SCHOOL OF
COMPUTING

Reasoning about shared-variable concurrency: interactions between research threads

C. B. Jones

TECHNICAL REPORT SERIES

No. CS-TR-1531 December 2019

TECHNICAL REPORT SERIES

No. CS-TR-1531 **December 2019**

Reasoning about shared-variable concurrency: interactions between research threads

C. B. Jones

Abstract

This paper addresses the topic of reasoning formally about concurrent computer programs that execute with variables that are shared between threads. The approach is to attempt to trace the key “insights”that have shaped the research.

Printed and published by Newcastle University,
School of Computing, Urban Sciences Building,
Newcastle upon Tyne, NE4 5TG, England.

Bibliographical Details: Reasoning about shared-variable concurrency: interactions between research threads

Title and Authors: Cliff B Jones

NEWCASTLE UNIVERSITY
School of Computing. Technical Report Series. CS-TR-1531

Abstract: This paper addresses the topic of reasoning formally about concurrent computer programs that execute with variables that are shared between threads. The approach is to attempt to trace the key “insights”that have shaped the research.

About the authors: Cliff B. Jones is currently Professor of Computing Science at Newcastle University. As well as his academic career, Cliff has spent over 20 years in industry. His 15 years in IBM saw, among other things, the creation –with colleagues in Vienna– of VDM which is one of the better known “formal methods”. Under Tony Hoare, Cliff wrote his doctoral thesis in two years. From Oxford, he moved directly to a chair at Manchester University where he built a world-class Formal Methods group which –among other projects– was the academic lead in the largest Software Engineering project funded by the Alvey programme (IPSE 2.5 created the “mural” (Formal Method) Support Systems theorem proving assistant). In 1996 he moved back into industry with a small software company (Harlequin), directing some 50 developers on Information Management projects and finally became overall Technical Director before leaving to re-join academia in 1999 to take his current chair in Newcastle. Much of his current research focuses on formal (compositional) development methods for concurrent systems and support systems for formal reasoning. Cliff is also a Fellow of the Royal Academy of Engineering (FREng), ACM, BCS, and IET. He has been a member of IFIP Working Group 2.3 (Programming Methodology) since 1973.

Suggested keywords: Shared-variable concurrency, History of formal methods, Compositionality

Reasoning about shared-variable concurrency: interactions between research threads

C. B. Jones

December 14, 2019

1 Introduction

This paper addresses the important topic of reasoning formally about concurrent computer programs that execute with variables that are shared between threads. The approach is to attempt to trace the key **"insights"** that have shaped the research. There have been some relatively linear sequences of ideas where research contributors build on preceding work; there have also been periods of strong interaction and friendly competition between adherents of different approaches.

Where dates are useful, they will normally be related to publication dates. In addition to publications, there have been a number of places where real progress has been made with researchers interacting face-to-face: the most influential venue might have been IFIP's Working Group WG2.3 on *Programming Methodology*,[1] further venues include meetings of lecturers at the Marktoberdorf Summer Schools, Schloss Dagstuhl and the UK *Concurrency Working Group*.

Some avenues of concurrency research focus on the avoidance of shared variables — *Process Algebras* and other approaches not addressed in the body of this paper are mentioned in Section 5.2. Despite the challenges that shared-variable concurrent programs present to developers, such programs are both historically important and remain in widespread use.

Even with primitive operating systems, the attempt to keep a CPU busy –whilst slower external devices consumed or delivered data– required care in program design. When there was a single CPU, programs could switch between threads in a way that gave rise to most issues about shared variables. As input/output processors became more independent from the CPU, flags could be set, interrupts generated and buffers filled independently of the program actually written by a developer.

Concurrency issues have become more important over time because of the creation of full-blown time-sharing systems, the emergence of applications that interact with a world external to the computer and multi-core hardware.

[1] With respect to the topics considered in this paper, the most productive period was probably the 1970s/80s but the whole history of WG2.3 deserves closer study.

Developing concurrent software poses many challenges including data races, deadlock and "livelock"[2] — the common cause of these issues is interference: the behaviour of the program that is actually written is influenced by external activities.

1.1 Refresher on reasoning about sequential programs

The key steps in research on reasoning about sequential (non-concurrent) software are described in [Jon03]: Hoare's *Axiomatic basis* paper [Hoa69] is taken as key;[3] Hoare acknowledges the influence of Floyd [Flo67], van Wijngaarden [vW66] and Naur [Nau66]. The trace in [Jon03] back to Turing [Tur49] and von Neumann [GvN47][4] shows a surprising hiatus in progress of this research area between 1949 and 1967.

Fig. 1 comes from a mimeographed draft of Floyd's 1967 seminal paper;[5] it shows that his annotations were attached to arcs in flowcharts. In contrast, Hoare introduced judgements –now referred to as "Hoare triples"– (now written as $\{P\}\quad S\quad \{Q\}$) in which P and Q are assertions (predicates) and S is a program text; Q is a post condition that expresses what the program text achieves providing the pre condition P holds before execution of S.

One reason that the move away from flowcharts was so important is that it points towards a development method instead of a way of checking completed programs. Hoare's approach lends itself to starting with a specification and decomposing it into smaller and smaller sub-problems until each can be achieved by statements of the desired programming language.[6] This "top-down" description might be viewed as idealistic in an environment where most programs evolve over time and were unlikely to have been designed initially in a systematic way but understanding an "ideal" can throw light on other methods.

Hoare axioms (or rules of inference) for a very simple language are given in Fig. 2; the issues in the remainder of this paper can be explained by focussing on the first rule. Such inference rules permit the conclusion of the triple below the horizontal line providing any judgements above the line can be proved. The rule

[2] This term is attributed to Ed Ashcroft whose contributions are covered in Section 2.1.

[3] Hoare is one of the most highly cited computer scientists — although his "CSP" paper has even more citations, [Hoa69] has over 7,000 (GS) citations and has maintained an almost constant level for many years.

[4] Mark Priestly's invited talk at Porto suggests a revision of the assessment of von Neumann's contribution. In particular, Priestly's archive research has precisely identified the letter to Goldstine in which von Neumann proposes "assertion boxes".

[5] Floyd acknowledges Gorn and Perlis as the originators of the ideas — Knuth suggested that this was overly modest.

[6] This only became clear in the published version of [Hoa71b]: an early draft offered a *post facto* proof of the final program and these proofs were extremely hard to check — Hoare revised the paper to show a stepwise development that was much more convincing (the title of the paper was not changed). It is worth noting that Hoare argued at the 1964 *Formal Language Description Languages* conference at Baden-bei-Wien [Ste66] for a language definition style that could leave things undefined. Also [Hoa69, §6] talks about language definition. It could be argued that what has become the cornerstone of 50 years of research into formal development of programs was found during an attempt to solve a different problem (i.e. that of writing a semantic description of a language).

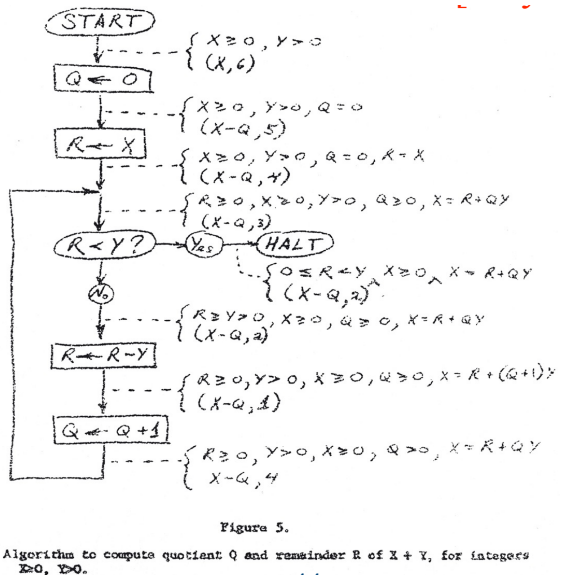


Figure 5.

Algorithm to compute quotient Q and remainder R of X ÷ Y, for integers X≥0, Y>0.

Figure 1: Floyd's hand drawn flowchart (with assertions) of division by successive subtraction

$$\boxed{;}\ \frac{\{P\}\ S_1\ \{Q\} \quad \{Q\}\ S_2\ \{R\}}{\{P\}\ S_1;\ S_2\ \{R\}}$$

$$\boxed{\textbf{while}}\ \frac{\{P \wedge b\}\ S\ \{P\}}{\{P\}\ \textbf{while}\ b\ \textbf{do}\ S\ \textbf{od}\ \{P \wedge \neg\, b\}}$$

$$\boxed{\textbf{if}}\ \frac{\{P \wedge b\}\ S_1\ \{Q\} \quad \{P \wedge \neg\, b\}\ S_2\ \{Q\}}{\{P\}\ \textbf{if}\ b\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \textbf{fi}\ \{Q\}}$$

$$\boxed{\leftarrow}\ \frac{}{\{P[e/x]\}\ x \leftarrow e\ \{P\}}$$

$$\boxed{\textit{consequence}}\ \frac{P' \Rightarrow P \quad Q \Rightarrow Q' \quad \{P\}\ S\ \{Q\}}{\{P'\}\ S\ \{Q'\}}$$

Figure 2: Hoare's axioms

for sequential composition can be viewed as supporting problem decomposition, it can inform the decomposition of the problem below the line indicated by pre condition P and post condition R into finding S_1 and S_2 with their respective pre and post conditions. A useful early survey of Hoare's approach is given in [Apt81, Apt83] a comprehensive and up-to-date survey is [AO19].

Crucially the approach is "compositional" in the sense that the developer of S_1 need only consider the specification P/Q — no awareness is needed of the sibling specification nor that of the overall statement. Later sections below make clear that compositionality is not easily achieved for concurrent programs precisely because of interference between threads. But, for sequential programs, pre and post conditions suffice: they record everything that the developer of a (sub-)program needs to achieve.

For concurrent programs, non-compositional methods were discovered first and they have a significant role in providing tools that analyse finished code. Apart from the ideal of top-down problem decomposition, compositional methods indicate how descriptions of complex systems can be decomposed into understandable pieces. Ideas from compositional methods can also provide useful abstractions for bottom-up approaches.

1.2 Useful background reading on concurrency

There are many technical challenges that have to be faced when designing concurrent programs. These include data races, atomicity, deadlocks, livelocks and fairness. Various programming language constructs have been proposed to help overcome these challenges. Semaphores were an early idea and higher level constructs such as conditional critical sections have been put forward. Readers unfamiliar with these ideas would learn enough from [BA90] to follow the rest of the current paper; [AO91], [Sch97] or [MK99] go much further into formal material.

1.3 Beyond the sequential case

Following on from the success of [Hoa69], Hoare and colleagues looked at a range of extensions to the axiomatic approach (e.g. [Hoa71a, CH72]). His first foray into applying the approach to parallel programs resulted in [Hoa72]. Looking back at the rule for sequential combination discussed here in Section 1.1, the dream would be to find a rule that permitted some simple combination of the pre and post conditions of two threads such as:

$$\boxed{?}\ \frac{\begin{array}{c}\{P1\}\ \ S_1\ \ \{Q1\} \\ \{P2\}\ \ S_2\ \ \{Q2\}\end{array}}{\{P1\ \textbf{and}\ P2\}\ \ S_1 \parallel S_2\ \ \{Q1\ \textbf{and}\ Q2\}}$$

As one would expect, Hoare gives a clear outline of the issues and notes that the above rule works (with logical conjunction in the conclusion) providing that S_1 and S_2 refer only to disjoint variables. The paper [Hoa72] goes on to

investigate ways in which interference can be ruled out (notably by "critical regions"); tackles a number of examples (including a bounded buffer, "dining philosophers" and a parallel version of Quicksort); and includes an extremely generous acknowledgement to the stimulus of Edsger W. Dijkstra.

The first **insight** then is that separation limits interference — with hindsight this might sound obvious but, in addition, an ideal form of parallel inference rule is given against which other proposals can be judged.

This sets the scene for an interesting split in research directions:

- one line of research is to look at explicit ways of reasoning about interference — this avenue is discussed in Section 2
- alternatively, researchers have investigated reasoning about separation even in the more complicated arena of heap variables — Section 3 reviews this approach

Both approaches are clearly important and have spheres of applicability; Sections 2 and 3 discuss the two avenues in roughly historical order; Section 4 outlines fruitful interactions between researchers pursuing the two avenues.

2 Reasoning about interference

Only with the benefit of hindsight did the criterion of compositionality became a key issue (see [dR01]) but, since there is also a historical development, the distinction is used to separate Sections 2.1 and 2.2. Sections 2.3 and 2.4 mention two important –but somewhat orthogonal– detailed issues.

The phrase "non-compositional" might sound negative but "bottom-up" methods that work with finished code have given rise to many useful tools (see below in Section 4) — part of the attraction of tools that work with finished programs is that they can do useful work relatively automatically.

2.1 Non-compositional approaches

A first step[7] towards reasoning formally about interfering threads was made by Ed Ashcroft and Zohar Manna in [AM71].

- Ashcroft had done his PhD at Imperial College (London) under the supervision of John Florentin.[8] Ashcroft was also known for his work on the "Lucid" language with Bill Wadge. Interestingly, Ashcroft supervised Matthew Hennessy's PhD — Hennessey's main research area is process algebras.

[7] A reader who is tempted to view this section in particular as too "linear" should remember that in the 1970s there were fewer active researchers than there are today. Furthermore, this paper is deliberately limited to shared-variable concurrency — subjects like process algebras were progressing in parallel (see Section 5.2).

[8] Florentin and the current author had extensive contacts during the 1960s/70s when the latter worked for IBM.

- Manna's Carnegie PhD was supervised by Floyd (which fact is important below). He made huge contributions to many areas[9] including Temporal Logics. A beautiful technical obituary is [DW19].

The 1971 paper was preceded by a Stanford Tech Report [AM70]. The ideas build mainly on the Floyd approach including an assumption of flowcharts as a presentation of the programs to be justified. There is, in fact, a rather offhand reference to "see also [Hoa69]". Manna proposed the idea of "non-deterministic programs" in [Man70][10] and the 1971 paper translates concurrent threads into such a non-deterministic program. The state of a computation is essentially the memory plus a program counter. As the authors of [AM71] concede, this program can be exponentially larger than the original concurrent text because it has to handle all mergings of the threads ([AM71, p 37] gives some actual sizes for the examples).

The approach is non-compositional in the sense that it does not support development from specifications because it relies on having full texts (flowcharts) of all threads. Furthermore, it makes the assumption that assignment statements can be executed atomically (this point is returned to below).

A number of ways of constraining the size of the generated non-deterministic program are explained in [AM71, Part 3]:

- nested parallelism is not allowed
- blocks are used to increase "granularity"
- "protected bodies" can be marked on the flowchart by dotted lines

The examples covered are all abstract programs rather than solutions to specified problems.

The **insight** is that shared-variable concurrency (without separation) creates huge non-determinacy because of the potential interleaving of statements in threads. Furthermore, a specific way of representing an equivalent non-deterministic is given.

The next step was taken by Ashcroft alone in [Ash75]. The paper starts with the prescient observation that reasoning formally about programs will become more popular with concurrency (because concurrent programs defeat a programmer's mental ability to consider all possible mergings).

Ashcroft commented on the exponential number of proof obligations required by the approach in [AM70] and set out to tackle this issue. He proposed employing "control states" which record the statements to be executed in each concurrent thread.[11]

[9] The current author was present at the 1968 *Mathematical Theory of Computation* conference at IBM Yorktown Heights where John McCarthy strongly advocated Manna's developments of Floyd's approach.

[10] Since this paper was published in an AI journal, the examples are mainly about search algorithms but McCarthy's "91 function" is also tackled.

[11] As an aside, this bears a strong resemblance to the Control Trees that are part of the "grand state" in early Vienna operational semantic descriptions of the semantics of programming languages (see [LW69] for more on VDL and [JA16] for a discussion of the problems with these descriptions).

(3.2) **await** $\dfrac{\{P \wedge B\}\ S\ \{Q\}}{\{P\}\ \textbf{await}\ B\ \textbf{then}\ S\ \{Q\}}$

(3.3) **cobegin** $\dfrac{\{P1\}\ S1\ \{Q1\}, \ldots, \{Pn\}\ Sn\ \{Qn\} \text{ are interference-free}}{\{P1 \wedge \ldots \wedge Pn\}\ \textbf{cobegin}\ S1\ //\ \ldots\ //\ Sn\ \textbf{coend}\ \{Q1 \wedge \ldots \wedge Qn\}}$

Figure 3: Proof rules from [OG76]

Ashcroft's 1975 approach reduces the proof obligation count to the product of the control points. It is still the case that the approach is non-compositional because it is based on complete flowcharts of all threads and it retains the unrealistic assumption that assignment statements can be executed atomically.

In contrast to the 1971 joint paper, Ashcroft's 1975 paper tackles a rather ambitious "Airline Reservation System" example.

The next step is far more widely known (than the foregoing) and was made by Susan Owicki — her thesis is [Owi75] and a more accessible source is the paper co-authored with her supervisor David Gries [OG76].[12] Commonly referred to as the "Owicki-Gries" approach, a proof rule is given for an **await** statement (see rule 3.2 in Figure 3). Furthermore the approach offers a semblance of compositionality. The first task is to prove that the threads satisfy their independent pre and post conditions. It is important for the next phase that these proofs contain complete proof outlines with assertions between every statement. The rule labelled 3.3 in Figure 3 looks close to the ideal rule in Section 1.3. The snag is that the "interference free" proof obligation (elsewhere "einmischungsfrei"[13]) requires proving that every assignment in one thread does not invalidate any proof step in another thread.

The Owicki-Gries method contributes the **insight** that interference can be judged by its impact on the proof steps of other threads. it proposes a specific proof obligation as a check.

In addition to a small technical example, [OG76] introduces a *FindPos* example that employs two threads to find the least index of an array A such that some predicate p holds at $p(A(i))$; a producer/consumer problem is also addressed.

As indicated in [dR01], this is non-compositional because the correctness of each thread can only be established with respect to the finished code of all threads. It would clearly be possible that all threads were developed according to their specifications but that the final einmischungsfrei proof obligation fails and the development has to be completely restarted.

The [OG76] paper contains a specific acknowledgement to IFIP Working

[12] This paper indicates that it was intended to be "Part I" but there is no trace of subsequent parts and a recent private contact with Owicki confirmed that none was written.

[13] Gries obtained his PhD from what is now known as Technische Universität München under supervision of Bauer which is the explanation of a German adjective for the key proof obligation in the approach.

Group WG2.3: Gries first observed[14] at the meeting in December 1974 and was elected a member in September 1975; Owicki was an observer at the July 1976 meeting. These contacts possibly increased the incentive to relate the Owicki-Gries approach to [Hoa69] but it is possible to see the Owicki-Gries approach as more strongly linked to the flowcharts of [Flo67]. Owicki's thesis [Owi75] provides soundness proofs for the proof obligations and cites Peter Lauer's research with Hoare [HL74].

A more subtle objection to the Owicki-Gries approach is the assumption that single statements can be executed atomically (recall that this is also the case with the two approaches above). The problem is that this assumption does not hold for any reasonable compiler. There is an argument given that the assumption holds if there is only one shared variable per assignment.[15] This would prompt splitting any assignment $x \leftarrow x + 1$ into two assignments using a temporary variable but this still leaves the programmer needing to reason about the interference between the statements.[16]

Another interesting discussion concerns "auxiliary" (or "ghost") variables – this topic is taken up in Section 2.4.

2.2 Recovering compositionality

It is clear that finding a compositional approach to the design of concurrent programs is challenging: because threads interfere, they can potentially be affected by statements in environmental threads; the code of all threads provides maximal information but is not available at the point when a developer suggests a split into parallel threads. Although it would be judgemental to class the need for code in the methods described in Section 2.1 as a flaw, it must be conceded that it makes it impossible to achieve the sort of top-down separation that was observed in Section 1.1 for the sequential composition of statements.

One challenge therefore was to find a useful level of abstraction that faced up to interference without having the code of all threads. The key step in the Rely/Guarantee approach [Jon81, Jon83a, Jon83b] was to characterise interference by relations.[17] Figure 4:

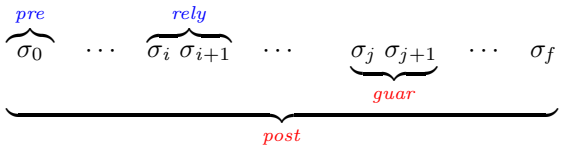
- shows pre and (relational) post conditions as in their standard VDM use
- marks that any environment interference can be thought of as an interfering step that satisfies a rely condition — it functions like a post condition of the interfering state transition

[14] IFIP working groups have a process of inviting observers (sometimes several times) before considering people for membership.

[15] This is sometimes referred to as "Reynolds' rule" but John Reynolds disowned it in a conversation with the current author.

[16] Another venue where useful exchanges on these topics occurred was Schloss Dagstuhl: there were two events on "Atomiciy" in April 2004 [BJ05, JLRW05] and spring 2006 [CJ07].

[17] VDM had consistently employed relations as post conditions — in fact, this goes back to before the name "VDM" was coined [Jon72b]. Use of data abstraction was also a key arrow in VDM's quiver [Jon72a] with [Jon80] being an early book to emphasise its use. This becomes important with Rely/Guarantee ideas — see Section 2.3.



pre/*rely* are assumptions the developer can make
guar/*post* are commitments that the code must achieve

Figure 4: A trace of states made by execution of a component and its context

- shows that the guarantee condition is also a relation and records the interference that the component being specified will inflict on the environment.

The pre, rely, guarantee and post conditions fit into the generic picture in Figure 4 and this moves the discussion on to finding an appropriate proof rule that can be used to justify steps of development that introduce concurrency. Pre, rely, guarantee and post conditions can be written as a quintuple wrapped around the program text that is to be executed: $\{P, R\}\ S\ \{G, Q\}$.[18] To indicate how the rely/guarantee rules relate to the non-interfering version of the parallel rule at the beginning Section 1.3, a slight simplification of the actual rule can be written:[19]

$$\boxed{||\text{-}R/G}\quad \frac{\begin{array}{c}\{P_1, R \vee G_2\}\ S_1\ \{G_1, Q_1\} \\ \{P_2, R \vee G_1\}\ S_2\ \{G_2, Q_2\}\end{array}}{\{P_1 \wedge P_2, R\}\ S_1\ ||\ S_2\ \{G_1 \vee G_2, Q_1 \wedge Q_2 \wedge \cdots\}}$$

One example of the use of R/G in development can be based on the "Sieve of Eratosthenes" for finding all primes up to some specified number. The specification of the interesting part of the algorithm is to remove all composite numbers from a set. Several papers [Jon96, HJ18] show how to tackle the design decision to achieve this by executing instances of $Rem(i)$ processes concurrently. The example indicates how the formulation of rely and guarantee conditions interacts with post conditions: The prime sieve example above uses symmetric (Rem) processes but this rule also caters for examples in which concurrent threads have different specifications (e.g. producer/consumer processes have asymmetric specifications).

Proof rules of the above form are proved sound with resect to a model-oriented semantics in [Jon81, Col08, Pre01]. The oldest of these proofs is fairly ugly having been based on a VDL semantics; Coleman's soundness argument is much nicer but is not machine checked; Prensa-Nieto's proof is checked on Isabelle but does not cover nested concurrency. Peter Aczel introduced a form of trace (now referred to as "Aczel traces") as a semantic model [Acz83]; they

[18] This quintuple version of rely-guarantee obviously follows Hoare triples (see Section 1.1). there are other ways of conveying the same information (see Section 5.2).

[19] The simplification is that a stronger post condition can use information from the guarantee conditions.

are employed in [dR01]. There are over twenty PhD theses around the Rely-Guarantee approach they include:

- both [Stø90, Xu92] consider progress arguments
- [Mid90] uses Temporal Logic to encode rely and guarantee conditions
- Dingel's [Din00] is an early attempt to combine refinement calculus ideas with the Rely/Guarantee approach
- [Pie09] tackles the challenging task of producing a clear formal development of the implementation by Hugo Simpson of "Asynchronous Communication Mechanisms"
- Hongjin Liang's[20] thesis [Lia14] proposes "RGSim" whose interference predicates also address ownership.

There are many examples of Rely/Guarantee developments in the literature including Owicki's *FindPos*, a concurrent cleanup addition to the Fisher-Galler implementation of "union-Find", Simpson's "four slot" algorithm, the Treiber stack and the concurrent prime sieve.

The **insight** here is that interference can be specified and reasoned about if relations are used to abstract information about interference; based on this, inference rules for parallelism can avoid the need for code of contextual threads. There are related approaches described in [dR01] under names such as "assume-commit".

The research described in this section sounds sufficiently linear that historians might fear a retrospective tidying up of the story. Two points are worth remembering: the linearity concerns only this narrow thread of research and Section 5.2 widens the viewpoint; strong interaction with other threads of research have arisen more recently and are mentioned in Section 4.

2.3 Role of data abstraction/reification

Abstract objects make it possible to write specifications which are far shorter than if the same task was specified in terms of the restricted data types of a programming language. Specifications can also postpone much algorithmic detail by employing data types that match the problem rather than efficient implementation. The process of (formally) designing a representation is referred to variously as "refinement" or "reification" (making concrete). The use of data abstraction in the specification of systems can be studied as a subject quite separate from concurrency; its origins are traced in [dRE99] and related to other aspects of program specification and development in [Jon03].

The reason for adding this subsection to the discussion of reasoning about interference is that the use of abstract data types appears to be particularly important in specifying and developing concurrent programs. It is certainly

[20]The supervisor was Xinyu Feng whose own research on "SAGL" is mentioned below.

the case that nearly all examples of Rely/Guarantee developments benefit from the use of abstract data types. To enlarge on one specific example, Simpson's "four slot" implementation of "Asynchronous Communication Mechanisms" is tackled in [JP11] where an abstract object of many "slots" is a shared variable but the rely conditions on this abstraction facilitate working out exactly what constraints need to be respected on Simpson's (four) "race free" slots.

2.4 Auxiliary variables

An issue that clouds a number of specifications and designs of concurrent programs is the use of "auxiliary" (or "ghost") variables. The idea is that variables can be inserted into a thread that do not influence its behaviour but make it easier to reason about a thread because the auxiliary variables record some information about the environment. In the extreme, such auxiliary variables could record everything about the environment including the steps to be taken by other threads. This would clearly subvert compositionality. Most uses of auxiliary variables are more constrained than this and the current author has argued that nearly all cases can be avoided if a more appropriate abstraction is found.

A delicate on-the-fly garbage collector is studied in [JY19] and the tentative conclusion is that the intimate connections between the mutator and collector threads force the use of an auxiliary variable. At the time of writing that paper, the authors were unaware of [dGR16] which might throw further light on the topic.

3 Avoiding/constraining interference

This section outlines the background of –and research in– what are termed "Concurrent Separation Logics" (CSL). There are in fact many forms of separation logic making this a large subject; here only the central points are covered and this facilitates the discussion in Section 4 on the interaction between research threads makes sense.

There are two distinguished parents of CSL research: John Reynolds' work on Separation Logic and Hoare's study [Hoa72] of how variable separation admits the use of the idealised parallel rule from the beginning of Section 1.3.[21]

A key summary of Reynolds' work on Separation Logic is [Rey02] in which he looks at the tricky topic of reasoning about programs that use dynamically allocated "heap" variables. Such programs are notoriously difficult to design and debug because mistakes can have effects that range far beyond the immediate code.

Many presentations of reasoning about such programs start with the code itself (rather than an abstract specification). Although they are in a formal

[21] Peter O'Hearn emphasised the debt to [Hoa72] during his talk in Cambridge honouring Tony Hoare in April 2009.

system, the discussions tend to be "bottom-up" in that they abstract a specification from code. (Recall that the point is made above that this offers a route to useful tool support for detecting errors in finished code.)

CSL itself also focusses on programs that employ heap variables. In a concurrent context, interaction between threads often involves an exchange of ownership of addresses between threads. To take the more obvious case of controlling which thread has ownership to write to and address, data races are avoided by making sure that only one thread has write ownership at any point in time but the logic must make it possible to reason about exchange of ownership between threads.[22]

The key reference to CSL is [O'H07] which records a talk given by Peter O'Hearn at the 2005 MFPS-XXI in honour of John Reynolds. A detailed and personal history of the evolution of CSL is available as [BO16].

Arranging that assertions cover heap addresses requires an extension of the idea of the state of a computation to include a mapping from addresses to values. Based on this it is then possible to build the notion of two assertions as having disjoint heap accesses: $P1 * P2$ can only hold if the addresses in $P1$ and $P2$ are disjoint but otherwise the asterisk functions as a logical conjunction. It is therefore easy to relate the following central CSL rule:

$$\boxed{||\text{-}CSL}\ \frac{\begin{array}{c}\{P1\}\ \ S_1\ \ \{Q1\}\\ \{P2\}\ \ S_2\ \ \{Q2\}\end{array}}{\{P1 * P2\}\ \ S_1\ ||\ S_2\ \ \{Q1 * Q2\}}$$

to the idealised rule at the beginning of Section 1.3 but it is important to remember that Hoare's rule dealt with (normal) stack variables and that the key to the above $||$ -*CSL* is handling heap variables.

Another rule that is considered important for CSL is the "frame rule" that makes it possible to apply an assertion on a limited set of variables to a larger state providing the variables are disjoint. This can be compared with the way that frames are defined for stack variables in Morgan's "refinement calculus" [Mor90] or even VDM's keyword oriented definition or read and write variables.

Having attributed the insight about separation to Hoare in Section 1.3, it could be thought that CSL "only" contributes its employment on heap variables. The current author's view is that the key **insight** is actually that CSL makes it possible to reason about ownership of heap addresses.

One issue with studying or reporting on separation logics for concurrency is their proliferation — a point made by Matthew Parkinson in the title of [Par10] ("The next 700 Separation Logics"). O'Hearn reproduces in [O'H07, Fig. 1] a chart (generated by Ilya Sergey) of developments that relates many of these logics. More recently, Parkinson and colleagues have proposed "Views" [DYBG$^+$13] as a common semantic underpinning of such logics. This at least reduces the burden of establishing the soundness of the many logics.

[22] At the MFPS-XXI conference referred to below, the current author suggested that the adjective "ownership" might describe the logic better than using "separation". The link back to Reynolds' research was too strong for this suggestion to be followed.

Research related to CSL has led to extremely successful tools that are applied in industry. Notably, O'Hearn and colleagues formed a company called "Monoidics" that was then acquired by FaceBook and reports of the impact of their tools (e.g. [DFLO19]) are extremely encouraging.

4 Productive interactions (between groups)

The title of the current paper talks about interactions between research groups and it is only for simplicity of presentation that the separation is made to appear strong between Sections 2 and 3. (Furthermore, the focus in this paper on shared-variable concurrency sidesteps discussion of many research avenues some of which are touched on in Section 5.2).

Researchers have had many interactions including:

- Both Peter O'Hearn and the current author spoke at MFPS-XXI in honour of John Reynolds — O'Hearn's [O'H07] tried to distinguish between CSL as reasoning about "race freedom" and Rely/Guarantee as tackling "racy programs";[23] Jones' contribution to the proceedings is [Jon07].
- An informal (mainly) UK *Concurrency Working Group* has met about once every nine months for over a decade
- Several of the prominent researchers involved in CSL(s) have been awarded prestigious Fellowships by the *Royal Academy of Engineering* — the current author has been the official "mentor" of most of the CSL-related awards and has found it an invaluable way of keeping in touch. In particular there were many fruitful visits to Cambridge to see Matthew Parkinson and his doctoral students.

Specific fruits of these interactions include:

- A friendly rivalry around getting clear specifications and justifications of Simpson's "four slot" implementation of "Asynchronous Communication Mechanisms" — see [JP08, BA10, JP11, BA13, JH16]
- Important attempts to bring Rely/Guarantee and CSL ideas into one framework [Vaf07, VP07, FFS07].
- Deny-Guarantee reasoning [DFPV09]
- Local Rely/Guarantee reasoning [Fen09]
- RGITL [STE+14] which combines Moskowski's [Mos85] "Interval Temporal Logic" with Rely/Guarantee ideas.

[23] The current author suspects that the negative flavour of the adjective was no accident. It does ignore the fact that there are examples where races on abstract variables are a stepping stone to designing race free representations (see [JP11]).

5 Concluding comments

The current paper has focussed on shared-variable concurrency and has identified key insights that have shaped 50 years of research in this arena. This is only one aspect of the broader subject of concurrency and this concluding section pinpoints some of the items that remain to be covered.

5.1 Recent references

This brief sub-section leaves some recent markers for researchers who might be tempted to extend this study. The work on "Views" [DYBG+13] is referenced earlier but its full impact has yet to be worked out. For example Matthew Windsor's PhD [Win19] looks at tool support directly for Views.

Section 2.3 discusses the use of data abstraction and reification in developing concurrent programs. A more recent paper [JY15] makes the point that many cases of separation can be handled by viewing separation *as* an abstraction. Two examples are presented where abstract variables that can be thought of as normal (disjoint) stack variables that can then be reified onto heap structures with the obligation that the disjointness must be established on the representation. It would be fruitful to examine many more examples.

Recent collaboration with Australian colleagues centred around Ian Hayes has led to a complete reformulation of the Rely/Guarantee approach that emphasises its algebraic properties — see [HJ18] and the references therein.

5.2 Further topics

There are many aspects of research on concurrency that have not been addressed in the body of this paper. These include:

- Process Algebras such as CSP [Hoa85], CCS [Mil89], ACP [BW90] and the π-calculus [MPW92, SW01], Hoare credits discussions with Dijkstra at the Marktoberdorf summer schools for some of the inspirations that led to CSP.
- Temporal Logics [MP91, MP95, Fis11] including TLA+ [Lam02]
- model checking — see [CHVB18]
- considerations of real time
- Petri net theory [BDK01, Old05, Rei12, Rei13]

A Leverhulme grant awarded to the current author will hopefully make it possible to cover many of these avenues.

Acknowledgements

This paper is a post-conference version of the talk given at the *History of Formal Methods* meeting in Porto in October 2019. The author is grateful to the organisers for the event and the audience for their feedback. Furthermore, Troy Astarte kindly commented on a draft of the current paper.

Past research has been funded by the EPSRC "Strata" Platform grant and earlier EPSRC Responsive mode funding on the Rely/Guarantee research. On the purely technical front, I am a Partner Investigator on Ian Hayes' ARC grant which is closely related to my concurrency research.

Our three-year Leverhulme provides funding to address more topics in the history of concurrency research.

References

[Acz83] P. H. G. Aczel. On an inference rule for parallel composition. (private communication) Manuscript, Manchester, 1983.

[AM70] E. A. Ashcroft and Z. Manna. Formalization of properties of parallel programs. Technical Report AIM–110, Stanford Artificial Intelligence Project, February 1970. Published as [AM71].

[AM71] E. A. Ashcroft and Z. Manna. Formalization of properties of parallel programs. In B. Meltzer and D. Michie, editors, *Machine Intelligence, 6*, pages 17–41. Edinburgh University Press, 1971.

[AO91] Krzysztof R. Apt and Ernst-Rüdiger Olderog. *Verification of Sequential and Concurrent Programs*. Springer-Verlag, 1991.

[AO19] K. Apt and E-R Olderog. Fifty years of Hoare's logic. *Formal Aspects of Computing*, 31(6):751–807, 2019.

[Apt81] Krzysztof R. Apt. Ten years of Hoare's logic: a survey—part I. *ACM Transactions on Programming Languages and Systems*, 3(4):431–483, October 1981.

[Apt83] Krzysztof R Apt. Ten years of Hoare's logic: A survey—part II: Nondeterminism. *Theoretical Computer Science*, 28(1-2):83–109, 1983.

[Ash75] Edward A Ashcroft. Proving assertions about parallel programs. *Journal of Computer and System Sciences*, 10(1):110–135, 1975.

[BA90] Mordechai Ben-Ari. *Principles of concurrent and distributed programming*. Prentice Hall International Series in Computer Science. Prentice Hall, 1990.

[BA10] Richard Bornat and Hasan Amjad. Inter-process buffers in separation logic with rely-guarantee. *Formal Aspects of Computing*, 22(6):735–772, 2010.

[BA13] Richard Bornat and Hasan Amjad. Explanation of two non-blocking shared-variable communication algorithms. *Formal Aspects of Computing*, 25(6):893–931, 2013.

[BDK01] Eike Best, Raymond Devillers, and Maciej Koutny. *Petri net algebra*. Springer Science & Business Media, 2001.

[BJ05] J. I. Burton and C. B. Jones. Atomicity in system design and execution. *Journal of Universal Computer Science*, 11(5):634–635, 2005.

[BO16] Stephen Brookes and Peter W O'Hearn. Concurrent separation logic. *ACM SIGLOG News*, 3(3):47–65, 2016.

[BW90] J. C. M. Baeten and W. P. Weijland. *Process Algebra*. Cambridge University Press, 1990.

[CH72] Maurice Clint and C. A. R. Hoare. Program proving: Jumps and functions. *Acta informatica*, 1(3):214–224, 1972.

[CHVB18] Edmund M Clarke, Thomas A Henzinger, Helmut Veith, and Roderick Bloem. *Handbook of model checking*, volume 10. Springer, 2018.

[CJ07] J. W. Coleman and C. B. Jones. Atomicity: A unifying concept in computer science. *Journal of Universal Computer Science*, 13(8):1042–1043, 2007.

[Col08] Joseph William Coleman. *Constructing a Tractable Reasoning Framework upon a Fine-Grained Structural Operational Semantics*. PhD thesis, Newcastle University School of Computer Science, January 2008.

[DFLO19] Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at Facebook. *CACM*, 62(8):62–70, 2019.

[DFPV09] Mike Dodds, Xinyu Feng, Matthew Parkinson, and Viktor Vafeiadis. Deny-guarantee reasoning. In Giuseppe Castagna, editor, *Programming Languages and Systems*, volume 5502 of *LNCS*, pages 363–377. Springer Berlin / Heidelberg, 2009.

[dGR16] Stijn de Gouw and Jurriaan Rot. Effectively eliminating auxiliaries. In *Theory and Practice of Formal Methods*, number 9660 in LNCS, pages 226–241. Springer, 2016.

[Din00] Jürgen Dingel. *Systematic Parallel Programming*. PhD thesis, Carnegie Mellon University, 2000. CMU-CS-99-172.

[dR01] Willem-Paul de Roever. *Concurrency Verification: Introduction to Compositional and Noncompositional Methods*. Cambridge University Press, 2001.

[dRE99] W.-P. de Roever and K. Engelhardt. *Data Refinement: Model-Oriented Proof Methods and Their Comparison*. Cambridge University Press, 1999.

[DW19] N. Dershowitz and R. Waldinger. Zohar Manna (1939–2018). *Formal Aspects of Computing*, 31(6):643–660, 2019.

[DYBG+13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: compositional reasoning for concurrent programs. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–300. ACM, 2013.

[Fen09] Xinyu Feng. Local rely-guarantee reasoning. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 315–327, New York, NY, USA, 2009. ACM.

[FFS07] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *ESOP: Programming Languages and Systems*, pages 173–188. Springer, 2007.

[Fis11] Michael Fisher. *An introduction to practical formal methods using temporal logic*. John Wiley & Sons, 2011.

[Flo67] R. W. Floyd. Assigning meanings to programs. In *Proc. Symp. in Applied Mathematics, Vol.19: Mathematical Aspects of Computer Science*, pages 19–32. American Mathematical Society, 1967.

[GvN47] Herman H. Goldstine and John von Neumann. Planning and coding of problems for an electronic computing instrument. Technical report, Institute of Advanced Studies, Princeton, 1947.

[HJ18] I. J. Hayes and C. B. Jones. A guide to rely/guarantee thinking. In Jonathan Bowen, Zhiming Liu, and Zili Zhan, editors, *Engineering Trustworthy Software Systems – Second International School, SETSS 2017*, LNCS. Springer-Verlag, 2018.

[HL74] C. A. R. Hoare and P. E. Lauer. Consistent and complementary formal theories of the semantics of programming languages. *Acta Informatica*, 3(2):135–153, 1974.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[Hoa71a] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engeler, editor, *Symposium On Semantics of Algorithmic Languages*, volume 188 of *LNM*, pages 102–116. Springer-Verlag, 1971.

[Hoa71b] C. A. R. Hoare. Proof of a program: FIND. *Communications of the ACM*, 14(1):39–45, January 1971.

[Hoa72] Charles Antony Richard Hoare. Towards a theory of parallel programming. In *The origin of concurrent programming*, pages 231–244. Springer, 1972.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[JA16] Cliff B. Jones and Troy K. Astarte. An Exegesis of Four Formal Descriptions of ALGOL 60. Technical Report CS-TR-1498, Newcastle University School of Computer Science, September 2016.

[JH16] Cliff B. Jones and Ian J. Hayes. Possible values: Exploring a concept for concurrency. *Journal of Logical and Algebraic Methods in Programming*, 2016.

[JLRW05] C. B. Jones, D. Lomet, A. Romanovsky, and G. Weikum. The atomic manifesto. *Journal of Universal Computer Science*, 11(5):636–650, 2005.

[Jon72a] C. B. Jones. Formal development of correct algorithms: an example based on Earley's recogniser. In *SIGPLAN Notices*, volume 7:1, pages 150–169. ACM, 1972.

[Jon72b] C. B. Jones. Operations and formal development. Technical Report TN 9004, IBM Laboratory, Hursley, September 1972.

[Jon80] C. B. Jones. *Software Development: A Rigorous Approach*. Prentice Hall International, Englewood Cliffs, N.J., USA, 1980.

[Jon81] C. B. Jones. *Development Methods for Computer Programs including a Notion of Interference*. PhD thesis, Oxford University, June 1981. Printed as: Programming Research Group, Technical Monograph 25.

[Jon83a] C. B. Jones. Specification and design of (parallel) programs. In *Proceedings of IFIP'83*, pages 321–332. North-Holland, 1983.

[Jon83b] C. B. Jones. Tentative steps toward a development method for interfering programs. *Transactions on Programming Languages and System*, 5(4):596–619, 1983.

[Jon96] C. B. Jones. Accommodating interference in the formal design of concurrent object-based programs. *Formal Methods in System Design*, 8(2):105–122, March 1996.

[Jon03] Cliff B. Jones. The early search for tractable ways of reasoning about programs. *IEEE, Annals of the History of Computing*, 25(2):26–49, 2003.

[Jon07] C. B. Jones. Splitting atoms safely. *Theoretical Computer Science*, 375(1–3):109–119, 2007.

[JP08] Cliff B. Jones and Ken G. Pierce. Splitting atoms with rely/guarantee conditions coupled with data reification. In *ABZ2008*, number 5238 in LNCS, pages 360–377. Springer, 2008.

[JP11] Cliff B. Jones and Ken G. Pierce. Elucidating concurrent algorithms via layers of abstraction and reification. *Formal Aspects of Computing*, 23(3):289–306, 2011.

[JY15] Cliff B. Jones and Nisansala Yatapanage. Reasoning about separation using abstraction and reification. In Radu Calinescu and Bernhard Rumpe, editors, *Software Engineering and Formal Methods*, volume 9276 of *LNCS*, pages 3–19. Springer, 2015.

[JY19] Cliff B. Jones and Nisansala Yatapanage. Investigating the limits of rely/guarantee relations based on a concurrent garbage collector example. *Formal Aspects of Computing*, 31(3):353–374, 2019. online April 2018.

[Lam02] Leslie Lamport. *Specifying systems: the TLA+ language and tools for hardware and software engineers*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[Lia14] Hongjin Liang. *Refinement Verification of Concurrent Programs and Its Applications*. PhD thesis, USTC, China, 2014.

[LW69] Peter Lucas and Kurt Walk. On the formal description of PL/I. *Annual Review in Automatic Programming*, 6:105–182, 1969.

[Man70] Zohar Manna. The correctness of nondeterministic programs. *Artificial Intelligence*, 1(1-2):1–26, 1970.

[Mid90] C. A. Middelburg. *Syntax and Semantics of VVSL: A Language for Structured VDM Specifications*. PhD thesis, PTT Research, Leidschendam, Department of Applied Computer Science, September 1990.

[Mil89] Robin Milner. *Communication and Concurrency*. Prentice Hall, 1989.

[MK99] Jeff Magee and Jeff Kramer. *State models and Java programs*. Wiley, 1999.

[Mor90] Carroll Morgan. *Programming from Specifications*. Prentice-Hall, 1990.

[Mos85] Ben Moszkowski. Executing temporal logic programs. In Stephen D. Brookes, Andrew William Roscoe, and Glynn Winskel, editors, *Seminar on Concurrency*, volume 197 of *LNCS*, pages 111–130. Springer Berlin Heidelberg, 1985.

[MP91] Z. Manna and A. Pnueli. *Temporal Logic of Reactive Systems*. Springer-Verlag, New York, 1991.

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems*. Springer-Verlag, 1995.

[MPW92] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes. *Information and Computation*, 100:1–77, 1992.

[Nau66] Peter Naur. Proof of algorithms by general snapshots. *BIT Numerical Mathematics*, 6(4):310–316, 1966.

[OG76] S. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs I. *Acta Informatica*, 6:319–340, 1976.

[O'H07] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1-3):271–307, May 2007.

[Old05] E-R Olderog. *Nets, terms and formulas: three views of concurrent processes and their relationship*, volume 23. Cambridge University Press, 2005.

[Owi75] S. S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Department of Computer Science, Cornell University, 1975. Published as technical report 75-251.

[Par10] Matthew Parkinson. The next 700 separation logics. In Gary Leavens, Peter O'Hearn, and Sriram Rajamani, editors, *Verified Software: Theories, Tools, Experiments*, volume 6217 of *LNCS*, pages 169–182. Springer Berlin / Heidelberg, 2010.

[Pie09] Ken Pierce. *Enhancing the Useability of Rely-Guaranteee Conditions for Atomicity Refinement*. PhD thesis, Newcastle University, 2009.

[Pre01] Leonor Prensa Nieto. *Verification of Parallel Programs with the Owicki-Gries and Rely-Guarantee Methods in Isabelle/HOL*. PhD thesis, Institut für Informatic der Technischen Universität München, 2001.

[Rei12] Wolfgang Reisig. *Petri nets: an introduction*, volume 4. Springer Science & Business Media, 2012.

[Rei13] Wolfgang Reisig. *Understanding Petri nets: modeling techniques, analysis methods, case studies.* Springer, 2013.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of 17th LICS*, pages 55–74. IEEE, 2002.

[Sch97] Fred B. Schneider. *On concurrent programming.* Springer-Verlag, 1997.

[Ste66] T. B. Steel. *Formal Language Description Languages for Computer Programming.* North-Holland, 1966.

[STE+14] Gerhard Schellhorn, Bogdan Tofan, Gidon Ernst, Jörg Pfähler, and Wolfgang Reif. RGITL: A temporal logic framework for compositional reasoning about interleaved programs. *Annals of Mathematics and Artificial Intelligence*, 71(1-3):131–174, 2014.

[Stø90] K. Stølen. *Development of Parallel Programs on Shared Data-Structures.* PhD thesis, Manchester University, 1990. Published as technical report UMCS-91-1-1.

[SW01] Davide Sangiorgi and David Walker. *The π-calculus: A Theory of Mobile Processes.* Cambridge University Press, 2001.

[Tur49] A. M. Turing. Checking a large routine. In *Report of a Conference on High Speed Automatic Calculating Machines*, pages 67–69. University Mathematical Laboratory, Cambridge, June 1949.

[Vaf07] Viktor Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, 2007.

[VP07] Viktor Vafeiadis and Matthew Parkinson. A marriage of rely/guarantee and separation logic. In Luís Caires and Vasco Vasconcelos, editors, *CONCUR 2007 – Concurrency Theory*, volume 4703 of *LNCS*, pages 256–271. Springer, 2007.

[vW66] Aadrian van Wijngaarden. Numerical analysis as an independent science. *BIT Numerical Mathematics*, 6(1):66–81, 1966.

[Win19] Matthew Windsor. *Starling: A Framework for Automated Concurrency Verification.* PhD thesis, University of York, 2019.

[Xu92] Qiwen Xu. *A Theory of State-based Parallel Programming.* PhD thesis, Oxford University, 1992.