and $\cup$ as lub. For further explanation of refinement, see section 2.2.

Distribution

All three binary operators distribute through choice.

Sequential and concurrent composition distribute through each other, as described by the following analogue of the exchange (or interchange) law of Category Theory:

$$(p \mid q)\,;\,(p' \mid q') \Rightarrow (p\,;\,p') \mid (q\,;\,q')$$

For further explanation of the exchange law, see section 2.3.

Iterations

The sequential iteration $p*$ performs a finite sequential repetition of p, zero or more times.

The concurrent iteration $p!$ performs a finite concurrent repetition of p , zero or more times.

Residuals

The weakest precondition q -; r (Dijkstra) is the most general specification of a program p which can be executed before q in order to satisfy specification r. The weakest precondition consequently cancels sequential composition (and vice-versa), but the cancellation is only approximate in the refinement ordering:

$$(q \text{ -; } r)\,;\,q \Rightarrow r \qquad \text{and} \qquad p \Rightarrow (q \text{ -;} (p\,;\,q))$$

The specification statement p ;- r (due to Back and Morgan) is the most general specification of a program q that can be executed after p in order to satisfy specification r.

p -| r the magic wand (due to O'Hearn and Pym) is similar to the above for concurrent composition.

Notes:

1. the result of the residuals is a specification rather than a program. Residuals are in general incomputable. That is why the residual operations are excluded from programming languages.
2. The constants $\top$ and $\bot$, and the operators of iteration and choice, are not available in the algebra of traces.

2.2 Refinement

The refinement relation $p \Rightarrow q$ expresses an engineering judgement, comparing the quality of two products p and q. By convention, the better product is on the left, and the worse one on the right. For example, the better operand p on the left may be a program with less possible executions than q. Consequently, if q is also a program, it is more non-deterministic than p, and so more difficult to predict and control. If p is a program and q is a specification, the refinement relation means that p meets the specification q, in the sense that everything that p can do is described by the specification q. And if they are both specifications, it means that p logically implies q. Consequently p places stronger constraints on an implementation, which can be more difficult to meet.

Refinement between programs may also account for failures and errors. For example, if p is a program that has the same observable behaviour as q , but q contains a generic programming error that is not present in p, this may be the

grounds for a judgment that p refines q. In other words, a program can be improved by removing its programming errors, but otherwise leaving its behaviour unchanged. Dually, if p is a specification, it is made weaker (easier to meet) by strengthening the obligation which it places on its environment. Thus the meaning of refinement is relative to the contracts between the components of a program, and between the whole program and its environment.

A more precise interpretation for refinement is usually made in a programming language definition. But a testing tool might allow the definition to be changed, to reflect exactly the purposes of each test.

2.3 The Exchange Law: $(p \,|\, q)\,;\,(p' \,|\, q') \;\Rightarrow\; (p\,;\,p') \,|\, (q\,;\,q')$

The purpose of this law is made clear by describing its consequences, which are to relate a concurrent composition to one of its possible implementations by interleaving. The law also permits events to occur concurrently, and requires dependent events like input output to occur in the right order. Inspection of the form of the law shows that the left hand side of the law describes a subset of the possible interleavings of the atomic actions from the two component threads $(p\,;\,p')$ and $(q\,;\,q')$ on the right hand side. This subset results from a scheduling decision that the two semicolons shown on the right hand side will be 'synchronised' as the single semicolon on the left.

The algorithm for finding an interleaving uses the recursive principle of 'divide-and conquer'. The interleavings $(p \,|\, q)$ before the semicolon on the left are formed from the two first operands p and q of the two semicolons on the right. The interleavings $(p' \,|\, q')$ after the semicolon on the left are formed from the two second operands p' and q' of the two semicolons on the right. Every execution of the left hand side is the sequential composition of a pair of executions, one from each of $(p \,|\, q)$ and $(p'|q')$. Each such execution achieves synchronisation of the two semicolons on the right, but it places no other constraint on the interleavings. (The constraints are specified in the definition of sequential composition).

By introduction and elimination of the unit 1, the exchange law can be adapted to cases where the term to be transformed has only two or three operands. The following three theorems are called frame laws:

$$p\,;\,q \Rightarrow p \,|\, q \qquad \text{(frame law 0)}$$

$$p\,;\,(q \,|\, r) \Rightarrow (p\,;\,q) \,|\, r \qquad \text{(frame law 1)}$$

$$(p \,|\, q)\,;\,r \Rightarrow p \,|\, (q\,;\,r) \qquad \text{(frame law 2)}$$

By commuting the operands of concurrent composition, the first frame law gives a weak principle of sequential consistency:

$$q\,;\,p \Rightarrow p \,|\, q, \qquad \text{from which, by covariance and idempotence,}$$

$$q\,;\,p \;\cup\; p\,;\,q \Rightarrow p \,|\, q$$

If p and q are atomic commands, then the left hand side of the above conclusion

shows the only two possible interleavings of their concurrent combination on the right hand side. A strong principle of sequential consistency would allow the conclusion to be strengthened to an equation (but only in the case of atomic commands). However, we will continue to exploit the weaker formulation of the principle.

When there are larger numbers of atomic commands in a formula, the exchange law can be used, in conjunction with commutation, association and distribution, to reduce the formula to a normal form in which the outer operator is union and the inner operator is sequential composition. The technique is to use the exchange law to drive the occurrences of concurrent composition to the atoms, and then apply the weak principle of sequential consistency given above.

For example, from the frame laws we get:

$$p\,;\,(q\,;\,r\ \cup\ r\,;\,q) \Rightarrow (p\,;\,q)\,|\,r \qquad \text{and} \qquad (p\,;\,r\ \cup\ r\,;\,p)\,;\,q \Rightarrow r\,|\,(p\,;\,q)$$

By commutation, distribution, covariance and idempotence, we can combine these to an analogue of Milner's expansion theorem for CCS:

$$p\,;\,q\,;\,r\ \cup\ p\,;\,r\,;\,q\ \cup\ r\,;\,p\,;\,q \Rightarrow (p\,;\,q)\,|\,r$$

This theorem remains valid when there are synchronised interactions between the concurrent commands. When an interleaving $\cdots p\,;\,q;\cdots$ violates a synchronisation constraint that p must follow q, the definition of sequential composition will ensure that this interleaving takes the value $\bot$, which is the unit of choice. This particular interleaving is thereby excluded from the left hand side of the theorem.

3 A Diagrammatic Model

The natural sciences often model a real-world system as a diagram (graph) drawn in two dimensions: a space dimension extends up and down the vertical axis, and a time dimension extends along the horizontal axis. We model what happens inside a computer in the same way. The fundamental components of the model are objects, which are represented by lines (trajectories) drawn from left to right on the diagram. An object has a unique identifier (name or address) associated with it at its allocation. This may be used just like a numeric value in assignments and communications.

Examples of objects are variables (local or shared), semaphores (for exclusion or synchronisation), communication channels (buffered or synchronised), and threads. These classes of object are often built into a programming language; but in an object-oriented language they can be supplemented by programmed class declarations.

The lines representing two or more objects may intersect at a point, which represents an atomic event or action in which the given objects participate simultaneously. Examples of events are allocations and disposals of an object, assignments or fetches of a variable, input or output of a message, seizures or

releases of a semaphore, and forking or joining of threads. An example of multiple participation is an atomic assignment, which involves fetches from many variables and assignments to one or more target variables, together with the thread that contains the assignment.

The line for each object passes through the time-ordered sequence of events in which the object engages. In the case of a thread object, the ordering of events within a thread is often called program order. It seems reasonable to require that no event can occur without participation of exactly one thread.

In the diagram for a Petri net, an event is drawn as a transition, in the form of a box or a bold vertical line. Extending the same convention, we will represent participation of an object in the transition as a line which passes straight through the transition. This contrasts with an allocation of a new object whose line begins at the transition, or with a disposal in the case of a line which ends at it. The other Petri net component (a place) represents choice; it is therefore not needed in the diagram for a single trace, for which all choices have already been made.

An arrow is defined as a pair of consecutive points on the same line. It is drawn with its source on the left and its target on the right. An arrow is labelled by a primitive constant predicate of an assertion language. For example, in standard separation logic, the primitive is a pair written (say) $101 \mapsto 27$, where the constant 101 is the unique identifier of the object, and 27 is the value that is held by the object between the event at the source of the arrow and the event at its target.

Arrows are classified as either local or global. The source and target of a local arrow must be events that involve the same thread, called its current owner: violation of this rule of locality is an error. In a language like occam, the compiler is responsible for detecting this error, and making sure that the program is not executed. However in a language like C this check would be too difficult, and it is not required. Instead, violation of locality is attributed as an error of the program.

An object is defined to be local if all its arrows are local, and volatile (shared) if all its arrows are global. An object with both kinds of arrow is one whose ownership may change between the event at the tail and the event at the head of any one of its global arrows. The distinction between local and global arrows is familiar from Message Sequence Charts. The local arrows representing concurrent tasks are drawn downwards, and global arrows are drawn between points on the vertical lines. The points represent calls, call-backs, returns, and other communications between the tasks.

In a diagram of program execution, an instant of time (real or virtual) can be drawn as a vertical coordinate which crosses just one arrow in the line for each object that is allocated but not yet disposed at that instant. The collection of labels on the arrows which cross a vertical coordinate describes the state of the entire system at the given instant. A global arrow denotes a message which has been buffered between the tail event of the arrow and its head. An arrow of a volatile object is effectively a special case of a message. A local arrow crossing the coordinate represents the value held in the computer memory allocated to

the owning object. The state of the entire local memory at the relevant instant is the relation whose pairs are $(loc,\ val)$, where the label on the crossing arrow is $loc \mapsto val$. This is necessarily a function, because no line can cross a coordinate twice: that would involve a backward crossing somewhere in between.

In a diagram of program execution, a point in space can be drawn as a horizontal coordinate separating the threads above it from the threads below it. The set of global arrows which cross the coordinate in either direction give a complete account of the dynamic interactions between the threads that reside on either side of the coordinate. They must all be global arrows. There is no significance attached to the vertical ordering of the horizontal coordinates. That is why concurrent composition commutes.

The important concept of causal dependency (happens before) is defined in terms of arrows. A causal chain is a sequence of arrows (taken usually from different object lines), in which the head of each arrow is the same point as the tail of the next arrow (if any). Occurrence of an event on a causal chain is a (necessary) cause of all subsequent events on the same chain; and it is dependent on all earlier events on the chain. Obviously, no event can occur before an event which it depends on, or after an event that depends on it. This is represented by the left-to-right direction of drawing the arrows.

In summary, the primitive concepts of our geometry are lines and points at which the lines meet. Arrows are defined and classified as local or global, and they are given labels. In terms of these primitives we define vertical and horizontal coordinates, system state, ownership and transfer of ownership, and causal dependency. The concept of synchrony can be defined as mutual dependency, and the concept of 'true' concurrency can be defined in the usual way as causal independence.

3.1 Decomposition of Diagrams

A diagram in plane geometry can be decomposed into segments in two ways, either horizontally or vertically. A horizontal segment (slice) contains the entire lines of a group of related objects, which interact by participating jointly with each other in their events. This segmentation is useful in analysing the behaviour of individual objects from the same class, or groups of interacting objects from the same package of classes.

A vertical segment is similarly separated from its left and right neighbours by two vertical coordinates, representing the initial and the final instants of time. The segment contains all events in the diagram which occurred between the two instants. This form of segmentation is useful in analysing everything that happens during a particular phase in the execution, for example, a method call.

A third form of decomposition mirrors the syntactic structure of the program whose execution is recorded in a trace. Each segment (called a tracelet) contains all the events that occurred during execution of one of the branches of the abstract syntax tree of the program; the tracelets for two syntactically disjoint commands of the program will have disjoint sets of events, as they do in reality.

Inside a diagram, the tracelet is surrounded by a perimeter, with vertical and horizontal sides. The west and east sides of the perimeter are segments from two vertical coordinates, and the north and south sides are segments of two horizontal coordinates. Consequently, the tracelet for a sequential composition $p\,;\,q$ is split vertically into two smaller tracelets, one for p and one for q, with no dependency of any event in p on any event in q; similarly, the tracelet for a concurrent composition is split horizontally, with no local arrows crossing the split. The whole plane is tiled by these splits, like a crazy paving. The tiles are often drawn as rectangles, but this is not necessary.

An arrow with its source outside the box and its target inside is defined as an input arrow; and an output arrow is defined similarly. The local input arrows represent the portion of local state (called the initial statelet) which is passed to the tracelet on entry. By convention, these arrows enter the box on the west side. Similarly, the local output arrows represent the final statelet, and leave the box on the right side. The global arrows may cross the north or the south sides of the box, as convenient. They represent dynamic interactions that take place with the environment of the tracelet between the start and the finish of its execution.

A fourth form of segmentation splits a tracelet into three segments, sharing just a single event. One segment contains all events that the shared event causally depends on. A second segment contains all events that depend on the shared event; and the remaining segment contains all remaining events, which are irrelevant to its occurrence.

The diagrammatic representation of the trace described in this section is intended to be helpful to the user of a visual debugging tool, by conveying an understanding of what has gone wrong in a failed test, and what can be done about it. For example, the segmentation into tracelets will give the closest possible indication of where in the source program an error has been detected. In a visual tool, a hover of the mouse on the perimeter of the tracelet should highlight the command in the original source program whose execution is recorded in the tracelet.

Similarly, the causal segmentation gives clear access to the events which may have caused the error: to prevent the error, at least one of these will have to be changed. When the culprit has been detected and corrected, the segment that is dependent on it contains all the events that may have been affected by the change. The remaining events in the third segment that are causally independent of the error could be greyed out on a display of the trace.

3.2 Refinement

We represent an error that is detected inside a tracelet by colouring its perimeter. We attribute the errors as described in section 2.2. If the error is attributed to the program being executed, the perimeter is drawn in red, or if it is attributed to the environment of the tracelet, it is marked blue. Where necessary, a single point can be coloured. For example, evaluation of an assertion to false is marked red, whereas a false assumption is marked blue. If no error is detected, a normal

black perimeter is drawn. A black perimeter with no points inside represents the execution of the SKIP command, which literally does nothing.

The refinement relation $p \Rightarrow q$ between tracelets p and q is defined by looking only at their events, and also at the colour of their two perimeters. The definition deliberately ignores the internal structure of tracelets within p or q. Validity of the refinement means that the diagram of one operand are just an isomorphic copy of the diagram in the other, and that the perimeter of p has a lower colour than that of q in the natural ordering, with blue below black and red above it. The definition of the isomorphism can be weakened by ignoring much of the internal content of the tracelet. However, the labels on the arrows that cross the perimeter must be preserved, and so must the causal dependencies between these arrows.

The laws of programming require observance of the following principles in colouring of perimeters. The first three principles state the obvious fact that a tracelet inherits all the errors that are contained in any of its subtracelets; but if it contains both a red and a blue error, a somewhat arbitrary decision states that the blue dominates. This is required by the zero laws for $\perp$: it is certainly justified when the bottom denotes a program with no executions.

1. If a tracelet contains a tracelet with a blue perimeter, it also has a blue perimeter.
2. Otherwise, if it contains a tracelet with a red perimeter, it also has a red perimeter.
3. Otherwise, both operands are black, and the whole tracelet has a black perimeter too.

Further rules are introduced in the definition of the two composition operators.

4. Any failure to observe the rules of sequential composition colours the perimeter blue.
5. Any failure to observe the rules of concurrent composition colours the perimeter red, except in the case that principle 1 requires it to be blue.

There is a choice of reasonable meanings for sequential composition. In the strongest variant, every event of the second operand must be dependent on every event of the first operand. This is an appropriate definition for sequential compositions which occur within a single thread. In most programming languages, this is the only kind of sequential composition that can be written in the program. But if strong composition is applied to a multi-threaded trace, it requires that all the threads pass the semicolon together, as in PRAM model of lock-step program execution.

The weakest variant of sequential composition involves the minimum of synchronisation. The principle is simply that no event of the first operand can depend on any event of the second operand. Violation of this principle would make it impossible to complete execution of the first operand before the second operand starts: this was quoted informally in section 2 as the general defining condition for sequential composition.

This definition is weak enough to allow the reordering optimisations that are commonly made in modern compilers for widely used languages. When applied to multi-threaded programs, it allows each thread to pass the semicolon at a different time.

Turning to concurrent composition, its weakest definition imposes only the condition that no local arrow can cross its north or south sides. A more realistic definition has to make the occurrence of deadlock into a programming error. More formally, the condition states that there is no dependency cycle that crosses from an event of one operand to an event of the other. An exception may be made to allow synchronised communication between an outputting and an inputting thread, as in CSP.

The principle that a local arrow cannot cross between threads ensures that in any correct trace there is no interference by one thread with the values of a local variable of another thread. Thus separation logic is a valid method of reasoning about concurrent programs, even in the presence of extra features like synchronisation, atomicity, and communication. This claim still needs to be checked in detail.

That concludes our informal description of a diagrammatic model for the algebra of traces. Our description has been analytic (decompositional). It is presented as a set of principles that are applied to test whether a given fully decomposed and annotated trace has been correctly decomposed, and whether its errors have been correctly attributed, according to the five principles above. This is in contrast to the usual approach of denotational semantics, which is compositional (synthetic): the denotation of the result of each operation is fully defined in terms of the denotations of its operands. The contrast between the decompositional and compositional interpretations is similar to the analytic and synthetic readings of a set of recursive syntactic equations of a context-free language.

The problem with such denotational definitions is that they are too prescriptive of all the details of the model. This is because every needed property of every aspect of the operator has to be deducible from its definition. In a decompositional presentation, each aspect of an operator can be described separately, and as weakly as desired. Indeed, the weakness is often desirable, because interesting variations of the operator can be identified, classified, and left for later choice.

The problem with the analytic approach is to decide when enough principles have been given. We suggest that the relevant criterion is simply that all the laws of programming are provably satisfied by the given collection of principles. Section 1.3 has presented evidence that the laws are sufficient as a foundation for reasoning about programs, and for the design of programming tools, which analyse, implement and verify them.

4 Sketch of a Formal CKA Model

4.1 Graphs and Tracelets

Definition 4.1 Given a set EV of *events* and a set AR of *arrows*, a *graph* is a structure $H = (E, A, s, t)$ where $E \subseteq \mathrm{EV}$, $A \subseteq \mathrm{AR}$ and $s, t : A \to E$ are total functions yielding *source* and *target* of an arrow. A *tracelet* is a pair $tr = (H, F)$ where $H = (E, A, s, t)$ is a graph, called the *overall trace*, and $F \subseteq E$ is a distinguished set of events, called the *focus* of tr and denoted by $foc(tr)$. The pairwise disjoint sets of *input, output* and *internal* arrows of the tracelet are given by

$$\begin{aligned} a \in in(tr) &\Leftrightarrow_{df} t(a) \in F \wedge s(a) \notin F \ , \\ a \in out(tr) &\Leftrightarrow_{df} s(a) \in F \wedge t(a) \notin F \ , \\ a \in int(tr) &\Leftrightarrow_{df} s(a) \in F \wedge t(a) \in F \ . \end{aligned}$$

As mentioned in Section 3, arrows are classified as local and global, but in this section we ignore the distinction.

We want to combine tracelets by connecting outputs of one tracelet to inputs of another. For separation we require the events of tr_1 and tr_2 to be disjoint. Moreover, the combination is meaningful only if both tracelets have the same overall trace. More precisely, consider tracelets tr_1, tr_2 with disjoint focuses but same overall trace H, and an arrow a in $out(tr_1) \cap in(tr_2)$. Then a is automatically an internal arrow of the tracelet $(H, foc(tr_1) \cup foc(tr_1))$. If a carries values of some kind, we view the combination as transferring these values from the source event of a in tr_1 to the target event of a in tr_2.

Definition 4.2 Two tracelets tr_1, tr_2 are *combinable* if $H_1 = H_2$ and $foc(tr_1) \cap foc(tr_2) = \emptyset$. Then their *join* is $tr_1 \sqcup tr_2 =_{df} (H_1, F_1 \cup F_2)$. Clearly, $tr_1 \sqcup tr_2$ is a tracelet again.

Since disjoint union is associative, also $\sqcup$ is associative. In the set of all tracelets with a common overall trace H the *empty tracelet* $\square_H =_{df} (H, \emptyset)$ with empty focus is the unit of $\sqcup$.

4.2 Tracelets and Colours

In Sect. 3.2 we have presented the idea of accounting for errors by colours. Formally we use *tiles*, i.e. pairs (tp, c) with a tracelet tp and a colour c. For abbreviation we represent the colours red, black and blue by the values $\bot, 1, \top$ ordered by $\bot \leq 1 \leq \top$.

For combining scores, we use the *summary* operator $\circ$ defined by the table at the right. Obviously, this operator is commutative and idempotent and has 1 as its unit which is indivisible, i.e., $c \circ c' = 1$ implies $c = 1 = c'$. The operator is also covariant w.r.t. $\leq$. Finally, it is also associative since it coincides with the supremum operator on the lattice induced by the second ordering $1 \preceq \top \preceq \bot$ (we are not going to use that ordering any further, though).

$\circ$	$\bot$	1	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
1	$\bot$	1	$\top$
$\top$	$\bot$	$\top$	$\top$

The *refinement relation*, a partial order between tiles, is given by

$$(p, c) \Rightarrow (p', c') \Leftrightarrow_{df} p = p' \wedge c \leq c' \ .$$

Tiles are composed by joining their tracelet parts and summarising their colours together with additional error information the joined trace may provide. For sequential and parallel composition ; and | this information is computed by two operators $\downarrow$ and $\uparrow$ mapping pairs of traces to colours.

We can realise $\downarrow$ and $\uparrow$ using binary relations R, R' that must hold between the events of joined traces. For combinable traces p, q we set

$$p{\downarrow}q =_{df} \begin{cases} 1 \text{ if } foc(p) \times foc(q) \subseteq R \ , \\ \bot \text{ otherwise} \ , \end{cases} \qquad p{\uparrow}q =_{df} \begin{cases} 1 \text{ if } foc(p) \times foc(q) \subseteq R' \ , \\ \top \text{ otherwise} \ . \end{cases}$$

Here R and R' are any of the relations listed for sequential and parallel composition, respectively, at the end of Section 3.

By definition, these operators satisfy $p{\downarrow}q \leq 1$ and $1 \leq p{\uparrow}q$. Moreover, $\uparrow$ is commutative. Using the indivisibility of 1 one sees that $\downarrow$ and $\uparrow$ distribute through trace join, i.e., $(p \sqcup q){\downarrow}r = p{\downarrow}r \circ q{\downarrow}r$ etc.

Definition 4.3 Sequential and parallel composition of tiles with combinable traces are defined as follows:

$$\begin{aligned} (p, s) \,;\, (p', s') &=_{df} (p \sqcup p', \ s \circ s' \circ p{\downarrow}p') \ , \\ (p, s) \mid (p', s') &=_{df} (p \sqcup p', \ s \circ s' \circ p{\uparrow}p') \ . \end{aligned}$$

Theorem 4.4 *The operators* ; *and* | *are associative and* | *is commutative. Moreover they satisfy the frame and exchange laws. Under the additional assumptions* $\Box_H {\downarrow} p = 1 = p{\downarrow}\Box_H = p{\uparrow}\Box_H$*, the tile* $(\Box_H, 1)$ *is a shared unit of* ; *and* | *in the set of all tracelets with common overall trace* H*.*

The latter assumptions mean that the empty tracelet is error-free, which is reasonable. By this Theorem we have provided a recipe for constructing specific models to meet specific purposes, with a prior guarantee that the model will satisfy the laws.

4.3 Programs and Lifting

A *program* is a set of tiles that is downward closed w.r.t. refinement $\Rightarrow$.

We already have presented the idea that operators on programs should arise by pointwise lifting of the corresponding operators on tiles. Of course, this makes sense only if also the laws for tiles lift to programs.

A sufficient condition for this is bilinearity, viz. that every variable occurs exactly once on both sides of the law. Examples are associativity, commutativity and neutrality in the case of equations and the frame and exchange laws in the case of refinement laws.

While it is clear what equality means for programs, i.e., downward closed sets of tiles, there are several ways to extend refinement to sets. We choose the following definition:

$$p \Rightarrow p' \quad \text{iff} \quad \forall\, t \in p : \exists\, t' \in p' : t \Rightarrow t' \ .$$

By this, a program p refines a specification p' if each of its tiles refines a tile admitted by the specification. Downward closure implies that $\Rightarrow$ in fact coincides with inclusion $\subseteq$ between programs. Hence the set of 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.

The operators at the tile level can be lifted to downward closed sets by forming all possible combinations of the tiles in the operands and closing the result set downward. For instance, $p\,;p'$ is defined as the downward closure $dc(\{t\,;t' \,|\, t \in p, t' \in p'\}) =_{df} \{r \,|\, r \Rightarrow t\,;t' \text{ for some } t \in p, t' \in p'\}$, and analogously for the other operators. The lifted versions of covariant tile operators are covariant again, but even distribute through arbitrary unions of programs. Therefore, by the Tarski-Knaster fixed point theorem, recursion equations have least and greatest solutions.

Moreover, it can be shown that with this construction bilinear refinement laws lift to programs. We illustrate this for the case of the frame law $p\,;p' \Rightarrow p \,|\, p'$.

Assume $r \in p\,;p'$. By the above definition there are $t \in p, t' \in p'$ such that $r \Rightarrow t\,;t'$. Since the frame law holds at the tile level, we have $t\,;t' \Rightarrow t \,|\, t'$. Moreover, $t \,|\, t'$ is in $dc(\{t \,|\, t' \,|\, t \in p, t' \in p'\}) = p \,|\, p'$ and we are done.

By this and the second part of Theorem 4.4 the program

$$1 =_{df} dc(\{(\square_H, 1) \,|\, H \text{ a graph}\})$$

is a shared unit of the liftings of ; and | to programs.

4.4 Residuals

By the distributivity of lifted covariant operators and completeness of the lattice of downward closed programs the residuals mentioned in Section 2.1 are guaranteed to exist. They can be defined by the Galois connections

$$\begin{array}{lll} p \Rightarrow q \text{ -; } r & \text{iff} & p\,;q \Rightarrow r \ , \\ q \Rightarrow p \text{ ;- } r & \text{iff} & p\,;q \Rightarrow r \ . \end{array}$$

This independent characterisation is necessary, since these operators cannot reasonably be defined as the liftings of corresponding ones at the tile level. An analogous definition can be given for the magic wand -| . The semi-cancellation laws of Section 2.1 are immediate consequences of these definitions. Residuals enjoy many more useful properties, but we forego the details.