A Calculus of Space, Time, and Causality: its Algebra, Geometry, Logic

Tony Hoare[1], Georg Struth[2], and Jim Woodcock[3]

[1] University of Cambridge, t-tohoar@outlook.com
[2] University of Sheffield, g.struth@sheffield.ac.uk
[3] University of York, jim.woodcock@york.ac.uk

Abstract. The calculus formalises human intuition and common sense about space, time, and causality in the natural world. Its intention is to assist in the design and implementation of programs, of programming languages, and of interworking by tool chains that support rational program development.

The theses of this paper are that Concurrent Kleene Algebra (CKA) is the algebra of programming, that the diagrams of the Unified Modeling Language provide its geometry, and that Unifying Theories of Programming (UTP) provides its logic. These theses are illustrated by a formalisation of features of the first concurrent object-oriented language, Simula 67. Each level of the calculus is a conservative extension of its predecessor.

We conclude the paper with an extended section on future research directions for developing and applying UTP, CKA, and our calculus, and on how we propose to implement our algebra, geometry, and logic.

Keywords: Concurrent Kleene Algebra (CKA) · Concurrent Separation Logic (CSL) · Calculus of Communicating Systems (CCS) · Communicating Sequential Processes (CSP) · Action Algebra · Discrete Euclidian Geometry · Cartesian Coordinates · Unified Modeling Language (UML) · Unifying Theories of Programming (UTP)

Foreword from Tony

I am deeply grateful to the organisers and attendants at this meeting of UTP 2019 in celebration of my 85th birthday. I could not ask for a better birthday present. I hope you enjoy hearing my presentation as much as I have enjoyed writing it with my co-authors.

Twenty years ago I had the privilege of delivering a presentation at the first World Congress on Formal Methods, FM'99, in Toulouse [74, 75]. I gave a talk entitled "Theories of Programming: Top-down and Bottom-up and Meeting in the Middle" [32]. I claimed that denotational semantics was at the top, operational semantics at the bottom, with algebraic semantics as a unifying link in the middle. My talk today is on the same subject. At the top, discrete geometric diagrams provide the denotations, and its rules of reasoning provide both

an operational semantics (e.g., Milner's CCS) and a verification semantics (e.g., O'Hearn's CSL).

Next year is the 60th anniversary of my invention of Quicksort [29], and I propose to retire from active personal research. This will be my farewell appearance at an international conference. I have taken advantage of the opportunity to present a sort of testament, reporting the results of the last ten years of my research. I hope you will find some of them helpful or inspiring for your next ten years of research into the theory and application of UTP.

* * *

1 Introduction

The purpose of this paper is to give independent descriptions of the features of programming languages, independently of the language in which they are embedded. It offers many examples, but does not make any recommendation on their selection or rejection. Above all it shuns any attempt to define a complete language. To achieve its purpose, it exploits the power of elementary algebra, geometry, and logic.

The basic insight of the paper is that causality, space, and time have the same meaning inside a computer as in the natural world outside it. These concepts do not require a mathematical semantics, but they will be used to give one.

The operators of sequential composition of operands executed in the same region of space at different intervals of time is an equal partner of concurrent composition of operands executed in the same interval of time in disjoint regions of space. They differ primarily in their domain of definition.

The other insight is that a program is a predicate. Each phrase of a structured program defines exactly, and in minute detail, the set of all its traces of its execution. Each trace records all the events that were performed, both internal to the computer and in interaction with its environment. The execution may be on any computer, at any time and at any place, and with any resolution of its internal non-determinism.

Specifications are predicates that extend the range of operators available in a programming language by including operators of the predicate calculus. Disjunction is most useful in the design phase of development to postpone decisions between design options until more information is available. Conjunction combines requirements in the specification phase of design. In general, its implementation is indescribably inefficient for a non-deterministic program.

Choice in a program is defined by disjunction of predicates, either finite or infinite. An internal choice introduces (demonic) non-determinism into the execution of programs. In making an external choice (for example, by a conditional or a guarded command), the surrounding environment (either the rest of the program or the external world) can prevent selection of one or more of the options.

The trace is written in a subset of the same language as its program, but avoids all forms of choice. It is pictured as an abstract syntax tree (AST): its

nodes represent component phrases of a structured program, and each leaf represents a unique execution of a basic commands of the program. Relations between traces are defined by structural induction on their ASTs, particularly the refinement relation and the function mapping each node of a trace to its leaves.

The language of traces contains only composition operators. Their events are just the union of the events of their operands. For example '**;**' describes sequential execution of its operands at separate instants in time; another operator '**|**' describes their concurrent execution at separate locations in space.

The operators are distinguished by different constraints on their implementation or on their use. For '**;**', the implementer of the language must ensure that no event in the first operand has a cause in the second operand. For '**|**' the user of the language must ensure that the trace contains no cyclic chain of causation between its events.

The operators of the trace calculus are lifted to sets in the usual way. The sets are downward closed wrto the refinement order. Thus if a program describes a trace, it also describes all the refinements of it. In any phase of development, this is what justifies replacement of an abstract program by one of its refinements.

Other operators can be defined by stronger constraints. For example, the CSP operator '$\rightarrow$', which separates the guard of a guarded command from its body, requires every event in the body to have a cause in the guard, and to be a cause of every event in the body [31]. The result of a concurrency operation of CSP must not contain a cyclic causal chain, because that would deadlock the implementation. For '$|||$' in CSP and for separating conjunction in CSL, there must be no causal link between the operands. This obviously prevents both races and deadlocks.

General negation is an incomputable operator. It must be included in a specification language to permit simple descriptions of safety and security by negating a description of what must not happen. But it cannot be included in any general-purpose programming language. It is therefore included in specification languages like UTP [34], CSL [37], and Concurrent Refinement Algebra (a foundation for rely/guarantee reasoning about concurrent programs) [14].

UTP is a special-purpose descriptive logic for specifying traces. It describes primarily the causal links crossing the interfaces between the phrases of a program, and abstracts from the internal events. Each interface is a labelling function from the links of the interface to a value that passed between from its cause to its effect. Examples are the function that represents a region of memory shared between the left and right operand of '**;**', and the trace of messages passing between concurrent operands, restricted to channels that they share.

2 Extended Summary

2.1 The calculus

1. An ordering relation is defined on traces by induction on its AST. Its left operand in general requires the same or fewer real processors for execution than the right operand.

2. It specifies a single decision step taken by a timesharing scheduler, which implements concurrency by sequentially interleaving the threads that appear as its operands. Details are given in [39] and [36].
3. The order is a precongruence, i.e., a preorder that makes all operands of the program monotonic (covariant). Its symmetric closure is an equivalence relation that satisfies all the equational axioms of a CKA [37] that are expressible without '+', which means choice in CKA.
4. The algebra is lifted to sets by precongruence closure, standard for converting a precongruence into an order [48, 8]. The same lifting is used by Dedekind to lift fractions to real numbers [9]. It is analogous to the equivalence class construction due to Frege and used by Russell in the definition of natural numbers.
5. Choice in a program is defined as set union. The ordering relation on traces lifts to refinement (set inclusion) on programs.
6. Further operators can be defined on sets, both algebraically and by proof rules; for example: iterators (e.g., the Kleene $*$), and residuals for all operators (e.g., weakest prespecifications ($/$) and postspecifications ($\backslash$) [33]), and fixed points [73].
7. A claim that the calculus can be applied to programs is supported by evidence of the large body of program proofs from either the axiomatic proof rules of CSL or the operational rules of CCS [52]. Hoare triples and Milner transitions are given algebraic definitions as a simple refinement in the calculus. The rules of both CSL and CCS are then proved in the calculus. They are three-line proofs in [36].
8. The definitions given above to triples and transitions are the same. By the reflexivity of equality, the theories differ only in notation! Verification logic and operational semantics have been unified in the closest possible way.

The rest of this summary has been included here only for background information. The text summarised has been excluded from the published article for reasons not unconnected with time and space.

2.2 Causality, space, and time

1. Causality denotes a familiar relation between events in the real world, and requires no mathematical semantics. Any such semantics (such as that given for Petri nets [63, 62]) can be considered as a scientific theory applicable to the natural world. Causality is represented graphically by drawing an arrow from the causing event to the caused event (its effect) at its head.
2. The essential property of causation is that no event can occur before its cause. But they can occur at the same time.
3. The collection of arrows between the events of a trace forms a directed graph, with events represented by points.
4. Arrows are classified as either vertical or horizontal. A vertical arrow is drawn between the successive events that occurred at the same location in space; events are performed by an object allocated at a given location of memory, for example a variable or a communication port of a channel.

5. A horizontal arrow is drawn between simultaneous events, each performed by a different object (e.g., a thread and a variable). The full set of simultaneous actions is known as a transition [62], or as a transaction [25].
6. The graph for a trace is segmented into subgraphs, one for each node and for each leaf of its AST. A leaf is the only occupant of its segment.
7. The graph of a sequential node is split by a horizontal cut between its operands. The only coordinates that it cuts are vertical. A concurrent node is split by a vertical cut, which cuts only horizontal coordinates.
8. The graph for any node of the AST is contained within a rectangular box, whose edges are cuts. The input arrows of the box are defined as those with only their heads in the box; and those with only tails are called its outputs. Its internal arrows have both ends in the box.
9. Conventionally, all vertical input arrows enter the box at its top edge, and vertical output arrows leave at the bottom edge. Horizontal input arrows enter at the left edge and horizontal output arrows leave it at the right.

2.3 Geometry

1. For purposes of program debugging, a box can be displayed as a diagram of discrete plane geometry. Its two axes represent time and space, and its points represent events.
2. A vertical coordinate is a chain of arrows containing the complete history of events performed by a single object. Examples of objects include variables, threads, communication ports, messages.
3. A horizontal coordinate is a set of arrows connecting all its points. Each point on it is shared by a distinct vertical coordinate. Examples include multiple assignments, communications, synchronising fences, object allocation, and disposal.
4. Any pair of vertical or of horizontal coordinates is mutually parallel in the sense of Euclid: they have no point in common.
5. As in Cartesian geometry, every point in a diagram is the unique element of the intersection of a horizontal and a vertical coordinate.
6. Further examples are presented in [35] and alternatively in [54]. They are called Sequence Diagrams in UML [59], or Message Sequence Charts in SDL [43].
7. In a debugging tool, each error revealed by the trace should be highlighted. The tool should also provide a means for navigating backwards from an error, travelling along vertical and horizontal coordinates to its direct and indirect causes (time-travel debugging [51]); also forwards to its results whenever possible after a non-fatal error.

2.4 Logic

1. According to its standard semantics, a proposition of predicate calculus describes all observations (aka, valuations) that satisfy it. An observation is a total function from all syntactically possible variable identifiers to their observed values.

2. For the predicates of CSL and of UTP, the observations are only partial functions, whose domain (aka footprint) is the set of all free variables of the proposition. Negation of an assertion preserves its footprint.
3. Separating conjunction in CSL describes concurrent composition of programs. It is defined only if the footprints of its two operands are disjoint, and neither is undefined. The footprint of a disjunction is the union of that of footprints of its operands.
4. The footprint of a predicate in UTP is defined in terms of the box diagrams of the trace described. For its top edge, it is the collection of unique names of the vertical coordinates that cross a horizontal boundary of the box diagram. For the bottom edge of the diagram, the names are annotated by a dash.
5. The value of an object observed at these edges is that which was assigned (or left unchanged) by the event at the tail of the cut arrow.
6. In the Circus variant of UTP [76, 61], a built-in variable *tr* stands for a record of the history of all input-output events that are recorded from the beginning of the entire trace. Each input of a message lies on the same horizontal interface as the output of the message.
7. Alternative conventions are often more intuitive. For example, the trace can be represented by a finite state diagram in which the nodes are annotated by an invariant that describes the values of the variables throughout the interval between its initial and its final horizontal coordinate.

3 The Calculus

Our terms are traces of execution of a program written in a language that includes events (to be defined later), a constant **1**, and two binary operators of sequential composition ('**;**') and concurrent composition ('**|**'). The context-free syntax of the terms of the calculus is:

$$\begin{array}{ll} \langle term \rangle & ::= \mathbf{1} \mid \langle event \rangle \mid \mathbf{(}\langle term \rangle \langle operator \rangle \langle term \rangle \mathbf{)} \\ \langle operator \rangle & ::= \text{`}\mathbf{;}\text{'} \mid \text{`}\mathbf{|}\text{'} \end{array}$$

By structural recursion we define the events of a term to be the set of events recorded in the whole trace:

$$\begin{array}{ll} events(\mathbf{1}) = \{\} & \\ events(k) = \{k\} & [\text{if } k \text{ is an event}] \\ events(p\,\mathbf{;}\,q) = events(p\,\mathbf{|}\,q) = events(p) + events(q) & \end{array}$$

where $\{k\}$ is the singleton set containing only k, and $+$ is disjoint union of sets, ensuring that each event only occurs once in the abstract syntax tree. If the operands of $+$ are not disjoint, then the result is undefined. This fact is expressed by the *ok* predicate of UTP [34], which satisfies the axioms

$$\begin{array}{ll} ok(s + t) \equiv (s \cap t = \{\}) \wedge ok(s) \wedge ok(t) & \\ ok(s) & [\text{if } s \text{ is a singleton set or empty}] \end{array}$$

Let p be a term $(r\,;s)$ and let e be a member of $events(r)$ and let f be a member of $events(s)$. Then the pair (e,f) is said to be sequentially separated within p. The set of pairs within p that are so separated is defined by

$$ssep(p) = \{\, events(r) \times events(s) \mid r\,;s \text{ is a subterm of } p \text{ (or } p \text{ itself)} \,\}$$

This can also be defined axiomatically without set notation by structural recursion:

$$ssep(1) = \{\}$$
$$ssep(p\,;q) = ssep(p) + ssep(q) + events(p) \times events(q)$$
$$ssep(p \mid q) = ssep(p) + ssep(q)$$

A definition of concurrent separation $csep(p \mid q)$ is similar.

The structure of two terms is compared by a relation $\leq$ between them.[4] It means that q has a denser sequential control structure than p. For example, $p\,;q \leq p \mid q$. (Similar relations can be defined with respect to $csep$.)

$$p \leq q \;\widehat{=}\; ssep(p) \subseteq ssep(q) \wedge events(p) = events(q)$$
$$p \equiv q \;\widehat{=}\; (p \leq q) \wedge (q \leq p)$$

This definition of the fundamental ordering relation is formulated in terms of syntax. This may violate one tradition of programming language semantics: that syntax and semantics should be totally separated. The syntactic definition is strongly welcomed in other traditions. It gives the strongest possible model of the calculus. Algebraists call it a word algebra, category theorists call it an initial or free algebra, and computer scientists call it fully abstract. A proof that CKA satisfies all these definitions is given in [47]. A concept with three or more equivalent definitions is usually important in mathematics, for example the axiom of choice in logic.

3.1 The algebra of traces

The axioms of the calculus are just those basic axioms of CKA [37] that can be expressed in the syntax; those involving choice (written as $+$), repetition (*) and residuation ($/$ and $\backslash$) are omitted. They will be re-introduced shortly.

Theorem 1.

1. $\leq$ *is a preorder* [reflexive and transitive]
2. *If* $q \leq p$ *then* $p \mid r \leq p \mid r$ *and* $r \mid q \leq r \mid p$ *and* $p\,;r \leq p\,;r$ *and* $r\,;q \leq r\,;p$ [monotonicity]
3. $(p\,;q)\,;r \equiv p\,;(q\,;r)$ *and* $(p \mid q) \mid r \equiv p \mid (q \mid r)$ [associativity]

[4] In UTP [17], refinement between pointwise relations is written as $P \sqsubseteq Q$ (or equivalently $Q \leq P$), and defined by $[\, Q \Rightarrow P \,]$. It asserts that every behaviour of Q is also a behaviour of P.

4. $p\,;\mathbf{1} \equiv p \equiv \mathbf{1}\,;p$ *and* $p\,|\,\mathbf{1} \equiv p \equiv \mathbf{1}\,|\,p$ [unit]

5. $(p\,|\,q)\,;(p'\,|\,q') \leq (p\,;p')\,|\,(q\,;q')$ [interchange]

The first two laws echo the familiar laws for equality, formulated by Euclid and Leibniz. They permit a refinement to be used as a single-directional substitution rule in algebraic reasoning. A standard structural induction from the second law says that refinement is preserved when the rule is applied to any sub-term of a given term. The third law allows redundant brackets to be omitted. And the fourth describes the steps that reduce a term to sequential normal form, in which all '$|$' are eliminated.

We obtain four small interchange laws from Theorem 1.5 by substituting units for each of the four variables.

$$p\,;(r\,|\,s) \leq (p\,;r)\,|\,s \qquad q\,;(r\,|\,s) \leq r\,|\,(q\,;s)$$

$$(p\,|\,q)\,;s \leq p\,|\,(q\,;s) \quad (p\,|\,q)\,;r \leq (p\,;r)\,|\,q$$

Two tiny interchange laws are derived by a second such substitution in the first line above:

$$p\,;r \leq p\,|\,r \quad q\,;r \leq r\,|\,q$$

The interchange axiom models the decisions of a timesharing scheduler operating at run time or at compile time. Its purpose is to reduce the number of actual processors needed for execution of a program below what it has explicitly called for. In combination with the equational axioms, it may be used as a single step in the reduction of any term of the calculus to a normal form that has no '$|$'. The equational axioms are used first on each step to select which '$;$'s and which '$|$'s to match to the left hand side of interchange. Different choices will result in different eventual interleavings. Each non-trivial application of interchange increases the membership of *sseq*, so the shuffling process must terminate. The corollaries of the axiom are what finally eliminates the '$|$'s.

3.2 Applications

The simplicity, relevance, and power of the calculus is demonstrated by its application to two well known and widely used theories of programming, separation logic [66, 60] (which includes Hoare Logic) and Milner's CCS [52]. The Hoare triple $\{p\}\,q\,\{r\}$ [30] is interpreted as saying that performance of q preceded by p is one of the ways of implementing r: i.e., $p\,;q \leq r$. (This is a generalisation of the original Hoare definition, which required that p and r be restricted to the events that evaluate assertions [72], in a similar manner to weakest prespecifications [33].) From this definition, the proof rules for sequencing and concurrency in CSL (Concurrent Separation Logic) [38]. Simpler proofs (three-liners mostly) are given in [38].

The Milner transition is written $r \xrightarrow{p} q$. In the small-step version of the transition, the program p is restricted to a singleton event. This triple is interpreted as the statement that one of the ways of implementing r is to perform p

first, saving its continuation q for later execution. Algebraically expressed, this is $p\,;q \leq r$, which is the same definition as the Hoare triple. By definition, the two calculi are the same! This claim can be checked by definitional substitution, which translates the defining axioms of each theory into those of the other. The unification is similar to that made by Dirac, when he showed the mathematical identity of the Schrödinger and the Heisenberg formulations of quantum theory with his own.

3.3 The algebra of programs

The behaviour of a program is defined as the set of all traces that can be produced by its execution. The operators are defined by complex product or convolution. Let capital letters stand for sets of traces. Define the operators on the traces by

$$\begin{array}{lcl} \mathbf{1} & \widehat{=} & \{\mathbf{1}\} \\ P\,;Q & \widehat{=} & \{\,(p\,;q) \mid p \in P \wedge q \in Q\,\} \\ P \mid Q & \widehat{=} & \{\,(p \mid q) \mid p \in P \wedge q \in Q\,\} \\ P \sqcap Q & \widehat{=} & P \cup Q \end{array}$$

Nondeterministic choice is defined as set union, and its algebraic properties are familiar from Boolean algebra: it is associative, commutative, and idempotent, and it has the empty set as its unit. Refinement is defined by set inclusion.

Linearity of the axioms in Theorem 1 ensures ensures that the equational properties of traces remain unchanged when they are lifted to sets of terms in the usual way, by the results of [20]. We would also like to remove all undefined terms from the sets. This is done by applying a familiar algebraic construction for turning a preorder into a partial order, namely by the downward closure of the sets, with respect to the preorder:

$$\begin{array}{lcl} P\,;Q & \widehat{=} & \{\,r \mid p \in P \wedge q \in Q \wedge r \wedge ok(r) \leq p\,;q\,\} \\ P \mid Q & \widehat{=} & \{\,r \mid p \in P \wedge q \in Q \wedge r \wedge ok(r) \leq p \mid q\,\} \\ P \leq Q & \widehat{=} & P \subseteq Q \end{array}$$

Further operators can be defined on sets, both algebraically and by proof rules; for example: iterators (e.g., the Kleene $*$), and residuals for all operators (e.g., weakest prespecifications (/)and postspecifications ($\backslash$) [33]), and fixed points [73]. Iteration is defined as the least fixed point of $x = SKIP \vee x \vee x\,;x$. An introduction to these topics is well presented by axioms and proof rules in [64], where a complete algebraic characterisation of iteration includes the elegant equation $p = (p/p)^*$, or equivalently $p = (p \setminus p)^*$, where p is an invariant of the loop. Pratt proved the axiom interdeducible with the proof rules that define either least or greatest fixed-point (depending on the order).

4 Symmetries

In the natural sciences, an experiment is designed to produce a result that all observers of a repeated experiment will agree on, no matter when and no matter

where it is viewed from. The raw observations will obviously be different, but agreement can be reached if the direct description of each raw observation is automatically translatable into a description made by any observer from a different viewpoint at a later time. The translation algorithm is called a symmetry.

It is therefore not surprising that the laws themselves are translatable by the same symmetry, and each translation to gives back either the same law or another one. That gives confidence of the universal applicability of the laws throughout space and time. It should certainly be checked by mathematical proof.

The axioms in Theorem 1 satisfy such symmetries. In the algebra, we model time reversal symmetry by a function v that swaps the arguments of '$;$', space inversion symmetry by a function h that swaps those of '$|$', and space-time symmetry by a function d that interchanges '$;$' and '$|$':

$$\begin{array}{lll} v(p\,;q) = v(q)\,;v(p), & v(p\,|\,q) = v(p)\,|\,v(q) & v(1) = 1 \\ h(p\,;q) = h(p)\,;h(q), & h(p\,|\,q) = h(q)\,|\,h(p) & h(1) = 1 \\ d(p\,;q) = d(p)\,|\,d(q), & d(p\,|\,q) = d(p)\,;d(q) & d(1) = 1 \end{array}$$

As before, '$|$' need not commute. All axioms in Theorem 1 are closed under these symmetries. We explain only the interchange law, which we write as

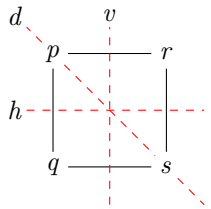
$$\frac{p}{q}\,;\frac{r}{s} \leq \frac{p\,;r}{q\,;s}$$

Applying the symmetries yields

$$\frac{v(r)}{v(s)}\,;\frac{v(p)}{v(q)} \leq \frac{v(r)\,;v(p)}{v(s)\,;v(q)} \quad \frac{h(q)}{h(p)}\,;\frac{h(s)}{h(r)} \leq \frac{h(q)\,;h(s)}{h(p)\,;h(r)} \quad \frac{d(p)\,;d(q)}{d(r)\,;d(s)} \geq \frac{d(p)}{d(r)}\,;\frac{d(q)}{d(s)}$$

The first law holds because *ssep* guarantees that the left-hand and right-hand sides of the two laws have the same points and the same arrows, which are just reversed in direction. The second one holds by a similar argument with respect to *csep*. The third law is valid because d interchanges *ssep* with *csep*, after which the value of *ssep* decreases from right to left. This explains the reversal of $\leq$. Application of d to the monotonicity axioms in Theorem 1 requires this reversal as well.

If p, q, r and s are leaves of the AST, then we can depict the nodes in both sides of the first interchange law as the following black square.



In the symmetric interchange laws, v then reflects the nodes in the vertical axis bisecting the square, h in the horizontal axis, and d in the diagonal axis through

p and s. In fact, h can be generated from d and v as $h = d \circ v \circ d$ and the full symmetry group of the square—reflection in the other diagonal and rotations by 0°, 90°, 180° and 270°—can be generated from these two elements as well. Interchange is therefore invariant under all eight symmetries.

In mathematics, symmetries are admired for their beauty. They arise as properties invariant under some transformation, usually the action of some group. Yet beyond their beauty, they have practical uses too. The symmetries of interchange are preserved by refinement, so any conjecture that does not preserve symmetry can be instantly rejected. Furthermore, every theorem proved automatically generates seven corollaries. Exploitation of symmetries by a proof tool can give further optimisations [13, 41, 68, 21].

5 Future Directions

In the shortest term, the authors plan to publish a journal version of this paper with missing sections restored. It is proposed to apply the syntactic methods of this paper to define features like probabilistic choice and delay commands that are found in Simula 67 [7].

An urgent development of the theory presented here is to model the layers of abstraction that are implemented by a hierarchy of class declarations in an object-oriented language. A layer includes all the subclasses of a class, and shares no resources with any other layer. It is sometimes called a component or a module. Abstraction scales: the very largest systems in worldwide use today could never have evolved without it.

5.1 Unifying Theories of Programming

The major problem facing verification today is that many large systems are written in a combination of languages:

- General purpose, application-oriented (e.g., scripting, discrete-event simulation, network design, security).
- Continuous control in both signal-oriented and equational styles.
- Hardware-oriented (e.g., GPU, FPGA, quantum).

How can we provide a common toolset for them all? Perhaps the algebraic methods introduced in this paper could be used to develop a semantics for existing and future languages, with compatible links for other languages used in the same product.

5.2 Applications of Concurrent Kleene Algebra

Kleene algebra is well known for vast simplifications and generalisations of the proofs of some important theorems. For example, in [45], Kozen gives a completely algebraic proof in KA with tests (KAT) that a program with nested loops

can be reduced to a program with just a single loop and some auxiliary variables (a classic folk theorem), and von Wright gives a very elegant, single-page proof of a theorem for atomicity refinement in action systems [78] that previously had taken Back many pages to accomplish [2]. Equally convincing results for concurrent programs with CKA are so far missing. Examples would include concurrency control or concurrency refinement laws.

KAT [45], Kleene algebras with domain [10, 11], and demonic refinement algebras [78] have been established as abstract semantics and verification methods for sequential programs and linked with concrete program semantics such as relations or predicate transformers. Hayes and co-workers have recently developed concurrent refinement algebras, which are inspired by CKA, and support rely-guarantee style reasoning with shared-variable concurrent programs [27, 28] and CCS/CSP-style reasoning [52, 31]. Similar applications in the semantics of concurrent programming languages remain to be explored. Many of the approaches mentioned have led to verification components with interactive theorem provers, notably with Isabelle/HOL [1, 24, 18]. For CKA, such components are under development.

5.3 Implementing the calculus

In the immediate future, we are planning that a research team at York will engage in developing a library of theories in our theorem prover, Isabelle/UTP [17, 19, 77], which is an implementation of UTP in Isabelle/HOL [58].

We hope to recruit collaboration with other centres of excellence to develop compatible extensions of the mechanisation in other proof tools, for example Coq [3], Lean [55], Maude [6], Agda [4], and FDR [23].

We will support the geometric presentation of the calculus using Eclipse [71], defining the abstract graphical syntax with the Eclipse Modeling Framework (EMF), its concrete syntax with the Graphical Modeling Framework (GMF), and transforming the models created with the language into Isabelle/UTP theories using the Epsilon model transformation tool [44]. This will follow the approach set out in [53, 79], where the graphical RoboChart language [53] is managed within the Eclipse-based RoboTool environment [67] and transformed to CSP [31], Prism [46], and Isabelle/UTP [19].

5.4 Object orientation and UTP

Object-oriented programming is the only known programming paradigm that makes writing massive software applications reasonably manageable, maintainable, and scalable. The research presented in this paper uses classes and their objects as the principal technique for abstraction. A full treatment of object orientation requires additional abstraction techniques that provide encapsulation and information hiding, supporting structuring and re-use of classes through inheritance (perhaps including multiple inheritance), behavioural subtyping, and polymorphism; it would permit the use of dynamic dispatch as a way of selecting different implementations.

There is already much significant work on OO in UTP, but an elegant and integrated treatment in UTP remains a significant ambition. Existing achievement include the following. Santos et al. [69] present a general theory of object orientation in UTP. Naumann et al. [57] give a semantics to class hierarchies and how to refactor them for representation independence. Cavalcanti et al. [5] report on unifying OO classes and CSP-like processes in *OhCircus*, an object-oriented extension of the UTP-based *Circus* multi-paradigm language [76, 61], with a formalisation of method calls and their refinement. Ramos et al. [65] give a semantics to active classes in UML-RT, the real-time profile for UML, via a mapping into *Circus*. Duran et al. [12] present a strategy for compiling classes, inheritance, and dynamic binding, following the compilation strategy for Dijkstra's guarded command language using refinement algebra in UTP [34, chap.6]. Silva et al. [70] present the laws of programming for object orientation with reference semantics and Gheyi et al. [22] give a complete set of object modelling laws for Alloy [42]. Finally, Zeyda and his colleagues [80] present a modular theory of object orientation in higher-order UTP [34, chap.9], all mechanised in Isabelle/UTP [17, 19, 77].

A huge challenge is to harmonise and extend these existing UTP theories to provide a simple and widely accepted treatment of all the main features of object orientation.

To test and evaluate the theory of classes, other concurrent programming design patterns should be specified experimentally as class declarations. At each layer, the programmer needs a way of specifying new behavioural type systems checkable at compile time and proof systems detectable at run time. Their purpose is to avoid violations of the protocols whose universal observance by user programs is required by the design pattern. Type inference algorithms should be specifiable within the algebra, perhaps by restricting refinement rules to Horn clauses [40]. They can then be directly executed by exhaustive tree search. The same restriction is also made in functional languages, but other syntactic restrictions ensure determinacy, so that tree search is not necessary.

5.5 Extensions of the calculus

Probabilistic Kleene Algebra (PKA) [49] and CKA [37] have been combined in Concurrent Probabilistic Kleene Algebra (CPKA) to provide a unified account of nondeterminism, probability, and concurrency, with models in probabilistic automata, modulo probabilistic refinement simulation [50]. This is a natural target for the extension of the algebra, geometry, and logic of our calculus.

A particular application area of great current interest is Cyber-Physical Systems (CPS). They use embedded computers and networks to compute, communicate, and control physical processes. Research in verification in this area has to provide the techniques and tools for checking the correctness of software and hardware platforms with respect to agreed requirements.

The notion of correctness has to be judged against runtime feedback on the validity of assumptions about the environment, and digital twin technology is

being proposed to handle this problem [26]).[5] Fitzgerald et al. [15] describe the beginnings of a generalised theory of CPS design, with an introduction to the formal foundations, methods, and integrated tool chains for CPS. Crucially, models of CPS are inherently heterogeneous and require unification of different languages, design methods, and verification techniques and their tools.

Modal Kleene Algebra (MKA) [10, 11] has recently been used with ordinary differential equations (ODEs) for the verification of hybrid systems, where discrete imperative program behaviour complements continuous physical dynamics [56]. Foster et al. [16] describe a generalisation of the UTP theory of reactive processes [34, chap.7] using abstract trace algebra. This extends the reactive process theory to continuous time traces, where events are replaced by piece-wise continuous functions of physical behaviour, and this gives a model of hybrid systems. A connection between the UTP and MKA is a long-term and very ambitious objective.

6 Conclusion

The long-term practical goal of a theory of programming is to provide a conceptual framework for the design of a coherent set of practical tools for program development. They should cover the features of modern general-purpose programming languages, and also special-purpose languages and design patterns that exploit synergy in the characteristics of particular applications, algorithms and hardware. The tools should cover the entire life cycle of large-scale program evolution, which starts from requirements and specifications, and continues through system architecture, program design, coding, static checking, compilation, optimisation, selective verification, testing, and correction, right up to delivery of the product. The cycle then repeats in subsequent evolution of the delivered product. The coherence of the theory enables the various languages to be used together in the same software architecture. The conceptual framework should ideally be accompanied by tools which give assistance in the life cycle of new special-purpose programming languages likely to emerge in the changing world.

It is comforting that the conceptual framework of causality, space, and time is the same as that of our common-sense world, and of the more advanced theories of modern science.

[5] In this extension of model-based engineering, a digital twin is a virtual model of the system, constructed from formal development artefacts and used throughout the lifetime of the product. This pairing of the virtual and physical worlds allows analysis of data and monitoring of systems to detect problems before they occur, prevent downtime, develop new application opportunities, and plan immediate and long-term behaviour using simulations. Since the virtual model captures the assumptions made about the environment during system development, these assumptions can be tuned to more accurately reflect reality.

Acknowledgements

Parts of this work were funded under EPSRC grants EP/R032351/1 on *Verifiably Correct Transactional Memory* and EP/M025756/1 on *A Calculus for Software Engineering of Mobile and Autonomous Robots*, and by a Royal Society grant on *Requirements Modelling for Cyber-Physical Systems*.

References

1. Armstrong, A., Gomes, V.B.F., Struth, G.: Building program construction and verification tools from algebraic principles. Formal Asp. Comput. **28**(2), 265–293 (2016)
2. Back, R.: A method for refining atomicity in parallel algorithms. In: Odijk, E., Rem, M., Syre, J. (eds.) PARLE '89: Parallel Architectures and Languages Europe, Volume II: Parallel Languages, Eindhoven, 12–16 June, 1989. Lecture Notes in Computer Science, vol. 366, pp. 199–216. Springer (1989)
3. Bertot, Y., Castran, P.: Interactive Theorem Proving and Program Development: Coq'Art The Calculus of Inductive Constructions. Springer (2010)
4. Bove, A., Dybjer, P., Norell, U.: A brief overview of Agda — A functional language with dependent types. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009: 22nd International Conference on Theorem Proving in Higher Order Logics, Munich, 17–20 August 2009. Lecture Notes in Computer Science, vol. 5674, pp. 73–78. Springer (2009)
5. Cavalcanti, A., Sampaio, A., Woodcock, J.: Unifying classes and processes. Software and System Modeling **4**(3), 277–296 (2005)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.L. (eds.): All About Maude — A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic, Lecture Notes in Computer Science, vol. 4350. Springer (2007)
7. Dahl, O., Myhrhaug, B., Nygaard, K.: Simula 67 Common Base Language. Tech. rep., NCC (May 1968)
8. Davey, B.A., Priestley, H.A.: Introduction to lattices and order. Cambridge University Press (1990)
9. Dedekind, R.: Stetigkeit und irrationale Zahlen. Verlag von Friedrich Vieweg und Sohn, Braunschweig (1872)
10. Desharnais, J., Möller, B., Struth, G.: Kleene algebra with domain. ACM Trans. Comput. Log. **7**(4), 798–833 (2006)
11. Desharnais, J., Struth, G.: Internal axioms for domain semirings. Sci. Comput. Program. **76**(3), 181–203 (2011)
12. Duran, A., Cavalcanti, A., Sampaio, A.: A strategy for compiling classes, inheritance, and dynamic binding. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003: International Symposium on Formal Methods Europe, Pisa, 8 14 September 2003. Lecture Notes in Computer Science, vol. 2805, pp. 301–320. Springer (2003)
13. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. Formal Methods in System Design **9**(1/2), 105–131 (1996)
14. Fell, J., Hayes, I.J., Velykis, A.: Concurrent refinement algebra and rely quotients. Archive of Formal Proofs **2016** (2016)

15. Fitzgerald, J.S., Gamble, C., Larsen, P.G., Pierce, K., Woodcock, J.: Cyber-physical systems design: Formal foundations, methods, and integrated tool chains. In: Gnesi, S., Plat, N. (eds.) FormaliSE 2015: 3rd IEEE/ACM FME Workshop on Formal Methods in Software Engineering, Florence, 18 May 2015. pp. 40–46. IEEE Computer Society (2015)
16. Foster, S., Cavalcanti, A., Woodcock, J., Zeyda, F.: Unifying theories of time with generalised reactive processes. Inf. Process. Lett. **135**, 47–52 (2018)
17. Foster, S., Woodcock, J.: Unifying theories of programming in Isabelle. In: Liu, Z., Woodcock, J., Zhu, H. (eds.) Unifying Theories of Programming and Formal Engineering Methods: International Training School on Software Engineering, Held at ICTAC 2013, Shanghai, 26–30 August, 2013. Lecture Notes in Computer Science, vol. 8050, pp. 109–155. Springer (2013)
18. Foster, S., Ye, K., Cavalcanti, A., Woodcock, J.: Calculational verification of reactive programs with reactive relations and Kleene algebra. In: Relational and Algebraic Methods in Computer Science - 17th International Conference, RAMiCS 2018, Groningen, The Netherlands, October 29 - November 1, 2018, Proceedings. pp. 205–224 (2018)
19. Foster, S., Zeyda, F., Woodcock, J.: Isabelle/UTP: A mechanised theory engineering framework. In: Naumann, D. (ed.) Unifying Theories of Programming: 5th International Symposium, UTP 2014, Singapore, 13 May 2014. Lecture Notes in Computer Science, vol. 8963, pp. 21–41. Springer (2015)
20. Gautam, N.D.: The validity of equations of complex algebras. Archiv für mathematische Logik und Grundlagenforschung **3**(3), 117–124 (Sep 1957)
21. Gent, I.P., Petrie, K.E., Puget, J.: Symmetry in constraint programming. In: Rossi, F., van Beek, P., Walsh, T. (eds.) Handbook of Constraint Programming, Foundations of Artificial Intelligence, vol. 2, pp. 329–376. Elsevier (2006)
22. Gheyi, R., Massoni, T., Borba, P., Sampaio, A.: A complete set of object modeling laws for Alloy. In: Oliveira, M.V.M., Woodcock, J. (eds.) SBMF 2009: 12th Brazilian Symposium on Formal Methods, Gramado, 19–21 August 2009. Lecture Notes in Computer Science, vol. 5902, pp. 204–219. Springer (2009)
23. Gibson-Robinson, T., Armstrong, P.J., Boulgakov, A., Roscoe, A.W.: FDR3: A parallel refinement checker for CSP. STTT **18**(2), 149–167 (2016)
24. Gomes, V.B.F., Struth, G.: Modal Kleene Algebra applied to program correctness. In: Fitzgerald, J.S., Heitmeyer, C.L., Gnesi, S., Philippou, A. (eds.) FM 2016: 21st International Symposium on Formal Methods, Limassol, 9–11 November 2016. Lecture Notes in Computer Science, vol. 9995, pp. 310–325. Springer (2016)
25. Gray, J., Reuter, A.: Transaction Processing: Concepts and Techniques. Morgan Kaufmann (1993)
26. Grieves, M., Vickers, J.: Digital twin: Mitigating unpredictable, undesirable emergent behavior in complex systems (excerpt). Tech. rep., University of Michigan (August 2016)
27. Hayes, I.J.: Generalised rely-guarantee concurrency: an algebraic foundation. Formal Asp. Comput. **28**(6), 1057–1078 (2016)
28. Hayes, I.J., Meinicke, L.A., Winter, K., Colvin, R.J.: A synchronous program algebra: A basis for reasoning about shared-memory and event-based concurrency. Formal Asp. Comput. **31**(2), 133–163 (2019)
29. Hoare, C.A.R.: Algorithm 64: Quicksort. Commun. ACM **4**(7), 321 (1961)
30. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(10), 576–580 (1969)
31. Hoare, C.A.R.: Communicating Sequential Processes. Prentice Hall (1985)

32. Hoare, C.A.R.: Theories of programming: Top-down and bottom-up and meeting in the middle. In: Wing et al. [74], pp. 1–27
33. Hoare, C.A.R., He, J.: The weakest prespecification. Inf. Process. Lett. **24**(2), 127–132 (1987)
34. Hoare, C.A.R., He, J.: Unifying Theories of Programming. Prentice Hall (1998)
35. Hoare, T.: Geometric theory of program testing. `www.cl.cam.ac.uk/~carh4/19.Jan.18.Lecture1.pdf`, accessed: 2019-07-11
36. Hoare, T., Mendes, A., ao F. Ferreira, J.: Logic, algebra, and geometry at the foundation of computer science. In: FMTea 2019: Formal Methods Teaching Workshop and Tutorial (2019)
37. Hoare, T., Möller, B., Struth, G., Wehrman, I.: Concurrent Kleene Algebra and its foundations. J. Log. Algebr. Program. **80**(6), 266–296 (2011)
38. Hoare, T., O'Hearn, P.W.: Separation logic semantics for communicating processes. Electr. Notes Theor. Comput. Sci. **212**, 3–25 (2008)
39. 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)
40. Horn, A.: On sentences which are true of direct unions of algebras. Journal of Symbolic Logic **16**(1), 14–21 (1951)
41. Ip, C.N., Dill, D.L.: Better verification through symmetry. Formal Methods in System Design **9**(1/2), 41–75 (1996)
42. Jackson, D.: Alloy: A lightweight object modelling notation. ACM Trans. Softw. Eng. Methodol. **11**(2), 256–290 (2002)
43. Jervis, C. (ed.): ITU-T: Recommendation Z.120 (04/04), Message Sequence Charts (MSC). International Telecommunication Union, Geneva (2004)
44. Kolovos, D., Paige, R., Polack, F.: The Epsilon Transformation Language. In: Vallecillo, A., Gray, J., Pierantonio, A. (eds.) Theory and Practice of Model Transformations. pp. 46–60. Springer (2008)
45. Kozen, D.: Kleene algebra with tests. ACM Trans. Program. Lang. Syst. **19**(3), 427–443 (1997)
46. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 585–591. Springer (2011)
47. Laurence, M.R., Struth, G.: Completeness theorems for pomset languages and Concurrent Kleene Algebras. CoRR **abs/1705.05896** (2017)
48. MacNeille, H.M.: Partially ordered sets. Transactions of the AMS **42**(3), 416–460 (1937)
49. McIver, A., Rabehaja, T.M., Struth, G.: On probabilistic Kleene algebras, automata and simulations. In: de Swart, H.C.M. (ed.) RAMICS 2011: 12th International Conference on Relational and Algebraic Methods in Computer Science, Rotterdam, 30 May–3 June, 2011. Lecture Notes in Computer Science, vol. 6663, pp. 264–279. Springer (2011)
50. McIver, A., Rabehaja, T.M., Struth, G.: Probabilistic concurrent Kleene algebra. In: Bortolussi, L., Wiklicky, H. (eds.) QAPL 2013: 11th International Workshop on Quantitative Aspects of Programming Languages and Systems, Rome, 23–24 March 2013. EPTCS, vol. 117, pp. 97–115 (2013)
51. Microsoft: Time Travel Debugging in WinDbg Preview! `blogs.msdn.microsoft.com/windbg/2017/09/25/time-travel-debugging-in-windbg-preview/`, accessed: 2019-07-01
52. Milner, R.: A Calculus of Communicating Systems, Lecture Notes in Computer Science, vol. 92. Springer (1980)

53. Miyazawa, A., Ribeiro, P., Li, W., Cavalcanti, A.L.C., Timmis, J., Woodcock, J.C.P.: RoboChart: Modelling and verification of the functional behaviour of robotic applications. Software & Systems Modeling (2019)
54. Möller, B., Hoare, T., Müller, M., Struth, G.: A discrete geometric model of concurrent program execution. In: Bowen, J., Zhu, H. (eds.) UTP 2016: International Symposium on Unifying Theories of Programming, 4–5 June 2016, Reykjavik. Lecture Notes in Computer Science, vol. 10134, pp. 1–25. Springer (2017)
55. de Moura, L.M., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean theorem prover (system description). In: Felty, A.P., Middeldorp, A. (eds.) CADE-25: 25th International Conference on Automated Deduction, Berlin, 1–7 August, 2015. Lecture Notes in Computer Science, vol. 9195, pp. 378–388. Springer (2015)
56. Munive, J.J.H., Struth, G.: Verifying hybrid systems with Modal Kleene Algebra. In: Desharnais, J., Guttmann, W., Joosten, S. (eds.) RAMiCS 2018: 17th International Conference on Relational and Algebraic Methods in Computer Science, Groningen, 29 October–1 November 2018. Lecture Notes in Computer Science, vol. 11194, pp. 225–243. Springer (2018)
57. Naumann, D.A., Sampaio, A., Silva, L.: Refactoring and representation independence for class hierarchies. Theor. Comput. Sci. **433**, 60–97 (2012)
58. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002)
59. Object Management Group: OMG: Unified Modeling Language: Superstructure 2.0 (2003)
60. O'Hearn, P.W.: Separation logic. Commun. ACM **62**(2), 86–95 (2019)
61. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Asp. Comput. **21**(1-2), 3–32 (2009)
62. Peterson, J.L.: Petri nets. ACM Comput. Surv. **9**(3), 223–252 (1977)
63. Petri, C.A.: Communication with automata. DTIC Res. Rep. AD0630125, Defense Tech. Inf. Cntr., Fort Belvoir, VA (1966)
64. Pratt, V.R.: Action logic and pure induction. In: van Eijck, J. (ed.) Logics in AI, European Workshop, JELIA '90, Amsterdam, 10–14 September, 1990. Lecture Notes in Computer Science, vol. 478, pp. 97–120. Springer (1991)
65. Ramos, R., Sampaio, A., Mota, A.: A semantics for UML-RT active classes via mapping into Circus. In: Steffen, M., Zavattaro, G. (eds.) FMOODS 2005: 7th IFIP WG 6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems, Athens, 15–17 June 2005. Lecture Notes in Computer Science, vol. 3535, pp. 99–114. Springer (2005)
66. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: 17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22–25 July 2002, Copenhagen. pp. 55–74. IEEE Computer Society (2002)
67. RoboTool: Graphical modelling, validation, and automatic generation of mathematical definitions for proof for RoboChart models. `www.cs.york.ac.uk/robostar/robotool/`
68. Sakallah, K.A.: Symmetry and satisfiability. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, vol. 185, pp. 289–338. IOS Press (2009)
69. Santos, T.L.V.L., Cavalcanti, A., Sampaio, A.: Object orientation in the UTP. In: Dunne, S., Stoddart, B. (eds.) UTP 2006: First International Symposium on Unifying Theories of Programming, Walworth Castle, County Durham, 5–7 February 2006. Lecture Notes in Computer Science, vol. 4010, pp. 18–37. Springer (2006)

70. Silva, L., Sampaio, A., Liu, Z.: Laws of object orientation with reference semantics. In: Cerone, A., Gruner, S. (eds.) SEFM 2008: 6th IEEE International Conference on Software Engineering and Formal Methods, Cape Town, 10-14 November 2008. pp. 217–226. IEEE Computer Society (2008)
71. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework 2.0. Addison-Wesley, 2nd edn. (2009)
72. Tarlecki, A.: A language of specified programs. Sci. Comput. Program. **5**(1), 59–81 (1985)
73. Tarski, A.: A lattice-theoretical fixpoint theorem and its applications. Pacific Journal of Mathematics **5**, 285–309 (1955)
74. Wing, J.M., Woodcock, J., Davies, J. (eds.): FM'99—Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, 20–24 September 1999, Volume I, Lecture Notes in Computer Science, vol. 1708. Springer (1999)
75. Wing, J.M., Woodcock, J., Davies, J. (eds.): FM'99—Formal Methods, World Congress on Formal Methods in the Development of Computing Systems, Toulouse, 20–24 September, 1999 Volume II, Lecture Notes in Computer Science, vol. 1709. Springer (1999)
76. Woodcock, J., Cavalcanti, A.: The semantics of Circus. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002: Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users, Grenoble, 23–25 January 2002. Lecture Notes in Computer Science, vol. 2272, pp. 184–203. Springer (2002)
77. Woodcock, S.F.J., Zeyda, F.: Unifying semantic foundations for automated verification tools in Isabelle/UTP. CoRR **abs/1905.05500** (2019)
78. von Wright, J.: Towards a refinement algebra. Sci. Comput. Program. **51**(1-2), 23–45 (2004)
79. Ye, K., Woodcock, J., Foster, S., Miyazawa, A., Cavalcanti, A.: RoboChart: Formal modelling and verification of the probabilistic behaviour of robotic applications. Tech. rep., University of York (2019)
80. Zeyda, F., Santos, T.L.V.L., Cavalcanti, A., Sampaio, A.: A modular theory of object orientation in higher-order UTP. In: Jones, C.B., Pihlajasaari, P., Sun, J. (eds.) FM 2014: 19th International Symposium on Formal Methods, Singapore, 12–16 May 2014. Lecture Notes in Computer Science, vol. 8442, pp. 627–642. Springer (2014)