Scalable Specification and Reasoning: Technical Challenges for Program Logic

Peter W. O'Hearn

Queen Mary, University of London

Abstract. If program verification tools are ever to be used widely, it is essential that they work in a modular fashion. Otherwise, verification will not scale. This paper discusses the scientific challenges that this poses for research in program logic, and suggests some test problems that would be useful in measuring advances on modular reasoning.

1 Introduction

Software verification has seen an upsurge of interest in recent years. Partly this is a result of a convergence that has resulted from maturation of proof tools and lowering of aims, from full behavioural specifications to partial (often safety) properties of a system. Prominent examples include the SLAM model checker and the ESC/Java tool. But modularity remains a problem.

For very special kinds of program, often used in the safety-critical realm, modular specification methods are indeed available. But, modularity is gained there by restricting the programming model, essentially to features where traditional Floyd-Hoare logic works well. Typically, this is for programs without pointers (in various of their guises) or concurrency. Verification tools aimed at widely-used programming languages sometime do work in a modular way, on a procedure by procedure basis, but are (intentionally) unsound because it is not clear how to work modularly in a sound manner in the presence of pointer aliasing (e.g., [19, 8]). Other tools (e.g., current software model checkers) simply diverge in the face of deep, heap-intensive properties.

There is a sense in which an assertion language based on classical first-order or higher-order logic is powerful enough for all of our specification needs. It can say all we want – but it does not always say so well. What one wants is a tractable specification formalism and, particularly when one considers programs with pointers and concurrency, the reasoning with classical logic can become so complex as to be detached from computational intuition. The best way to illustrate this claim is with examples, and I consider three, describing what the more general technical challenges are as we go along. Some relevant work on Separation Logic [29] is described, and the promise of and challenges for this approach are discussed. Finally, some possible test codes are mentioned, which might be used to judge progress on a Program Verifier.

There are many obstacles facing any Program Verifier grand challenge project [12] – particularly, the strength of theorem provers – and I am not saying that

full solutions to the problems I discuss are necessary for it to have some success. But, I would hope that any such project would take on the problems posed by popular languages. My aim here is to communicate some unsolved problems in program logic which, if progress were made on them, could have a considerable positive impact on the project.

2 Framing and Indirection

I begin with a simple program and consider how one might specify it using traditional Floyd-Hoare logic. The specification is found to be unsatisfactory, and then is amended to provide a technically correct one. It is then argued that this technically correct specification is conceptually wrong.

2.1 An Incorrect Specification

Consider a procedure for disposing a tree, held as a linked structure in memory.

```
procedure DispTree(p)
local i, j;
if  p≠nil then
    i = p→l ; j:= p→r;
    DispTree(i);
    DispTree(j);
    dispose(p)
```

This is the expected procedure that walks a tree, recursively disposing left and right subtrees and then the root pointer. It uses a representation of tree nodes with left, right and data fields, and the empty tree is represented by nil.

A first attempt at a specification might be something like

$$\{\mathsf{tree}(p) \wedge \mathtt{reach}(p, n)\}\ \ \mathtt{DispTree}(p)\ \{\neg\mathtt{allocated}(n)\}$$

assuming that we have defined the predicates that say when p points to a (binary) tree in memory, when n is reachable (following l and r links) from p, and when n is allocated. This spec says that any node n which is in the tree pointed to by p is not allocated on conclusion.

While this specification says part of what we would like to say, it leaves too much unsaid. It does not say what the procedure does to nodes that are not in the tree; we have left out the notorious *frame axioms* [17].

The result is that, while the specification is something that we would expect to be true of the procedure, it is too weak to use at many call sites. For example, consider the first recursive call, `DispTree`(i), to dispose the left subtree. If we use the specification (instantiating p by i) as an hypothesis, in the usual way when reasoning about recursive procedures [9], then we have a problem. For, the specification does not rule out the possibility that the procedure call alters the right subtree j, perhaps creating a cycle or even disposing some of its nodes. As a consequence, when we come to the second call `DispTree`(j), we will not know

that the required $\mathsf{tree}(j)$ part of the precondition will hold. So our reasoning will get stuck.

The moral of this story is that

> *if one does not have some way of representing or inferring frame axioms, then the proofs of even simple programs with procedure calls will not go through.*

The DispTree program makes this point especially vivid because of its use of recursion, where the spec and the call sites have to get along:

> *for recursive programs attention to framing is essential if one is to obtain strong enough induction hypotheses.*

The problem does not depend on having low-level operations such as pointer disposal. For example, specifying tree copying leads to similar difficulties.

2.2 An Unfortunate Fix

How can we fix the specification of DispTree? Here is my attempt:

$\{\mathsf{tree}(p) \wedge \mathtt{reach}(p, n) \wedge \neg\mathtt{reach}(p, m) \wedge \mathtt{allocated}(m) \wedge m.f = m' \wedge$
$\neg\mathtt{allocated}(q)\}$
$\mathtt{DispTree}(p)$
$\{\neg\mathtt{allocated}(n) \wedge \neg\mathtt{reach}(p, m) \wedge \mathtt{allocated}(m) \wedge m.f = m' \wedge$
$\neg\mathtt{allocated}(q)\}$

This says, in addition, that any allocated cell not reachable from p has the same contents in memory and that any previously unallocated cell remains unallocated. The additional clauses are the frame axioms. (I am assuming that m, m', n and q are auxiliary variables, guaranteed not to be altered. The reason why, say, the predicate $\neg\mathtt{allocated}(q)$ could conceivably change, even if q is constant, is that the `allocated` predicate refers to a behind-the-scenes heap component. f is used in the spec as an arbitrary field name.)

I *believe* that this specification is strong enough to prove the procedure, but I have never attempted to carry out a proof. It would be complex. But, more importantly, I believe that the specification is badly wrong from a conceptual point of view.

The problem is not that we cannot specify DispTree at all, but rather is that final specification makes ugly statements about what is not reachable and what is not allocated that have, really, nothing to do with the program. Programmers *think locally*, and when reasoning about a program they concentrate on the resources that are relevant to its correct operating [23]. The need to state these frame axioms explicitly is violently at odds with programming intuition. So, even if technically alright, I view such a specification as wrong, a symptom of a problem in program logic.

2.3 The Frame Problem

The frame problem is that, traditionally, an inordinate amount of effort needs to be spent specifying what a program doesn't change, so much so that these frame axioms distract from the main concern – what changes [17, 28, 4]. In the absence of pointers what doesn't change can be succinctly summarized using modifies clauses, which list the program variables corresponding to locations that can be altered by a program. But, in the presence of pointers of other forms of indirect addressing the relevant locations are not always directly named by program variables, and the idea of modifies clause is then much more difficult to make work. The unhappy consequence is that sound, modular specification methods are lacking for widely-used programming languages such as C and Java.

A full solution to the frame problem would allow us to make a positive statement about what changes, like in our first, faulty, specification, with the frame axioms coming along for free. A partial solution would at least let us represent the frame axioms compactly and intuitively.

The frame problem is extremely irritating. When you see it, you expect that there should be some sort of easy solution. It should be possible for a specification to say just what is relevant, for the rest (the frame axioms) to come along for free. I have often felt that way. However, it must be admitted that, over 35 years since it was identified [17], there is still no satisfactory solution.

Why do I believe that there is no solution yet, given that the problem has been so intensely studied in AI? One reason is that, as far as I know, if you apply any of the prominent AI techniques to the problem of reasoning about low-level programs then what you get is much more complex than Separation Logic's approach; nothing approaching the "small axioms" of [23] pops out. While saying this I admit to great respect for some of the works in the AI literature, particularly the extremely clear work of Reiter [28]. But, we should demand of any claimed general solution to the frame problem that it demonstrate itself not just on the favourite toy AI examples (Yale shooting problem, etc), but on those most real agents of change, computer programs.

The frame problem is stated above in a decidedly negative manner. It is perhaps useful to take a more positive point of view:

> *When specifying a program, it should be possible to concentrate exclusively on the information (data, resources, etc) that is relevant to its correct operating. Any information it is independent of should not have to be mentioned.*

Although the frame problem is irritating, it is genuine, and a central problem in modular reasoning. But it is not the whole story.

3 Independence, Interference and Concurrency

Reasoning about concurrency is a subject that has received significant attention, and for good reason. The tremendous number of potential interactions between

concurrent processes makes concurrent programs hard to grasp; a successful Program Verifier could provide considerable help to the concurrent programmer.

But, though it has received much attention, the difficulties that the theory meets on even simple examples are not as widely appreciated as perhaps they ought to be. To illustrate, I consider a very simple program: parallel mergesort.

$\{\mathit{array}(a,i,j)\}$
`procedure` $\texttt{ms}(a,i,j)$
`local` $m := (i+j)/2;$
`if` $i < j$ `then`
  $\big(\texttt{ms}(a,i,m) \parallel \texttt{ms}(a,m+1,j)\big);$
  $\texttt{merge}(a,i,m+1,j);$
$\{\mathit{sorted}(a,i,j)\}$

For simplicity this specification just says that the final array is sorted, not that it is a permutation of the initial array.

Now, this program displays a very simple form of concurrency: *disjoint concurrency*. The recursive calls are completely independent, because they act on disjoint array segments. And yet, the program causes immediate difficulties for all of the best known proof methods.

Part of the difficulty is similar to the frame problem. With the given precondition and postcondition we do not know, just from the spec, that the leftmost parallel call $\texttt{ms}(a,i,m)$ does not alter the array outside the segment from i to m. It might interfere with the right call. Similarly, the right call might alter the segment from i to m.

Hoare had provided a simple and beautiful rule for disjoint concurrency [10]

$$\frac{\{P\}C\{Q\} \qquad \{P'\}C'\{Q'\}}{\{P \wedge P'\}C \parallel C'\{Q \wedge Q'\}}$$

where C does not modify any variables free in P', C', Q', and conversely. Unfortunately, using this rule we cannot reason about the parallel calls in mergesort, because Hoare logic treats array-component assignment globally, where an assignment to $a[i]$ is viewed as an assignment to the entire array

$$\{P[(a \mid i\colon E)/a]\}\; a[i] := E\; \{P\}$$

In this view the two parallel calls to `ms` are judged to be altering the *same* variable, a.

Cliff Jones has proposed a powerful approach to reasoning about concurrency, in his rely-guarantee formalism [14] (see also, [18]). For this example, we would add two conditions to the pre/post specification, formalizing the

- **Rely**: No other process touches my array segment $\mathit{array}(a,i,j)$; and
- **Guarantee**: I do not touch any storage outside my segment $\mathit{array}(a,i,j)$.

The Guarantee condition here is something like a frame axiom. The Rely, however, goes beyond the frame issue (one might fancifully consider it a kind of inverse frame axiom).

The point of this example is that it illustrates a breakdown of modularity. The guarantee condition (when formalized) talks about parts of the array not touched by a procedure call. In the worst case, this would have to be extended to other parts of memory than the single array given as a parameter. The issue is not just the cost for individual steps of reasoning, but rather that the rely and guarantee conditions, which are present to deal with subtle issues of interference, complicate the specification itself, even when no interference is present.

I have focussed on rely-guarantee here because is rightly lauded as providing a compositional approach to reasoning about concurrency.[1] My point is that compositionality in program text does not guarantee locality in reasoning about resources such as program state: compositional reasoning can be extremely global. Also, I used a pre/post specification just because it is appropriate to the example, but the same modularity problem I have described here arises as well in temporal logics.

My remarks on the rely-guarantee method should be taken in the right spirit. Indeed, they agree with a criticism of it lodged by Jones himself [15]. What he wants, and what I want, is a way to use complex methods where necessary to deal with interference when it is present, but to contain this complexity and default to simpler specification forms for interfaces between components that do not interfere with one another. The desire is to prevent *interference flooding*, where the mere possibility of interference complicates the specification notation, even in situations where there is a great degree of independence. Perhaps a marriage of rely-guarantee and Separation Logic (Section 5) is possible.

4 Information Hiding

Pointers can wreak havoc with data abstraction. It is difficult to keep track of aliases, different copies of the same address, and so it is difficult to know when there are no pointers into the internals of a module. This problem has received attention in the object-oriented types community in work on ownership and confinement [7, 1], stemming Hogg's colorful declaration "that objects provide encapsulation is the big lie of object-oriented programming [13]". Further difficulties, beyond confinement, are caused by low-level features such as address arithmetic and storage deallocation.

A good example is a resource management module, that provides primitives for allocating and deallocating resources, which are held in a local free list. A client program should not alter the free list, except through the provided primitives; for example, the client should not tie a cycle in the free list. In short, the free list is owned by the manager, and it is (intuitively) hidden from client programs. However, it is entirely possible for a client program to hold an alias to an element of the free list, after a deallocation operation is performed; intuitively, the "ownership" of a resource transfers from client to module on disposal, even if

[1] As an aside, I am unsure of how the classic Owicki-Gries method [26] would deal with an example like this, because I am unsure how the notion of interference with a proof meshes with using triples as hypotheses in the treatment of recursive procedures.

many aliases to the resource continue to be held by the client code. In a language that supports address arithmetic the potential difficulties are compounded: the client might intentionally or unintentionally obtain an address used in an internal representation, just by an arithmetic calculation.

As an example, suppose that we have written our own memory manager, with operations $\texttt{alloc}(x)$ and $\texttt{free}(x)$ for allocating and deallocating records. Suppose that our implementation uses a free list in the usual way.

A first attempt at specification might be something like

$$\begin{array}{l}\big\{\texttt{allocated}(y) \land y.f = m \land \neg\texttt{allocated}(z)\big\}\\ \texttt{alloc}(x)\\ \big\{\texttt{allocated}(y) \land y.f = m \land \texttt{allocated}(x) \land y \neq x\\ \land (z \neq x \Rightarrow \neg\texttt{allocated}(z))\big\}\end{array}$$

$$\begin{array}{l}\big\{\texttt{allocated}(y) \land y.f = m \land \texttt{allocated}(x) \land y \neq x \land \neg\texttt{allocated}(z)\big\}\\ \texttt{free}(x)\\ \big\{\texttt{allocated}(y) \land y.f = m \land \neg\texttt{allocated}(x) \land y \neq x \land \neg\texttt{allocated}(z)\big\}\end{array}$$

where, in addition to saying that x is allocated or deallocated, I have included a lot of frame axioms. I admit to some unease, I am not sure I have got the frame axioms exactly right (echoing the discussion from earlier), but there is a further problem I want to show, so let us assume that these are indeed the correct frame axioms. Here, I am again assuming that all variables other than x are auxiliary variables that are guaranteed not to be changed, and that $\{x\}$ is the entire modifies set of the specs (modifies for variables, not heap cells).

The further problem is that this specification does not stop a user of the memory manager from corrupting the free list, breaking the abstraction. For example, a sequence of statements

$$\texttt{alloc}(x)\,;\ \ \texttt{free}(x)\,;\, x{\rightarrow}r := x$$

might tie a cycle in the free list, if the implementation uses the r field to point to the next record in the free list.

We can get around this problem by adding an invariant to the specifications. To each precondition and postcondition we add a predicate $\texttt{freelist}(\mathit{free})$ saying that variable free used by the manager points to a linked list without cycles, and where $\neg\texttt{allocated}(n)$ holds for each element in the list.

This fix, though, has come at great cost: we have exposed the invariant describing the ostensibly private storage of the memory management module. To see the cost, suppose a program makes use of n different modules. It would be unfortunate if we had to thread descriptions of the internal resources of each module through steps when reasoning about the program. Even worse than the proof burden would be the additional annotation burden, if we had to complicate specifications of user procedures by including descriptions of the internal resources of all modules that might be accessed. A change to a module's internal representation would necessitate altering the specifications of all other

procedures that use it. The resulting breakdown of modularity would doom any aspiration to scalable specification and reasoning.

Stated plainly,

> *information hiding should be the bedrock of modular reasoning, but it is difficult to support soundly*

and this presents a great challenge for research in program logic. Information hiding is much easier to achieve for simplified programming languages (without pointers and objects, or concurrency), and then classical approaches such as [11] are very appropriate. But for more expressive languages nothing approaching a canonical solution has emerged, despite some worthwhile work [20, 16, 21, 22].

5 Separation Logic

The Separation Logic specification of DispTree is just

$$\{\mathtt{tree}(p)\}\ \mathtt{DispTree}(p)\ \{\mathtt{empty}\}$$

which says that if you have a tree at the beginning (and nothing else) then you end up with the empty heap at the end. It deals with the recursive calls using an inference rule, the "frame rule", for inferring frame axioms [23]. Similarly, it's treatment of alllocation and disposal, using the "small axioms", is much easier to understand than the specification in the previous section here.

Just as the specification of DispTree avoids mentioning frame axioms, parallel mergesort can be treated without using rely and guarantee conditions [24]. The crucial part of the proof is the following proof figure for the parallel composition.

$$\begin{array}{cccc} & \{array(a,i,m) * & array(a,m{+}1,j)\} & \\ \{array(a,i,m)\} & & & \{array(a,m{+}1,j)\} \\ \mathtt{ms}(a,i,m) & & \| & \mathtt{ms}(a,m{+}1,j) \\ \{sorted(a,i,m)\} & & & \{sorted(a,m{+}1,j)\} \\ & \{sorted(a,i,m) * & sorted(a,m{+}1,j)\} & \end{array}$$

The use of the $*$ connective in $array(a,i,m) * array(a,m{+}1,j)\}$ implies that the array segments occupy separate memory, and we can then use a proof rule

$$\frac{\{P\}C\{Q\} \qquad \{P'\}C'\{Q'\}}{\{P * P'\}C \parallel C'\{Q * Q'\}}$$

that lets us reason independently about the two processes. The reason we can get away with just Hoare triple specifications here stems from an interplay between the separating conjunction, $*$, and a "tight" interpretation of Hoare triples [30], which ensures that well-specified processes mind their own business [6].

Further, there has been some progress on information hiding with Separation Logic [25, 27].

Given how well Separation Logic deals with the examples above, and how poorly specification with standard classical logic fares, you might think that

this paper has been a set-up. But it has not. I am not proposing that Separation Logic as it is is right for the Program Verifier project. It is a recent development, and there are too many outstanding problems that need more work. There are no general theorem provers for it yet, only some limited decidability results [2, 3]. The examples in this paper are optimal for Separation Logic, and things do not always go as well: we struggle, for example, when faced with recursive graph algorithms [5]. The logic is (presently) oriented to a fixed, low level of abstraction, making it not particularly suited for use in a developmental fashion. And, while it deals well with concurrent programs that exhibit a great degree of independence, it has not been convincingly demonstrated on tightly-coupled, racy programs, where the rely-guarantee method might be superior.

Rather, I wanted to show some difficulties as regards modularity that traditional program logic has on even some simple examples, and how it it is not impossible to do much better, at least on those examples.

6 Conclusion

Generally speaking, what I would like is to have is an elegant and very modular specification method, that worked on the design as well as code level, but that did not eschew features of popular programming languages or important programming idioms expressed in them. That is the challenge for program logic. In any case I, for one, do not know how to achieve it. The specific examples I have given in this paper, as well as illustrating the problems, might also be seen as little, initial tests for specification methods.

I have not mentioned model checking, but the modularity problems there are even more severe. For, software model checking attempts to relieve annotation burden by inferring certain specifications. More positively, we might hope that the development of modular methods of specification and proof could be used in model checking as well as in verification of annotated programs.

I would like to conclude by mentioning some larger test problems which should be challenging for a Program Verifier. These are intentionally not on the grand level of, e.g., "specify linux", but are (I think) beyond current technology and so are examples of intermediate problems that might prove useful in evolving Program Verifier technology.

Graph Algorithms. Graph algorithms must deal with sharing, which complicates both specifications and proofs. Even the simplest recursive algorithms are difficult to properly specify at the moment [5], and to have fully automatic proofs of such algorithms would be a very good step forward indeed.

A perhaps more stringent test would be to verify an implementation of graph-reduction for a functional programming language.

Resource Manager. Verify one of the leading memory manager implementations, such as Doug Lea's. Do so in a modular way where the internal representation (of the free lists) is not revealed in the interface specification. Ideally, provide a way to automatically check that client programs do not access the

module internals, and demonstrate this using list, tree and graph algorithms that call the manager.

Similarly, verify implementations of Java thread pools, and apply this to some web services that use thread pools to increase throughput.

java.util.concurrent. This library makes uses both pointers and concurrency, and employs a great many advanced tricks. Verify as much of it as possible.

Process Manager. This is a test for both concurrency and information hiding. A process manager (among other things) schedules different processes for execution, but the internal workings are (for the most part) hidden from the processes themselves. Concretely, a user process might believe in the rule of sequencing

$$\frac{\{P\}C\{Q\} \qquad \{Q\}C'\{R\}}{\{P\}C\,;C'\{R\}}$$

while in the implementation many processes are using the CPU at different times. Verify that a process manager correctly implements a sequential point of view within a process (thus achieving information hiding for control and not just data).

Of course, in the above, "verify" might mean to find bugs, alter, and then verify. Equally, a developmental approach which starts from a high-level specification and ends up with different code than an existing target would be just as well, though one would not want to pay a large performance penalty in the resulting code. In any case, everyone will have their own favourite problems, and collecting descriptions of them, and why they are difficult, will be very valuable for a challenge project. Indeed, it is plausible that an outcome of a Program Verifier project will not be a completely generic tool, but a list of specification idioms telling in what situations they are useful and what proof methods to apply. This is by analogy with the development of *patterns* as a pragmatic reaction to the unrealized ideal of a general language for reusable software components.

References

1. A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *J.ACM*, to appear, 2005.
2. J. Berdine, C. Calcagno, and P. O'Hearn. A decidable fragment of separation logic. Proceedings of FSTTCS, LNCS 3328, Chennai, December, 2004.
3. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. Draft, June 2005.
4. A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions of Software Engineering*, 21:809–838, 1995.
5. R. Bornat, C. Calcagno, and P.W. O'Hearn. Local reasoning, separation and aliasing. In the informal proceedings of *SPACE 2004*, January 2004.
6. S. D. Brookes. A semantics for concurrent separation logic. This Volume, Springer LNCS, *Proceedings of the 15th CONCUR*, London. August, 2004.

7. D. Clarke, J. Noble, and J. Potter. Simple ownership types for object containment. *Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 53-76, Springer LNCS 2072, 2001.
8. D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. CASSIS, pp108-128, 2004.
9. C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In E. Engler, editor, *Symposium on the Semantics of Algebraic Languages*, pages 102–116. Springer, 1971. Lecture Notes in Math. 188.
10. C. A. R. Hoare. Towards a theory of parallel programming. In Hoare and Perrot, editors, *Operating Systems Techniques*. Academic Press, 1972.
11. C.A.R. Hoare. Proof of correctness of data representations. Acta Informatica 4, 271-281, 1972.
12. C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
13. J. Hogg. Islands: aliasing protection in object-oriented languages. 6th OOPSLA, 1991.
14. C. B. Jones. Specification and design of (parallel) programs. *IFIP Conference*, 1983.
15. C. B. Jones. Wanted: A compositional approach to concurrency. In A. McIver and C. Morgan, editors, *Programming Methodology*, pages 1–15, 2003. Springer-Verlag.
16. K. R. M. Leino and G. Nelson. Data abstraction and information hiding. *ACM Trans. Program. Lang. Syst.*, 24(5):491–553, 2002.
17. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.
18. J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.
19. A. Moller and M. Schwartzbach. The pointer assertion logic engine. Proceedings of PLDI, 221-231, 2001.
20. P. Müller and A. Poetzsch-Heffter. Modular specification and verification techniques for object-oriented software components. In *Foundations of Component-Based Systems*. Cambridge University Press, 2000.
21. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular specification of frame properties in JML. *Concurrency and Computation: Practice and Experience*, 15:117–154, 2003.
22. D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *19th LICS*, 2004.
23. P. O'Hearn, J. Reynolds, and H. Yang. Local reasoning about programs that alter data structures. In *Proceedings of 15th Annual Conference of the European Association for Computer Science Logic*, LNCS, pages 1–19. Springer-Verlag, 2001.
24. P. W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR'04: 15th International Conference on Concurrency Theory*, volume 3170 of *Lecture Notes in Computer Science*, pages 49–67, London, August 2004. Springer. Extended version to appear in *Theoretical Computer Science*.
25. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *31st POPL*, pages 268–280, 2004.
26. S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, (19):319–340, 1976.
27. M. Parkinson and G. Bierman. Separation logic and abstraction. Proceedings of POPL, 2005.

28. R. Reiter. The frame problem in the situation calculus: a simple solution (sometimes) and a completeness result for goal regression. In V. Lifschitz, editor, *Artificial Intelligence and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380. Academic Press, 1991.
29. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pp 55-74, 2002.
30. H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *Foundations of Software Science and Computation Structures*, Springer LNCS 2303., 2002.