Concurrent Separation Logic

Stephen Brookes
Carnegie Mellon University

Peter W. O'Hearn
University College London

Concurrent Separation Logic (CSL) was originally advanced in papers of the authors published in Theoretical Computer Science for John Reynolds's 70th Birthday Festschrift (2007). Preliminary versions appeared as invited papers in the CONCUR'04 conference proceedings. Foundational work leading to these papers began in 2002. Since then there have been significant developments stemming from CSL, both in theoretical and practical research. In this retrospective paper we describe the main ideas that underpin CSL, placing these ideas into historical context by summarizing the prevailing tendencies in concurrency verification and programming language semantics when the logic was being invented in 2002-2003. We end with a snapshot of the state-of-the-art as of 2016. Along the way we describe some of the main developments in the intervening period, and we attempt to classify the work that has been done, along broad lines. While we do not intend an exhaustive survey, we do hope to provide some general perspective on what has been achieved in the field, what remains to be done, and directions for future work.

1. CONTEXT, *CIRCA* 2002-2003

1.1. Verification of Concurrent Programs

> For the last thirty years experts have regarded pointer manipulation as an unsolved challenge for program verification and shared-memory concurrency as an even greater challenge. *2016 Godel Prize citation*[1]

As of 2002-2003, there had already been significant foundational work on the verification of concurrent programs, beginning with the classical work of Owicki and Gries [Owicki and Gries 1976a], who adapted ideas from Hoare's logic for sequential programs to a concurrent setting. Further important developments were made by Pnueli [Pnueli 1981], bringing temporal logic to concurrency, and by Jones [Jones 1983], who advanced the rely-guarantee proof method for compositional reasoning. There was also a large school (more accurately, separate schools) of work that had built up around process algebras such as CCS, CSP and the pi-calculus.

However, it is fair to say that at that time few realistic concurrent programs had been subjected to proof, and this was especially true for mechanized proof. Yet, around 2002-2003, the field of mechanized program verification was going through a renaissance, spurred by verification-oriented abstract interpreters such as SLAM and ASTREE, and advances in proof assistants such as Coq, Isabelle and HOL. The hitch was

[1]http://eatcs.org/index.php/component/content/article/1-news/2280-2016-godel-prize

that most of this work was for sequential programs. Model checking had been used to verify certain finite-state concurrent programs used to describe hardware, with notable success, but much less had been achieved by then for concurrent software.

Around the same time, the separation logic for sequential programs had recently been advanced by O'Hearn, Reynolds and others as a way to provide efficient reasoning about pointer manipulation [Reynolds 2000; Ishtiaq and O'Hearn 2001; O'Hearn et al. 2001; Reynolds 2002]. Separation logic was a significant advance over prior formalisms for reasoning about pointers (see [Bornat 2000] and its references for the state of the art as of 1999), providing inference rules to localize reasoning about heap chunks. Pointer mutation was also one of the key blockers in applying the foundational approaches provided by the Owicki-Gries, temporal logic and rely-guarantee methodology to real-world concurrent programs. Programs in C, Java and many current languages often make non-trivial use of pointer manipulation, to great effect, but reasoning about the correctness of such code is difficult even when the code is sequential, because of issues such as aliasing. Pressing these classic works into service for proving correctness of code using pointers requires treating the memory as a global array, and this leads to global proofs which are complicated and rather remote from programming intuition. Often, programmers think of portions of memory a little bit at a time, and employ localized reasoning that does not mention the whole memory; to force programmers to think in terms of global state flies in the face of such intuition. It therefore made sense to attempt to update one of these classical approaches to reasoning about concurrent programs by putting it together with separation logic, in which localized reasoning is very natural.

O'Hearn considered all three of these prior foundational works as possibilities for such an update, but gravitated to an even earlier work, Hoare's 1972 paper "Towards a Theory of Parallel Programming" (TTPP, [Hoare 1972]). Hoare's paper might be considered to have been superceded by its successor works, because there were many programs which could not be expressed in TTPP because of syntactic restrictions used to rule out race conditions and other forms of interference. But when it worked it allowed for delightfully simple proofs.

A useful perspective is brought by noting, and contrasting, the existence of not one but two Owicki-Gries logics. One [Owicki and Gries 1976b], which we will refer to as *race-free Owicki-Gries*, extends TTPP by slightly relaxing syntactic restrictions in a way that allows auxiliary variables to be used more freely in proofs, but still maintaining control over interference. The other [Owicki and Gries 1976a], which we will refer to as *interference-allowing Owicki-Gries*, allows interference between threads but uses a novel means of checking that a thread does not inferfere with a *proof* of another thread (note the concept of non-interference with proof, not with thread). The interference-allowing form of Owicki-Gries is often referred to (without qualification) as *the* "Owicki-Gries method", and it attracted more attention than race-free Owicki-Gries in work between the mid 1970s and 2002-2003, likely because it was applicable to more programs. But Owicki and Gries also state that "the proof process becomes much longer" in the less restrictive, interference-allowing Owicki-Gries system.

The design of Hoare's TTPP approach (and also race-free Owicki-Gries) was based almost completely, albeit implicitly, on intuitions of separation, so it just made sense to try to recast these intuitions explicitly by making use of the more recent formalism of separation logic.

1.2. Semantics of Concurrent Programs

At the same time that O'Hearn was considering ways to blend separation logic with Hoare-style concurrency logics, the state-of-the-art in programming language semantics of concurrency stood on foundations laid by David Park in the 1970s. Traditional approaches, whether denotational or operational, relied on traces of some kind. In Park's seminal work [Park 1979] a trace is a sequence of steps, each step representing the effect of an atomic action performed on the global state by the program, assumed to be running in an environment of concurrent processes. To achieve a compositional account the trace set of a program includes traces with "gaps" allowing for the environment to make changes to the global state. Concurrent processes are assumed to be executed in a sequentially consistent manner, with the atomic actions of all processes being interleaved. In later work Brookes used traces in which the steps represent the effect of a finite sequence of atomic actions, a simple variation that led to a fully abstract semantics [Brookes 1996].

These semantic models were only applicable to shared-memory programs without data structures that could be mutated through pointer manipulation, a class of programs dubbed "simple" shared-memory by John Reynolds. For many years it had been widely regarded as a difficult task just to find a tractable denotational model for simple shared-memory, let alone develop a semantics that could deal with the combination of shared-memory concurrency and mutable data structures. These traditional semantic models were all based on global states and typically assumed that assignment to a variable was an atomic action, ignoring the potential for race conditions, even though it was known that when one process writes to a shared variable being read or written by another there is a danger of unpredictability. Instead one of the rôles of logics such as race-free Owicki-Gries [Owicki and Gries 1976b] and TTPP was to shift the burden of race-detection from semantics to logic: an assertion $\{P\}C\{Q\}$ approvable in race-free Owicki-Gries logic comes with the guarantee that when C is executed from a global state satisfying P there are no race conditions and upon termination the global state satisfies Q. This logic-based methodology only works for races involving program variables, which are statically detectable: they get dealt with in Owicki-Gries by imposing static constraints on variable usage in programs. This idea fails for programs using pointers, because aliasing of pointers is not statically determinable.

In summary, in 2002-2003 there was not yet any suitable semantic foundation for exploring the behaviour of concurrent programs that make non-trivial use of pointers. And the currently existing program logics that worked well enough for pointer-free concurrent programs relied on syntactic constraints to rule out race conditions, which prevents their use in richer settings where the existence of a potential race cannot merely be deduced from program syntax.

2. THE BIRTH OF CONCURRENT SEPARATION LOGIC

2.1. Logic

The most important proof rule of Concurrent Separation Logic is the

PARALLEL COMPOSITION RULE

$$\frac{\{P_1\}\,C_1\,\{Q_1\}\ \cdots\ \{P_n\}\,C_n\,\{Q_n\}}{\{P_1 * \cdots * P_n\}\,C_1 \parallel \cdots \parallel C_n\,\{Q_1 * \cdots * Q_n\}}$$

Here, the separating conjunction $P_1 * \cdots * P_n$ in the precondition of the parallel composition is true of a state that can be partitioned into substates making the conjuncts true, and similarly for the post-condition. So the rule says: To prove a parallel composition we give each process a separate piece of state, and separately combine the postconditions for each process.[2] The rule supports completely independent reasoning about

[2]In the earliest versions of CSL these rules were accompanied by side conditions governing variables (but not the heap), such as: no variable free in P_i or Q_i is changed in C_j when $j \neq i$. This was an historical pre-

processes. And, it is unsound if $*$ is replace by $\wedge$ because mutations in one process would invalidate the reasoning about common state in another process.

In this expository paper we will show "proof outlines" rather than logical proofs, and without even formally defining the logic in full; we hope that the spirit of the proof methodology can be gleaned from the assertions placed at program points.

An example use of the parallel composition proof rule is given by a proof of parallel mergesort. In the key step

$$\begin{array}{ccc}
\multicolumn{3}{c}{\{\mathit{array}(a,i,m) * \mathit{array}(a,m+1,j)\}} \\
\{\mathit{array}(a,i,m)\} & & \{\mathit{array}(a,m+1,j)\} \\
\mathtt{ms}(a,i,m) & \| & \mathtt{ms}(a,m+1,j) \\
\{\mathit{sorted}(a,i,m)\} & & \{\mathit{sorted}(a,m+1,j)\} \\
\multicolumn{3}{c}{\{\mathit{sorted}(a,i,m)\ *\ \mathit{sorted}(a,m+1,j)\}}
\end{array}$$

we first reason independently about two independent recursive calls which operate on disjoint subarrays, and then reason about merging the sorted subarrays.

This method of reasoning about parallel mergesort is simple, straightforward, almost trivial. And that is the point. In the previous formalisms (interference-allowing Owicki-Gries, Rely-Guarantee, Temporal logic) parallel mergesort would require a complicated proof which had to explicitly describe, and then rule out, the possibility of interference. On the other hand, in TTPP and race-free Owicki-Gries parallel mergesort is not even syntactically well-formed, because assignments to components of an array in separate threads are viewed logically as assignments to the same variable; although these systems are intuitively about separation, their syntactic constraints are coarser than separation-expressed-with-logic. The CSL proof, in contrast, follows the informal reasoning directly. Although this simple reasoning pattern naturally has its limitations, it illustrates a principle which should be central in the subject of logics of programs: that is, *simple proofs for simple programs*. It is all too easy to get caught up in completeness and related issues for formal systems that turn out to be too complicated when humans try to apply them; it is more important first to get a sense for the extent to which simple reasoning is or is not supported.

Still, if CSL had only been able to reason about "disjoint concurrency", where there is no inter-process interaction, then it would have rightly been considered rather restrictive. A proof rule for conditional critical regions (a forerunner of monitors) allows reasoning about such interaction.

CRITICAL REGION RULE

$$\frac{\{(P * RI_r) \wedge B\}\, C\, \{Q * RI_r\}}{\{P\}\ \mathtt{with}\ r\ \mathtt{when}\ B\ \mathtt{do}\ C\ \{Q\}}$$

This rule assumes given an association of a "resource invariant" RI_r, to each "resource" r appearing in the program. A resource is like a monitor lock: it provides mutual exclusion for different occurrences of critical regions for r in a program. The rule supports modular reasoning by using the separating conjunction to control the visibility of the state described by the invariant, as well as the invariant itself.

An important early example done with CSL was the pointer-transferring buffer. In this example one thread allocates a pointer and puts it into a buffer, while the other thread reads it out and disposes (frees) it. The important thing about this code is that

sentational choice to do with keeping consistent with earlier Hoare logics; more recently, logics are given in which $*$ handles all noninterference ("variables as resource", [Parkinson et al. 2006]) or where local variable mutation is restricted to be thread-local.

not only is the pointer deemed to transfer from one process to another, but the "knowledge that it is allocated" or the "right to dereference it", or more simply "ownership" of the pointer, transfers with the proof. From the point of view of the buffer clients the proof of ownership transfer looks like this

```
            {emp ∗ emp}
{emp}                      {emp}
x := cons(a, b);   ||      get(y);
{x ↦ –, –}                 {y ↦ –, –}
put(x);                    use(y);
{emp}                      {y ↦ –, –}
                           dispose(y);
                           {emp}
            {emp ∗ emp}
            {emp}
```

where the buffer code itself is

```
put(x) ≜  with buf when ¬full do
                 c := x; full := true;

get(y) ≜  with buf when full do
                 y := c;  full := false;
```

To reason about the buffer code using the critical region rule we supply a resource invariant

$$RI_{buf}: \quad (full \wedge c \mapsto -,-) \vee (\neg full \wedge \texttt{emp})$$

saying that the buffer owns the binary cons cell associated with c when *full* is true, and otherwise it owns no heap cells. The entire program begins with an initialization *full* := `false` that establishes the invariant.

Then we can provide a proof outline for the body of the `with` command in `put`(x).

```
{(RI_buf ∗ x ↦ –, –) ∧ ¬full}
{(¬full ∧ emp) ∗ x ↦ –, –}
{x ↦ –, –}
 c := x; full := true
{full ∧ c ↦ –, –}
{RI_buf}
{RI_buf ∗ emp}
```

The rule for `with` commands then gives us

$$\{x \mapsto -,-\}\texttt{put}(x)\{\texttt{emp}\}.$$

The postcondition indicates that the sending process gives up ownership of pointer x when it is placed into the buffer, even though the value of x is still held by the sender. A crucial point in the proof of the body is the penultimate step which passes from

$$full \wedge c \mapsto -,-$$

to RI_{buf}, reflecting the idea that the knowledge "x points to something" flows out of the user program and into the buffer resource. Some inverse manipulations give us the spec

$$\{\texttt{emp}\}\texttt{get}(y)\{y \mapsto -,-\}.$$

for getting an element out of the buffer, where the knowledge that y points to something materializes in the postcondition.

The ownership transfer idea illustrated by this buffer example made it clear that quite a few concurrent programs would have much simpler and more intuitive proofs than before, and that a wider class of programs would be susceptible to formal analysis. Modular proofs were provided of semaphore programs, of a toy memory manager, and programs with interacting resources. Generally speaking, ownership transfer allows modular reasoning about 'daring concurrency', where where a piece of state is accessed even outside of groupings of mutual exclusion. It seemed as if the logic could explain the way that synchronisation had been used in the fundamental works on concurrent programming by Dijkstra, Hoare and Brinch Hansen. For example, in the paper that essentially founded concurrent programming, Dijkstra (Co-operating Sequential Processes, [Dijkstra 1968]) had explained that the point of synchronisation was to enable programmers to avoid minute considerations of timing, in order to simplify reasoning. Brinch Hansen had emphasized the importance of speed independence and resource separation for simplifying thinking about concurrent processes when O'Hearn was his colleague at Syracuse in the 1990s, and what he said seemed to be mirrored in the proofs in this early concurrent separation logic.

In summary, the most important contributions of the logic are as follows.

— it provides a modular way of reasoning about concurrent programs by separating the state that different threads access,
— while achieving this in the presence of pointer access and mutation,
— and it supports modular reasoning about process interaction, even in the presence of 'daring concurrency' where common state is safely accessed by different processes outside of critical regions or other mutual exclusion constructs (as in a memory manager or the pointer-transferring buffer).

But, soundness was challenging.

2.2. Semantics

It was the very feature that gave rise to the unexpected power in the logic, the ownership or knowledge transfer, that made soundness non-obvious. O'Hearn was able to state principles intended to give some justification for the logic, such as the following.

> *Ownership Hypothesis*. A code fragment can access only those portions of state that it owns.
>
> *Separation Property*. At any time, the state can be partitioned into that owned by each process and each grouping of mutual exclusion.

But these principles were stated only informally. O'Hearn worked hard on the soundness problem for several months in the second half of 2001 and early in 2002, and got nowhere near a formal soundness theorem. During 2002 he asked Brookes, an expert in the semantics of concurrency, for help.

To establish soundness of CSL we needed a semantic model capable of dealing both with concurrency and pointer maipulation, suitable for formalizing and supporting accurate reasoning about ownership and separation. Although these notions seem intuitive, it was not at all obvious at the time how to design a suitably comprehensive denotational semantics, and how to make rigorous the idea behind ownership transfer and the notion that (provable) processes mind their own business. A key design choice was to take a more localized view of states, so that a global state could be seen as a combination of separate pieces of state deemed to belong to processes and resources.

While it may appear fairly straightforward to extend Park-style global traces to handle pointers and mutable state, this leads again to a semantics based on global states and it was unclear how to build in a more localized view of state. Instead we had the idea of abstracting away from state and using traces built from "actions" , initially uninterpreted and only characterized abstractly, which can later be interpreted as having an effect on state. In this two-stage approach, we would be free to instantiate the notion of state in whatever way we needed, either as global or as local. We would be able to analyze program behavior, independently of the logic, using global states; and to formalize ownership transfer and separation properties by taking a local view of traces. Another advantage, which emerged later, is that the *same* action trace semantics would turn out to be usable in validating a number of later program logics inspired by CSL, such as *logics of permissions*, merely by interpreting actions over an alternative notion of state, such as *permissive states*.

To deal with race-detection we introduced a form of interleaving operator for traces that detects race conditions and treats them as catastrophic. This is the right choice to obtain a semantics capable of establishing when a program is race-free. The new semantic model, which we refer to as *action traces*, is strongly influenced by Dijkstra's Principle [Dijkstra 1968]:

> ... processes should be loosely connected; by this we mean that apart from the (rare) moments of explicit intercommunication, the individual processes are to be regarded as completely independent of each other.

In other words, concurrent processes do not interfere except through explicit synchronization. Action trace semantics reflects this idea through the interplay between traces, which describe interleaved behaviors of processes, and a "local enabling" relation that implements the "no interference from outside" notion. This interplay is crucial in permitting a formalization of O'Hearn's intuitive concept of "processes that mind their own business". To our knowledge, this was the first semantics in which such a formalization is possible.

The key novel features of this semantics:

— a compositional action-trace semantics built from "uninterpreted" actions, capable of incorporating both concurrency and pointers;
— a race-detecting interpretation of parallel composition;
— a global-state interpretation of actions and traces, consistent with a standard operational notion of execution;
— a local-state interpretation of actions and traces ("local enabling"), suitable for formalizing ownership transfer and the separation principle

Local enabling formalizes the notion of a process executing in an environment that respects resources (obeys the separation principle) and "minds its own business" by following the ownership transfer discipline embodied by the resource invariants. A key ingredient in the soundness proof is a Parallel Decomposition Lemma; in simplified form, this says that when $c_1 \| c_2$ is a race-free program, every interleaved computation of $c_1 \| c_2$ can be decomposed into "local" computations of the constituent processes c_1 and c_2 which are interference-free except for interactions with protected resources.

Our soundness proof assumes that each resource invariant is *precise*, so that every time a program acquires or releases a resource there is a uniquely determined portion of the heap whose ownership can be deemed to transfer. This does not seem to be a major limitation, since all of O'Hearn's examples involve precise invariants, and a methodology based on precision seems very natural [O'Hearn 2007]. Moreover this limitation is sufficient to ensure soundness, and it suffices to avoid a counterexample discovered by John Reynolds showing unsoundness when resource invariants are allowed to be arbitrary separation logic formulas and the usual Hoare logic proof rule for conjoining postconditions is allowed. During the evolution of the semantic foundations John Reynolds played an important guiding rôle, and it is entirely appropriate that the two CSL papers bear dedication to him.

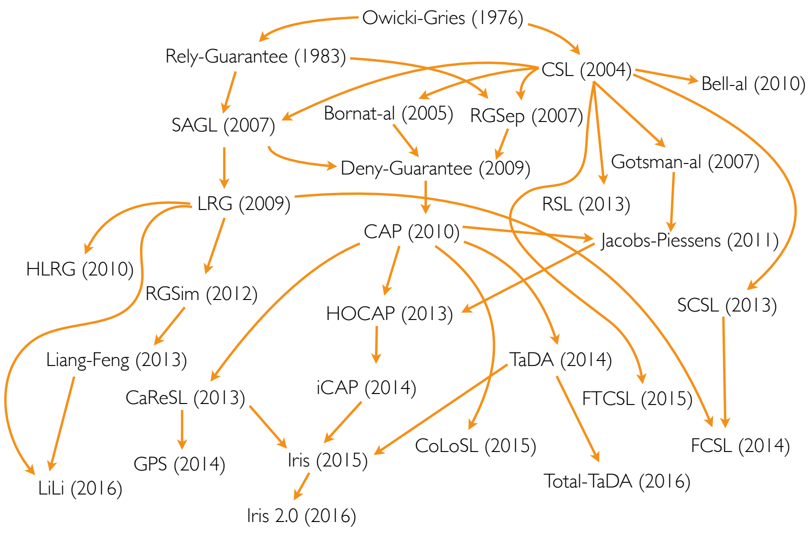

Fig. 1. CSL Family Tree (courtesy of Ilya Sergey)

3. DEVELOPMENTS IN THEORY

3.1. Logic

There have been many logics that extend or build on the ideas in CSL; Figure 1 gives an indication. The motivation for several of these logics has simply been to treat varied programming primitives that are used when writing concurrent programs. For example:

— *Storable locks.* The original CSL used statically allocated locks, whereas real programs often use dynamically-allocated locks that can themselves be stored [Gotsman et al. 2011];
— *Re-entrant locks.* CSL's critical sections cannot be nested, and similarly if you hold a semaphore and attempt to grab it again you will deadlock. On the other hand, Java's locks are such that a thread that holds a lock can acquire it again [Haack et al. 2008];
— *Fork/join.* CSL is formulated using structured parbegin/parend or parallel composition $\|$ to describe concurrent threads, which has been extended to fork/join concurrency constructs [Haack and Hurlin 2008; Dodds et al. 2009];
— *Message Passing.* [Villard et al. 2009; Bell et al. 2010; Lei et al. 2014];
— *Relaxed memory.* [Vafeiadis and Narayan 2013].

A very unexpected development has been the demonstration that the most basic principles of concurrent separation logic, particularly independent reasoning about threads using the separating conjunction, cover a much broader range of situations than we ever expected. There have been proofs of fine-grained locking and non-blocking concurrency, and cases that involve interference and general graph structures, what might have been thought of originally as cases that don't fit well with the ideas of separation logic.

Interestingly, the unexpected power of this is based on what you might call "non-standard models" of separation logic; we mean this by analogy with the usual situation in logic, where a theory (e.g. reals, or integers) has an intended model, but also additional non-standard models of the same axioms. The proof theory may then accomplish unexpected things when applied to the non-standard models. The standard model of separation logic is the original model based on splitting portions of the heap, or heaplets. There are many other models stemming from the "resource semantics" of bunched logic invented by David Pym [Pym et al. 2004]; the preconditions and post-conditions in separation logic and an instance of bunched logic [Ishtiaq and O'Hearn 2001]. Indeed, "abstract separation logic" [Calcagno et al. 2007a] defines a version of CSL starting from an arbitrary partial commutative monoid of program states, in place of the standard partial monoid of heaplets and disjoint union. A partial commutative monoid $(M, \bullet, e)$ induces an ordered total commutative monoid on the powerset $\mathcal{P}(M)$, and this is used as the semantics of $P * Q$. In fact, even more generally we could go beyond powerset models and define an abstract separation logic where a model is an ordered total commutative monoid; this gives enough structure to formulate the proof rule for parallel composition and the rule of consequence. More structure can be added (such as meets and joins) to interpret other logical connectives, depending on what is needed in an application.

The surprise is that some of these nonstandard models involve composing highly intertwined structures and interfering processes, what might have been considered bad cases for the ideas behind separation logic. Gardner coined the phrase "fiction of separation" to describe this phenomenon in the nonstandard models. Here are some representative works.

—*Permissions.* These are models where heap locations have additional information attached, which sometimes affects heap composition. An insightful paper of Boyland [Boyland 2003] influenced CSL work where fractional and counting permission models were given to account for concurrent reads and for the classic readers-and-writers problem[Bornat et al. 2005], and since then a wide variety of permission models have been invented and applied.
—*Abstraction and Fictions of Separation.* The logic of Concurrent Abstract Predicates (CAP) of [Dinsdale-Young et al. 2010a] provides a powerful means of disjoint reasoning about processes accessing an abstract module, when the concrete implementations are in fact not disjoint. This aspect, which builds on observations about the "fiction of disjointness" from work in sequential separation logic [Dinsdale-Young et al. 2010b], significantly expands the possibilities for applying CSL-style reasoning to fine-grained concurrent algorithms. CAP provides means of connecting an abstract module to a concrete implementation in a way that allows more apparent separation on the abstract level. Extensions of CAP have been used to do impressive verifications of libraries of synchronization primitives (e.g., [Svendsen et al. 2013; Dodds et al. 2016]). CAP has influenced many follow-on logics, as can be seen in Figure 1.
—*Views.* Dinsdale-Young, Parkinson and colleagues show that a simple abstract version of concurrent separation logic can embed many other techniques for reasoning about concurrency including type systems and even the classic rely-guarantee method, which was invented for the purpose of reasoning about interference [Dinsdale-Young et al. 2013]. Earlier work had sought combinations of separation logic and rely-guarantee [Feng et al. 2007; Vafeiadis and Parkinson 2007], but this work and [Dodds

et al. 2009] demonstrated that by picking a suitable monoid that expresses interference rely-guarantee could be expressed with a separation logic, constituting a valuable synthesis. Views also provides logical or fictional separation. It is a framework that can be instantiated to many logics, and so does this in a considerably more general way than CAP, while partly taking inspiration from it.
The logic of Views is similar to that of Abstract Separation Logic, but the semantics is different, and this is what opened up the possibility of a broader variety of instantiations of the framework: we say more on this in the next section.
— *Modular reasoning about history.* It is very natural to use temporal reasoning when reasoning about concurrent algorithms. For example, there are cases when wants to talk about the value of a piece of state sometime in the past, or between two time-points. [Fu et al. 2010] and [Sergey et al. 2015] define logics that support modular ways for reasoning about both space and time, based on ways to compose histories, and they illustrate the power of this form of reasoning by proving optimistic and fine-grained locking concurrent algorithms. The previously-mentioned logics for message passing [Villard et al. 2009; Bell et al. 2010; Lei et al. 2014] also include support for reasoning about history.

Before moving on it will be instructive to consider an example of a non-standard model from the Views work [Dinsdale-Young et al. 2013]. Assume a partial commutative monoid $(M, \bullet, e)$ (e.g, the monoid where $M = L \rightharpoonup_f V$ is heaps and $\bullet$ is union of disjoint heaps). An *interference relation* is a subset $R \subseteq M \times M$ satisfying certain conditions. Then we can define a total commutative monoid $(Stab, *, \{e\})$ where the set $Stab = \{p \subseteq M \mid Rp \subseteq p\}$ consists of those sets (predicates) that are stable under the interference relation. $*$ is a lifting of $\bullet$, where $p * q = \{h_p \bullet h_q \mid h_p \in P \text{ and } h_q \in q \text{ and } h_p \bullet h_q \downarrow\}$. The (unstated, here) conditions on R ensure that $\{e\}$ and $p * q$ are stable.

The notion of stability under interference is one of the hallmarks of rely-guarantee reasoning. Models based on monoids like $(Stab, *, \{e\})$ provide one of the ways that separation logics can be used to reason modularly about interfering processes.

Even with the tremendous progress that has occurred, techniques for modular reasoning in the presence of fine-grained interference continue to evolve, with several substantial works appearing just in the past year or so; e.g. [Raad et al. 2015; Jung et al. 2015; da Rocha Pinto et al. 2016; Liang and Feng 2016].

3.2. Semantics

The action traces semantics of Brookes was denotational by choice: a denotational semantic description lends itself naturally to compositional reasoning, and this can be an advantage because most of the inference rules of CSL are syntax-directed, like the semantic clauses. (The exceptions are rules such as the Frame Rule, and the Conjunction Rule, in which the premiss and conclusion involve the same program and such rules typically express some general semantic or behavioral property applicable to all programs.)

Nevertheless, much of the notation used by Brookes in the soundness paper was deliberately chosen to emphasize intuitive similarities with operational style. For example, the notation for local enabling and global enabling resembles the labelled transition relations common in operational semantics. These remarks reflect the view that denotational and operational semantics should be seen as complementary styles of language description, not mutually exclusive.

As we have seen, CSL led to a proliferation of successor logics. Some of these can claim common philosophical links with the original logic, in that they are based on analogous to the Separation Principle and Ownership Transfer, while departing in ways from the original, while others use proofs that do not depend on these principles.

— [Brookes 2006]: Brookes extends his semantics to cover permission-based variants of CSL, such as fractional and counting permissions [Bornat et al. 2005] and variables as resource [Parkinson et al. 2006].
— [Calcagno et al. 2007a]: the soundness result for Abstract Separation Logic is given by a generalization of the approach in Brookes's semantics, from the particular model of heaps to work for arbitrary (cancellative) partial commutative monoids, assuming that primitive operations satisfy a locality conditions ensuring that they "mind their own business".
— [Hayman and Winskel 2008]: a "true concurrency" denotational semantics based on Petri nets. The intuitions expressed in the Separation Property and in Dijkstra's principle of loose connectedness are givien formal underpinnings using the concept of independence from Petri nets. This explains the CSL phenomenon "the order of certain interleavings doesn't matter for provable programs" in terms of a well established, prior notion of independence.
— [Hobor et al. 2008]: a Concurrent Separation Logic with first-class locks and threads, with a soundness proof based on an operational semantics. The work is notable for introducing two operational semantics, one a standard concurrent semantics and the other an "oracular" semantics which is closer to the intuitions of CSL; the logic is proven sound wrt the oracular semantics, and then the semantics is separately connected to the standard concurrent model. This division mirrors Brookes's use of global and local enabling relations, but is done employing operational rather than denotational semantics.
— [Vafeiadis 2011]: a soundness result formulated in an inductive manner that matches the stepwise "small-step" operational style of semantics. In contrast to the aforementioned works, Vafeiadis does not employ an additional semantics (the oracle semantics, or local enabling relation) which is more logical than a standard semantics; he connects the logic directly to the semantics in one step, with a novel interpretation of Hoare triples. This would be no win if the connection was more complex than the composition of two semantics, but Vafeiadis's proof is notable as well for being mathematically very elementary.
  Vafeiadis is also able to show soundness of CSL without precise invariants and without the conjunction rule, as well as of the original CSL.
— [Gotsman et al. 2011]: another operational approach, but where Vafeiadis uses a structural operational semantics à la Plotkin, which involves program rewriting, Gotsman et al use a fixed-program semantics that updates a program counter without rewriting the program; this is the kind of semantics often used in the temporal logic and model checking communities. With this form of semantics they are able to give a very direct expression of the Separation Property, and also a mathematically elementary proof. And, again like Vafeiadis, they show soundness for versions of separation logic with and without precise resource invariants.

There are extensions of relatives of CSL that are not conceptually based so directly on the Ownership+Separation ideas, and then it is more difficult to employ a proof technique that directly extends Brookes's original approach. In particular, CAP and its descendants and other related logics use the rule of parallel composition to reason about interfering processes. A very general account of this kind of situation is provided by the Views work [Dinsdale-Young et al. 2013]. As we mentioned above, the logic of Views is a close relative of Abstract Separation Logic, but its semantics is entirely different.

— Views considers a model of a separation logic assertion as an *abstract interpretation [Cousot and Cousot 1977]* of a concrete semantics, not the concrete semantics itself. There is a concretization function $\lfloor\cdot\rfloor$ taking an abstract state p to a set of concrete states $\lfloor p\rfloor$. Furthermore, $*$ is required to exist in the abstract semantics, but not in the concrete.
— A non-standard interpretation of Hoare triples is given that “bakes in” the frame rule of separation logic for every step of the execution of C: $\{p\}C\{q\}$ is true of views (abstract states) p and q iff for each trace $\overline{a}$ of C, there exists an $\overline{s}$ such that the triple $\{\lfloor s_i * r\rfloor\}a_i\{\lfloor s_{i+1} * r\rfloor\}$ is true in the usual interpretation of triples in the concrete semantics for all r, and $\overline{s}_0 = p$, and if $\overline{a}$ is finite then its last state is q.
— The soundness property is stated with respect to a standard concrete operational semantics, without employing an additional, more abstract operational or denotational semantics.

Considering separation logic states as abstract states, or views of concrete states, makes great sense. Even in the original heap splitting model, a finite “heaplet” $h : L \rightharpoonup_f V$ would rightly be considered as a portion of many actual machine states, not a machine state on its own; it was convenient and simple at the time not to formalize the fact that the heaplets were abstractions of many states; we can easily draw these portions of heap, without talking about all the enclosing global heaps. But as the states in the models become more intricate, the punning of separation logic models as “like” real states starts to break down. In a fractional permission model a heaplet $[10 \mapsto 47, 1/2]$ says that location 10 holds value 47 and has “permission 1/2”. The operational sensibility of permission models is much easier to grasp when such a heaplet is thought of as an abstract state rather than a concrete one on which programs execute. And beyond making intuitive sense, the Views semantics provides a validation of the rules of a version of CSL for models which allow for interference. The operative concept for Views is that of context: every step of the program must preserve all possible contexts (expressible by framing via $*r$). The Ownership Hypothesis and Separation Property together give a simple way to obtain contexts that are preserved, but Views both formalizes and exploits a more general setup in which there are others.

We remark that the proof technique of [Vafeiadis 2011], discussed above, also takes the tack of baking in the frame rule. It is similarly applicable to CAP and other logics where the Separation Property and Ownership Hypothesis would not apply. Thus, we see that the conceptual foundation of CSL-like logics has broadened over time, and with that logics that exploit the broadening have appeared.

Another trend in work on separation logic has been towards taking an axiomatic perspective, where what counts as a model of $*$ is subject to axioms rather than fixing on a single model to work with. This trend can be seen in Abstract Separation Logic and Views, as well as in other advanced logics such as Iris and HoCAP where the programmer (or human verifier) gets to “pick a monoid”. The axiomatic perspective has been taken further still in recent work on Concurrent Kleene Algebra [Hoare et al. 2011; Hoare et al. 2014]: they abstract not only from the semantics of logical assertions, but even from the semantics of programs. In brief:

— A CKA is an ordered monoid (;, `skip`) and an ordered commutative monoid (‖, `skip`), linked by the exchange law

$$(p \parallel r); (q \parallel s) \sqsubseteq (p; q) \parallel (r; s).$$

(Variants on CKA can have more structure, such as involving meets or joins.)
— A concurrent separation logic can be derived from a CKA by making the definition $\{p\}c\{q\} \;=\; p; a \sqsubseteq q$.

— An operational semantics related to Milner's for CCS can be derived by making the definition $p \overset{a}{\to} q \ = \ p \sqsupseteq c; q$.
— The operational semantics and logic are automatically in unison; we don't need to prove a theorem connecting them.
— A number of concrete models (instances of the algebra) have been given, particularly ones based on a true concurrency view of the world where ; is weak sequential composition and $\|$ is a form of composition of pomsets.

Concurrent Kleene Algebra is more abstract than most other work on CSL, in that it does not depend on a particular semantics of programs. While abstract in this sense, the operational semantics that it derives has not been shown to cover all the cases yet that (say) the Views theory covers; see [O'Hearn et al. 2015] for further discussion. So while CKA is more general that Views or Abstract Separation Logic on some dimensions, it is not (known to be) on others. However, it is a beautifully simple theory which is remarkable for the unification of logic and operational semantics it achieves when it is applicable, and this can be taken as an inspiration or a starting point for further work.

4. DEVELOPMENTS IN MECHANIZED VERIFICATION

Mostly-Automatic Verification. Smallfoot [Berdine et al. 2005], the first separation logic verification tool, included support for CSL from the beginning. The user would input resource invariants and other annotations such as procedure pre/post specs, and then Smallfoot would attempt to construct a program proof. For the pointer-transferring buffer, given the resource invariant and pre/post specs for the `put` and `get` operations it can verify pointer safety and race freedom of the client code. Smallfoot used a decidable fragment of separation logic oriented to linked lists and simple trees, and implemented a special theorem prover for this fragment.

Smallfoot is an example of a verifier where the programmer helps the tool along by inserting (some) annotations, but then the verifier behaves automatically. There are a number of other tools for mostly-automatic verification of this kind, which extend or build on the reasoning in CSL.

— *SmallfootRG* [Calcagno et al. 2007b] is a verifier for a marriage of separation logic and the classic rely/guarantee method for concurrent programs [Vafeiadis and Parkinson 2007].
— *Heap Hop* [Villard et al. 2010] is another extension of Smallfoot. It implements the copyless message passing logic of [Villard et al. 2009] and checks their protocols using certain automata.
— *Chalice* [Müller and Summers 2016] uses an expressive permission system and also targets reasoning about deadlock. It is based on a variant or relative of separation called "implicit dynamic frames" [Parkinson and Summers 2012], and leverages an embedding into first order logic, then utilizing an SMT solver rather than a custom theorem prover like that used by Smallfoot.
— *VeriFast* [Jacobs et al. 2016] is an advanced mostly-automatic verifier. Where the previously-mentioned tools work for toy, illustrative programming langauges, VeriFast applies to C and Java programs. It has been used to produce proofs for programs ranging from object-oriented patterns to highly concurrent algorithms to systems programs. Examples of fine-grained concurrent programs proven include hand-over-hand locking on linked lists and lock-free queues.
— *Mezzo* [Balabonski et al. 2014] and *Asynchronous Liquid Separation Types* [Kloos et al. 2015] use novel type systems which incorporate ideas from CSL into a programming language. The mode of usage is similar to the other mostly-automatic verifiers

above, but the aim is to be less expressive (and less of a burden) and to weave the specifications in with the types of a programming language. Related ideas can be found earlier in Cyclone, and more recently in the ownership typing that happens in the Rust language. It seems as if there is a lot of room for experimentation and innovation in this space.

Interactive Verification. In interactive verification the human helps with the proof effort, commonly in a proof assistant such as Coq, HOL or Isabelle. With interactive verification it is possible to get closer to proofs of full functional correctness than it is with automatic verifiers, but the cost is higher. There are a number of embeddings of CSL and relatives in the Coq proof assistant, and more complex programs have been proven over time.

— [Feng et al. 2008] treat *interrupts and preemptive threads* as found in OS kernels. They use an ownership-transfer semantics where portions of state are transferred between interrupt handlers and threads. Pre-emptive thread libraries for locks, conditions variables and context switching, comprising some 300 lines of x86 assembly code, were machine certified; the proof took 82k lines of Coq, including more than 1k definitions and 1.8k lemmas and theorems
— [Sergey et al. 2015] report on the verification using the aforementioned Fine-grained CSL. They provide verifications of a variety of low-level algorithms including a CAS-lock, a Ticketed lock, a GC allocator, a non-blocking stack, and a concurrent spanning tree construction. Proofs range from hundreds to 2k lines of Coq. An emphasis is placed on reusability; for instance, the stack uses the GC allocator, which in turn uses a lock, but the stack uses the spec of the allocator and the allocator uses the spec rather than the implementation of a lock.
— [Xu et al. 2016] verify key modules of a commercial preemptive OS kernel, the μC/OS-II kernel. The authors report that the ownership transfer idea of CSL plays a key role in the specifications and proofs. Modules verified include the scheduler, interrupt handlers, message queues, and mutex locks. 1.k lines of C code is verified using 216k lines of Coq, including framework code. It took 4 person years to develop the framework, 1 person year to prove the first module (for message queues), and then the remaining modules, consisting around 900 lines of C code, were done in 6 person-months.
This is apparently the first commercial *pre-emptive* kernel to have been verified (a number of high profile non-premptive kernel verification efforts, such as the celebrated seL4 project, have gone before).

Automatic Program Analysis.. With a verification-oriented abstract interpreter the program annotations that a human would supply to a mostly-automatic verifier – such as loop invariants, and sometimes pre/post specs – are inferred using a variety of techniques such as fixed-point iteration, widening and narrowing, interpolation, and abduction. Naturally, a verification tool will be able to prove less when the human is not supplying annotations, but the corresponding gain is that the techniques can be wholly automatic, working on bare code without asking the human for help to get started. Thus, verification-oriented abstract interpreters can be deployed with less friction than can mostly-automatic verifiers.

Abstract interpretation with sequential separation logic has seen rather a lot of attention; see [Calcagno et al. 2011] and its references. There have been comparatively fewer works on abstract interpretation with CSL (admittedly, a very difficult problem).

— [Gotsman et al. 2007] were the first to show how to infer resource invariants in CSL. They use reachability in the abstract heap to decide how to divide the state between shared and thread-local after each critical region. Then, the invariants are discovered by convergence of a fixed point in the usual way of abstract interpretation.

— [Calcagno et al. 2009] also infers resource invariants, but differs from the work of Gotsman et al by using the concept of footprint rather than reachability to divide the state after critical regions. This allows them to prove some of the ownership transfer example, such as O'Hearn's pointer-transferring buffer, that cannot be proven with the technique of Gotsman et al. Footprints are approximated using the abductive inference technique of [Calcagno et al. 2011].
— [Botincan et al. 2012] describe an algorithm that takes as input a sequential program with a proof in separation logic and some additional annottions, and outputs a concurrent program with a proof in CSL. Abduction is used to perform the decomposiitons needed to do the CSL proof.

These works make good steps, but program analysis with CSL is underdeveloped compared to the work in mostly-automatic and interactive verification. This is an imbalance that is deserving of further attention in the research community, especially since the possibility for broad impact is, except possibly in the very long term, much greater with verification-oriented abstract interpreters than less automatic tools.

5. CONTEXT, *CIRCA* 2016

At the beginning of the paper we mentioned that, as of 2002-2003:

— few realistic concurrent programs had been subjected to proof, and
— the semantics of concurrency for "simple" shared memory programs was well developed, but we were lacking tractable models for shared memory programs with structured state accessed via pointer manipulation.

Expanding on the latter, we needed to be able to explain the connection between such concepts as race freedom, the Separation Property and Ownership Hypothesis, and the sense of independence in Dijkstra's principle of loosly connected processes. It was easy to define *some* model for a concurrent language, but not one that let us delve into these sorts of properties; and they were all intimately related to the initial Concurrent Separation Logic.

Fast-forward to 2016 and, on the verification side:

— many small but realistic and increasingly intricate concurrent programs have been verified interactively and semi-automatically, and there are tens of implemented tools to do so, and
— a significant portion of an industrial, pre-emptive OS kernel has been verified.

These advances are beyond what we would dared to imagine in 2002-2003. Shortly after CSL was introduced, we were given the friendly challenge by Doug Lea (author of java.util.concurrent, a library full of advanced tricks) to verify optimistic, non-blocking algorithms. These algorithms seemed beyond the intuitive reach of the initial CSL techniques, and we wrestled with them for quite some time (see, e.g., [Parkinson et al. 2007] for early struggles). But with perseverence and insight the field as a whole has moved to the point where mechanized verification of another of these algorithms now comes as no surprise. Concerning OS verification, as soon as CSL was available Berdine and O'Hearn became keenly interested in the problem of attacking a microkernel; microkernel designs seemed to fit CSL conceptually, but verification technology for concurrent programs was not sufficiently developed, at least as far as we knew, to make such an effort feasible as of 2002-2003.

While the distance we have come in verification is very positive, to keep some perspective on what has been achieved it is important to stress that the tools referred to here are from the research sector. We have not yet seen concurrent software routinely verified within industry, by industry and not by researchers from either academia or industry; that is, verification of concurrent software is not yet *deployed*.

On the semantics side, there has been a flurry of development. We have models supporting increasingly simple and powerful proof methods. Soundness has been proven for concurrency logics not only with higher-order predicates but even with higher-order store obtained from storing resource invariants. The semantic models incorporate separation and interference at the same time. And there are general frameworks like in the Views work which give general conditions that allow to formulate concurrent separation logics for a given situation.

So there has been significant progress in logic and semantics. But there is much more work to do.

One important direction in pure theory is unification. The diagram in Figure 1 showing the CSL family tree shows a lot of variation, and it would be valuable to bring to a form of conclusion all of the developments on non-standard models and variations of CSL. The Views and Concurrent Kleene Algebra work both provide interesting starting points. There are also a lot of related ideas swirling around in models for weak memory and for distributed databases that have not been fully worked out. It seems that there is good and possibly deep theoretical work to be done.

As we have seen there has been a lot of progress in mechanized verification of concurrent programs; but there has been less in automatic program analysis. With program analysis we would like to give programmers feedback without requiring annotations, say by trying to prove specific integrity properties (such as memory safety or race freedom); annotations can help the analysis along, but are not needed to get started, and this greatly eases broad deployment. A number of prototype concurrency analyses based on CSL have been developed, as we mentioned in the previous section, but there has been much more work applying sequential separation logic to program analysis. For example, the Infer program analyser [Calcagno et al. 2015], which is in production at Facebook, is applied automatically on a daily basis to hundreds and sometimes thousands of modifications made to the Facebook code bases; it uses sequential but not concurrent separation logic. To make advanced program analysis for concurrency which brings value to programmers in the real world is in the main an open problem, and not an easy one.

Finally, we would like to say that CSL is but one part of work that goes on in the broader field of concurrency verification and analysis. There is much interesting work happening in dynamic concurrency analysis (e.g., the T-SAN tool), in symbolic model checking for concurrent software (e.g. CBMC), in modelling weak memory (e.g., the Herd tool), in logics and model checkers for models of distributed systems (e.g., TLA), and on more closely related logics and tools (e.g., VCC). In this retrospective paper we have not discussed or even referenced work other than that related to CSL, because to do justice to the other work would take too much more time and space. But, if we were to look at the state of the broader field in 2016 versus 2002 we would similarly see quite a significant level of advance, particularly in tools. While that is certainly positive and a cause for optimism, making concurrency theory+logic+tools simple and tractable enough to be deployed is still a great challenge, and a most worthwhile one because concurrent programs remain amongst the most difficult to understand.

ACKNOWLEDGEMENTS. Thanks to Mike Mislove and Prakash Panangaden for inviting us to write this article, and to Philippa Gardner, Matthew Parkinson and Ilya Sergey for discussion on some of the recent developments.

REFERENCES

Thibaut Balabonski, François Pottier, and Jonathan Protzenko. 2014. Type Soundness and Race Freedom for Mezzo. In *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*. 253–269. DOI:http://dx.doi.org/10.1007/978-3-319-07151-0_16

Christian J. Bell, Andrew W. Appel, and David Walker. 2010. Concurrent Separation Logic for Pipelined Parallelization. In *SAS (LNCS)*, Radhia Cousot and Matthieu Martel (Eds.), Vol. 6337. Springer, 151–166.

Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. 2005. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *FMCO (LNCS)*, Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem P. de Roever (Eds.), Vol. 4111. Springer, 115–137.

Richard Bornat. 2000. Proving Pointer Programs in Hoare Logic. In *MPC (LNCS)*, Roland Carl Backhouse and José Nuno Oliveira (Eds.), Vol. 1837. Springer, 102–126.

Richard Bornat, Cristiano Calcagno, Peter W. O'Hearn, and Matthew J. Parkinson. 2005. Permission accounting in separation logic. In *POPL*, Jens Palsberg and Martín Abadi (Eds.). ACM, 259–270.

Matko Botincan, Mike Dodds, and Suresh Jagannathan. 2012. Resource-sensitive synchronization inference by abduction. In *Proceedings of the 39th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2012, Philadelphia, Pennsylvania, USA, January 22-28, 2012*. 309–322. DOI:http://dx.doi.org/10.1145/2103656.2103694

John Boyland. 2003. Checking Interference with Fractional Permissions. In *SAS (LNCS)*, Radhia Cousot (Ed.), Vol. 2694. Springer, 55–72.

Stephen D. Brookes. 1996. Full Abstraction for a Shared-Variable Parallel Language. *Inf. Comput.* 127, 2 (1996), 145–163. DOI:http://dx.doi.org/10.1006/inco.1996.0056

Stephen D. Brookes. 2006. Variables as Resource for Shared-Memory Programs: Semantics and Soundness. *Electr. Notes Theor. Comput. Sci.* 158 (2006), 123–150. DOI:http://dx.doi.org/10.1016/j.entcs.2006.04.008

Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. 2015. Open-sourcing Facebook Infer: Identify bugs before you ship. (2015). https://code.facebook.com/posts/1648953042007882/open-sourcing-facebook-infer-identify-bugs-before-you-ship/.

Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. 2011. Compositional Shape Analysis by Means of Bi-Abduction. *J. ACM* 58, 6 (2011), 26. DOI:http://dx.doi.org/10.1145/2049697.2049700 Preliminary version appeared in POPL'09.

Cristiano Calcagno, Dino Distefano, and Viktor Vafeiadis. 2009. Bi-abductive Resource Invariant Synthesis, See Hu [2009], 259–274. DOI:http://dx.doi.org/10.1007/978-3-642-10672-9_19

Cristiano Calcagno, Peter W. O'Hearn, and Hongseok Yang. 2007a. Local Action and Abstract Separation Logic. In *LICS*. IEEE Computer Society, 366–378.

Cristiano Calcagno, Matthew J. Parkinson, and Viktor Vafeiadis. 2007b. Modular Safety Checking for Fine-Grained Concurrency. In *SAS (LNCS)*, Hanne Riis Nielson and Gilberto Filé (Eds.), Vol. 4634. Springer, 233–248.

Patrick Cousot and Radhia Cousot. 1977. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. 238–252.

Pedro da Rocha Pinto, Thomas Dinsdale-Young, Philippa Gardner, and Julian Sutherland. 2016. Modular Termination Verification for Non-blocking Concurrency. In *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*. 176–201. DOI:http://dx.doi.org/10.1007/978-3-662-49498-1_8

E. W. Dijkstra. 1968. Cooperating Sequential Processes. In *Programming Languages*, F. Genuys (Ed.). Academic Press, 43–112. Reprinted in [**?**].

Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *40th POPL*. 287–300.

Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010a. Concurrent Abstract Predicates. In *24th ECOOP*. 504–528.

Thomas Dinsdale-Young, Philippa Gardner, and Mark J. Wheelhouse. 2010b. Abstraction and Refinement for Local Reasoning. In *3rd VSTTE*. 199–215.

Mike Dodds, Xinyu Feng, Matthew J. Parkinson, and Viktor Vafeiadis. 2009. Deny-Guarantee Reasoning. In *18th ESOP*. 363–377.

Mike Dodds, Suresh Jagannathan, Matthew J. Parkinson, Kasper Svendsen, and Lars Birkedal. 2016. Verifying Custom Synchronization Constructs Using Higher-Order Separation Logic. *ACM Trans. Program. Lang. Syst.* 38, 2 (2016), 4. DOI:http://dx.doi.org/10.1145/2818638

Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. 2007. On the Relationship Between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *16th ESOP*. 173–188.

Xinyu Feng, Zhong Shao, Yuan Dong, and Yu Guo. 2008. Certifying low-level programs with hardware interrupts and preemptive threads. In *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation, Tucson, AZ, USA, June 7-13, 2008*. 170–182. DOI: http://dx.doi.org/10.1145/1375581.1375603

Ming Fu, Yong Li, Xinyu Feng, Zhong Shao, and Yu Zhang. 2010. Reasoning about Optimistic Concurrency Using a Program Logic for History. In *CONCUR (Lecture Notes in Computer Science)*, Paul Gastin and François Laroussinie (Eds.), Vol. 6269. Springer, 388–402.

Alexey Gotsman, Josh Berdine, and Byron Cook. 2011. Precision and the Conjunction Rule in Concurrent Separation Logic. *Electr. Notes Theor. Comput. Sci.* 276 (2011), 171–190.

Alexey Gotsman, Josh Berdine, Byron Cook, and Mooly Sagiv. 2007. Thread-modular shape analysis. In *PLDI*, Jeanne Ferrante and Kathryn S. McKinley (Eds.). ACM, 266–277.

Christian Haack, Marieke Huisman, and Clément Hurlin. 2008. Reasoning about Java's Reentrant Locks. In *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*. 171–187. DOI: http://dx.doi.org/10.1007/978-3-540-89330-1_13

Christian Haack and Clément Hurlin. 2008. Separation Logic Contracts for a Java-Like Language with Fork/Join. In *Algebraic Methodology and Software Technology, 12th International Conference, AMAST 2008, Urbana, IL, USA, July 28-31, 2008, Proceedings*. 199–215. DOI: http://dx.doi.org/10.1007/978-3-540-79980-1_16

Jonathan Hayman and Glynn Winskel. 2008. Independence and Concurrent Separation Logic. *Logical Methods in Computer Science* 4, 1 (2008).

C. A. R. Hoare. 1972. Towards a theory of parallel programming. In *Operating Systems Techniques*, Hoare and Perrot (Eds.). Academic Press.

Tony Hoare, Bernhard Möller, Georg Struth, and Ian Wehrman. 2011. Concurrent Kleene Algebra and its Foundations. *J. Log. Algebr. Program.* 80, 6 (2011), 266–296.

Tony Hoare, Stephan van Staden, Bernhard Möller, Georg Struth, Jules Villard, Huibiao Zhu, and Peter W. O'Hearn. 2014. Developments in Concurrent Kleene Algebra. In *Relational and Algebraic Methods in Computer Science - 14th International Conference, RAMiCS 2014, Marienstatt, Germany, April 28-May 1, 2014. Proceedings*. 1–18. DOI: http://dx.doi.org/10.1007/978-3-319-06251-8_1

Aquinas Hobor, Andrew W. Appel, and Francesco Zappa Nardelli. 2008. Oracle Semantics for Concurrent Separation Logic. In *ESOP (LNCS)*, Sophia Drossopoulou (Ed.), Vol. 4960. Springer, 353–367.

Zhenjiang Hu (Ed.). 2009. *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*. LNCS, Vol. 5904. Springer.

Samin S. Ishtiaq and Peter W. O'Hearn. 2001. BI as an Assertion Language for Mutable Data Structures. In *POPL*. 14–26.

Bart Jacobs, Jan Smans, and Frank Piessens. 2016. VeriFast project website. (2016). https://people.cs.kuleuven.be/ bart.jacobs/verifast/.

C. B. Jones. 1983. Specification and design of (parallel) programs. (1983). *IFIP Conference*.

Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. 2015. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 637–650. DOI: http://dx.doi.org/10.1145/2676726.2676980

Johannes Kloos, Rupak Majumdar, and Viktor Vafeiadis. 2015. Asynchronous Liquid Separation Types. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*. 396–420. DOI: http://dx.doi.org/10.4230/LIPIcs.ECOOP.2015.396

Jinjiang Lei, Zongyan Qiu, and Zhong Shao. 2014. Trace-Based Temporal Verification for Message-Passing Programs. In *2014 Theoretical Aspects of Software Engineering Conference, TASE 2014, Changsha, China, September 1-3, 2014*. 10–17. DOI: http://dx.doi.org/10.1109/TASE.2014.14

Hongjin Liang and Xinyu Feng. 2016. A program logic for concurrent objects under fair scheduling. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*. 385–399. DOI: http://dx.doi.org/10.1145/2837614.2837635

Peter Müller and Alex Summers. 2016. Chalice project website. (2016). http://www.pm.inf.ethz.ch/research/chalice.html.

Peter W. O'Hearn. 2007. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.* 375, 1-3 (2007), 271–307. Prelim version appeared in the proceedings of CONCUR'04.

Peter W. O'Hearn, Rasmus Lerchedahl Petersen, Jules Villard, and Akbar Hussain. 2015. On the relation between Concurrent Separation Logic and Concurrent Kleene Algebra. *J. Log. Algebr. Meth. Program.* 84, 3 (2015), 285–302. DOI:http://dx.doi.org/10.1016/j.jlamp.2014.08.002

Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. 2001. Local Reasoning about Programs that Alter Data Structures. In *CSL (LNCS)*, Laurent Fribourg (Ed.), Vol. 2142. Springer, 1–19.

S. Owicki and D. Gries. 1976a. An axiomatic proof technique for parallel programs. *Acta Informatica* 19 (1976), 319–340.

S. Owicki and D. Gries. 1976b. Verifying properties of parallel programs: An axiomatic approach. *Comm. ACM* 19, 5 (1976), 279–285.

David Michael Ritchie Park. 1979. On the Semantics of Fair Parallelism. In *Abstract Software Specifications (Lecture Notes in Computer Science)*, Dines Bjørner (Ed.), Vol. 86. Springer, 504–526.

Matthew J. Parkinson, Richard Bornat, and Cristiano Calcagno. 2006. Variables as Resource in Hoare Logics. In *21th IEEE Symposium on Logic in Computer Science (LICS 2006), 12-15 August 2006, Seattle, WA, USA, Proceedings*. 137–146. DOI:http://dx.doi.org/10.1109/LICS.2006.52

Matthew J. Parkinson, Richard Bornat, and Peter W. O'Hearn. 2007. Modular verification of a non-blocking stack. In *POPL*, Martin Hofmann and Matthias Felleisen (Eds.). ACM, 297–302.

Matthew J. Parkinson and Alexander J. Summers. 2012. The Relationship Between Separation Logic and Implicit Dynamic Frames. *Logical Methods in Computer Science* 8, 3 (2012). DOI:http://dx.doi.org/10.2168/LMCS-8(3:1)2012

A. Pnueli. 1981. The temporal semantics of concurrent programs. (1981). *Theoretical Computer Science*, 13(1), 45–60.

D. Pym, P. O'Hearn, and H. Yang. 2004. Possible worlds and resources: the semantics of BI. *Theoretical Computer Science* 315, 1 (2004), 257–305.

Azalea Raad, Jules Villard, and Philippa Gardner. 2015. CoLoSL: Concurrent Local Subjective Logic. In *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*. 710–735. DOI:http://dx.doi.org/10.1007/978-3-662-46669-8_29

John C. Reynolds. 2000. Intuitionistic Reasoning about Shared Mutable Data Structure. In *Millennial Perspectives in Computer Science (Cornerstones of Computing)*, Jim Davies, Bill Roscoe, and Jim Woodcock (Eds.). Palgrave Macmillan.

John C. Reynolds. 2002. Separation Logic: A Logic for Shared Mutable Data Structures. In *LICS*. IEEE Computer Society, 55–74.

Ilya Sergey, Aleksandar Nanevski, and Anindya Banerjee. 2015. Mechanized verification of fine-grained concurrent programs. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*. 77–87. DOI:http://dx.doi.org/10.1145/2737924.2737964

Kasper Svendsen, Lars Birkedal, and Matthew J. Parkinson. 2013. Joins: A Case Study in Modular Specification of a Concurrent Reentrant Higher-Order Library. In *ECOOP 2013 - Object-Oriented Programming - 27th European Conference, Montpellier, France, July 1-5, 2013. Proceedings*. 327–351. DOI:http://dx.doi.org/10.1007/978-3-642-39038-8_14

Viktor Vafeiadis. 2011. Concurrent Separation Logic and Operational Semantics. *Electr. Notes Theor. Comput. Sci.* 276 (2011), 335–351.

Viktor Vafeiadis and Chinmay Narayan. 2013. Relaxed separation logic: a program logic for C11 concurrency. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. 867–884. DOI:http://dx.doi.org/10.1145/2509136.2509532

Viktor Vafeiadis and Matthew J. Parkinson. 2007. A Marriage of Rely/Guarantee and Separation Logic. In *CONCUR (LNCS)*, Luís Caires and Vasco Thudichum Vasconcelos (Eds.), Vol. 4703. Springer, 256–271.

Jules Villard, Étienne Lozes, and Cristiano Calcagno. 2009. Proving Copyless Message Passing, See Hu [2009], 194–209.

Jules Villard, Étienne Lozes, and Cristiano Calcagno. 2010. Tracking Heaps That Hop with Heap-Hop. In *TACAS (LNCS)*, Javier Esparza and Rupak Majumdar (Eds.), Vol. 6015. Springer, 275–279.

Fengwei Xu, Ming Fu, Xinyu Feng, Xiaoran Zhang, Hui Zhang, and Zhaohui Li. 2016. A Practical Verification Framework for Preemptive OS Kernels. (2016). CAV.