Scalable Specification and Reasoning: 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. Some recent work on separation logic is described, and test problems that would be useful in measuring advances on modular reasoning are suggested.

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 [2] and the ESC/Java static assertion checker [19, 14]. But modularity is a problem.

Modularity is essential for scalable specification and reasoning. If we find ourselves in a position where the specification of one program component must talk about all other components in a system, or the states of other components, then we will very quickly be overwhelmed by the complexity of specifications. Programming features such as pointers (in various of their guises), concurrency and reflection raise particularly challenging problems for program logic. Simple methods for achieving modularity, such as listing the variables that might change (using "modifies" clauses), are not sufficient for common programs written in widely-used languages, which feature complex and dynamically changing interconnections between program components.

The problem faced by program logic is not an in-principle one – being able to describe behaviours at all – but rather is one of tractability. For example, when one considers programs with pointers and concurrency, reasoning with traditional program 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 [46, 26, 37, 47] is described, and the promise of and problems for this approach are discussed. Finally, some wholly unresolved problems are mentioned.

There are many obstacles facing any Program Verifier challenge project [24] – 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. My aim here is just to communicate some unsolved problems in program logic which, if progress were made on them, could have a considerable positive impact.

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  $p \neq \mathsf{nil}$ then
    $i = p \to l$ ; $j := p \to r$;
    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

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

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* [33].

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 [22], 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 tree(j) part of the precondition will hold. So our reasoning will get stuck.

The moral of this story is that [37]

> *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:

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

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 [37]. 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 conceptually 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 [33]. 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, like in our first specification of `DispTree`, and for the rest (the frame axioms) to come along for free. I have often felt that way.

The frame problem has been intensely studied in AI, and there are too many papers to survey here; I mention only one, the extremely clear paper of Reiter [44], which can serve as a good introduction to the problem. Unfortunately, there has been little crossover work applying the techniques there to programs (a notable exception is [9]). Although the frame problem is irritating, it is genuine, and a central problem in modular reasoning. But it is not the whole story, as we shall see in later sections.

The frame problem is stated above in a decidedly negative manner. I prefer to take a more positive perspective [37]:

> *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.*

3 Separation Logic

The separation logic specification of `DispTree` is just

$$\{\mathsf{tree}(p)\}\ \texttt{DispTree}(p)\ \{\texttt{empty}\}$$

which says that if you have a tree at the beginning then you end up with the empty heap at the end. And the proof is very simple. The crucial part, in the `else` branch, looks like this:

```
$\{p \mapsto [l\colon x, r\colon y] * \mathsf{tree}(x) * \mathsf{tree}(y)\}$
   $i$ := $p \to l$;  $j$ := $p \to r$;
$\{p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)\}$
   DispTree($i$);
$\{p \mapsto [l\colon i, r\colon j] * \mathsf{tree}(j)\}$
   DispTree($j$);
$\{p \mapsto [l\colon i, r\colon j]\}$
   dispose $p$;
{empty}
```

After we enter the conditional statement we know that $p{\neq}\mathsf{nil}$, so that p is an allocated node that points to left and right subtrees occupying separate storage. Then the roots of the two subtrees are loaded into i and j. Notice how the proof steps then follow operational intuition. The first recursive call removes the left subtree, the second call removes the right subtree, and the final instruction removes the root pointer p. This verification is carried out using the procedure specification as an assumption, as in the usual treatment of recursive procedures in Hoare logic [22].

I have just given you a proof snippet in what is probably an unfamiliar formalism, so some explanation is in order. To understand separation logic intuitively you should think in terms of *heaplets*, portions of heap, rather than the whole global heap. The separating conjunction $P * Q$ holds of a given heaplet if it can be split into two disjoint heaplets, one of which satisfies P and the other of which satisfies Q. So, the assertion $p{\mapsto}[l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)$ describes a portion of heap with a pointer p that points to a record with l and r fields holding values i and j that themselves point to trees. The use of $*$ indicates that there is no overlap between p and i's tree and j's tree.

A question that often comes up is whether a pointer can go from one $*$-conjunct to another. The answer is yes. For instance, $p{\mapsto}[l\colon i, r\colon j]$ describes just a single cell, p, whose contents i and j point across $*$ into other heaplets in $p{\mapsto}[l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)$. it helps to use a graphical intuition: take a directed graph, and then draw a line, partitioning it in two. Some of the links in the graph will go over the partition. The p to the left of $\mapsto$ corresponds to the sources of links to targets i and j.

There is a subtle point in the specification of `DispTree` that the reader might have noticed: In order to get the empty heap in the postcondition the precondition must say that "p points to a tree, *and there are no other cells in the given heaplet*". For, if there were other cells then you could not conclude `empty`, those cells that were not originally in the tree would still be around. This "no other cells" aspect is treated implicitly in separation logic. The tree predicate satisfies the recursive specification

$$\begin{aligned}\mathsf{tree}(E) \iff & (E{=}\mathsf{nil} \wedge \ \ \mathtt{empty}) \\ & \vee (\exists x, y.\, E{\mapsto}l\colon x, r\colon y \ * \ \mathsf{tree}(x) \ * \ \mathsf{tree}(y))\end{aligned}$$

where the use of `empty` when $E{=}\mathsf{nil}$ leads, inductively, to $\mathsf{tree}(E)$ not having additional cells.

Finally, there is a crucial interplay between the separating conjunction and a "tight" interpretation of Hoare triples [37, 51, 50]. A specification $\{P\}C\{Q\}$ means that C will (if it terminates) transform a heaplet satisfying P into one satisfying Q. It does this transformation in an in-place fashion, leaving the global heap surrounding the input heaplet unchanged. This in-place aspect can be seen clearly in the proof steps. For instance, for the first recursive call to `DispTree` the precondition is $p{\mapsto}[l\colon i, r\colon j] * \mathsf{tree}(i) * \mathsf{tree}(j)$, and this does not match up with the overall specification, which would expect only $\mathsf{tree}(i)$. What we do is use the overall specification to replace $\mathtt{tree}(i)$ by $\mathtt{empty}$, obtaining $p \mapsto [l\colon i, r\colon j] * \mathtt{empty} * \mathtt{tree}(j)$, and then we can take one further step using the identity $\mathtt{empty} * P \leftrightarrow P$.

These intuitions about heaplets and in-place update are codified in an inference rule, the frame rule

$$\frac{\{P\}\ C\ \{Q\}}{\{R * P\}\ C\ \{R * Q\}}\ \text{ModifiesOnly}(C) \cap \text{free}(R) = \emptyset$$

The R here is a frame axiom. The idea of this rule is that if C works on a portion of heap described by P, then it will not alter any additional heaplet described by R. There is also a side condition which has to do with named variables; e.g., the i and j in `DispTree`. (It is an embarassment that the heap is treated more cleanly than simple variables here, and we hope someday to get rid of the variable conditions altogether; see [10].)

As it is a relatively recent development, research on mechanized reasoning with separation logic is just beginning. The Smallfoot static assertion checker discovers proofs of lightweight shape specifications done using the logic [5]. And there are developing applications using interactive proof tools [32, 49] and abstract interpretation [17, 12, 7, 20, 21].

4 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.

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

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 trivial 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.

Hoare had provided a beautiful rule for disjoint concurrency [23]

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

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

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

In this view the two parallel calls to `ms` are judged to be altering the *same* variable, a. So, the rule does not apply.

Cliff Jones has proposed a powerful approach to reasoning about concurrency, in his rely-guarantee formalism [27] (see also, [34]). 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. 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.

Because it is intuitively about separation, this example can be treated very easily in a concurrent extension of separation logic [39, 11]. The crucial part of the proof is the following proof figure for the parallel composition.

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

The use of the $*$ connective in $\mathit{array}(a,i,m) * \ \mathit{array}(a,m{+}1,j)\}$ 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 independently.

This rule is, of course, a descendent of Hoare's rule for disjoint concurrency. There are two reasons why we are able to treat this example where the original rule was not: (i) the assignment $a[i] := e$ is not viewed by separation logic as an assignment to a, but rather to a single cell; (ii) $*$ can be used to describe partitioning of an array that is dynamic, depending on the program state.

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 [28]. 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.

I do not claim that concurrent separation logic in its current state is the answer. It is good at specifying independence, but struggles with tightly-coupled, interfering processes. In contrast, rely-guarantee is good at describing interference, but is not well oriented to specifications of independent processes. Recently, there have been attempts to marry the advantages of concurrent separation logic and rely/guarantee [41, 18]; these are perhaps further steps on the way to modular reasoning about (shared variable) concurrent processes.

5 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 [13, 3], stemming Hogg's colorful declaration "that objects provide encapsulation is the big lie of object-oriented programming [25]". Further difficulties, beyond confinement, are caused by low-level features such as address arithmetic and storage deallocation.

A good initial challenge which illustrated many issues 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. 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.

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, where our implementation uses a free list in the usual way. A first attempt at specification might be something like

$\{\texttt{allocated}(y) \wedge y.f = m \wedge \neg\texttt{allocated}(z)\}$
$\texttt{alloc}(x)$

$$\big\{\texttt{allocated}(y) \wedge y.f = m \wedge \texttt{allocated}(x) \wedge y \neq x \\ \wedge (z \neq x \Rightarrow \neg\texttt{allocated}(z))\big\}$$

$$\big\{\texttt{allocated}(y) \wedge y.f = m \wedge \texttt{allocated}(x) \wedge y \neq x \wedge \neg\texttt{allocated}(z)\big\} \\ \texttt{free}(x) \\ \big\{\texttt{allocated}(y) \wedge y.f = m \wedge \neg\texttt{allocated}(x) \wedge y \neq x \wedge \neg\texttt{allocated}(z)\big\}$$

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 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.

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.

This sort of example has been successfully treated in separation logic [38]. The details are much more involved than the earlier examples, and I will not give the proof here. The basic idea is that the $*$ connective allows the separation of the state owned by a client and the state owned by the manager (the free list). Crucially, since $*$ is a logical connective, the partition it describes can change over time: in a sense, the logic tracks the right to dereference a cell transfers back and forth between client and module.

.

6 The Boogie Methodology and Relatives

Many of the issues touched on in this paper have also been approached in work on the "Boogie methodology" [30, 36, 4], and also in its precursors (see [29]). The basic idea of Boogie it to use certain auxiliary variables, such as ones to describe "ownership" of heap cells, to structure specifications and to constrain who can access what and when. Boogie builds on type systems for ownership [13, 16], but uses assertions rather than types. Ownership gives a way to express a form of separation, and frame axioms are avoided by using general invariants which relate the states of auxiliary variables and the program state. The auxiliary variables allow fine control over when certain assertions, such as object invariants, must hold; this has allowed a novel approach to the old and vexing problem of object invariants for re-entrant modules (which allow implicit or explicit recursion).

I discussed an example similar to the first one in this paper with Peter Müller (we discussed copytree rather than disposetree). The early versions of Boogie could not handle that example due to inadequate framing properties, but a later version [31] could. Conversely, the earliest approach to information hiding using separation logic [38] could not handle re-entrant modules, but the later approach of [40] can. As shown in [8], the approach pioneered in [40] can be understood as using quantified predicates in a way that is analogous to the use of polymorphic typing to account for hiding of internal representation types [45, 35]. On the other hand, Boogie has "pack" and "unpack" primitives which are intuitively similar to the corresponding primitives for existential types.

I just wanted to mention Boogie, to acknowledge (and point the reader to) the advances it and its relatives have made on difficult problems concerning modular reasoning about object-oriented programs. The exact relationship between Boogie and separation logic is not clear; there are similarities in intuition, but many differences in technique. The reader is referred to [29] for more information on this line of work, including work on ESC/Java and JML that I have not mentioned here.

7 Conclusion

In this paper I wanted to show some difficulties as regards modularity that traditional program logic has on even simple examples, and how it is not impossible to do much better, at least on those examples. In doing this I purposely started from programs rather than specifications; it is a good way to show where formalisms have difficulties. There are many other, more difficult, programs that can serve as challenging test cases.

Although I enjoy starting from programs, I would also love to be able to arrive at the kinds of program I considered by refinement, starting from a simple specification. I just don't know how to do so. The refinement formalisms that I am aware of (VDM, B, etc) are based on a static form of modularity, where the state that a program component can change is listed in a fixed collection of variables, and the frame properties used are with respect to modifies clauses for these variables. This fixed modularity does not deal well when the partitions between the state used by program components is more dynamic, as is the case in parallel mergesort, in the resource manager example, and typically in systems programs. Of course, this last point should be taken as a challenge. It seems inconceivable that the modularity issues that separation logic and Boogie attempt to address should not show up as well on a design level. Furthermore, there are all sorts of dynamic, interconnected structures other than the program heap, those obtained from networks and message passing being prime examples. One might hope for a design formalism (say, an analogue of B or Z) that goes beyond static modularity, and that has the specific heap modularity of separation logic or Boogie as an instance.

Similar remarks apply to my focus on imperative programs, and shared-variable concurrency. A good problem would be to obtain a reasoning formalism for, say, the pi-calculus or for socket programs that displays the same sort of modularity in its account of channel usage as separation logic or Boogie does for the heap.

All of the examples in this paper have concerned safety properties. Recently, there has been progress on automatic proofs of liveness properties properties of software, using novel applications of abstract interpretation [42, 15, 6]. The problem of modular, or local, specifications and verifications of liveness properties of concurrent processes looms as an extremely difficult one; see [48, 1] for important work in this direction.

Finally, one might question whether modular reasoning methods for software are in general even possible. In temporal logic there have been negative technical results [43], and we should be on the lookout for others. But, there has been considerable progress on modular reasoning about programs and this author, for one, plans to continue searching.

References

1. M. Amadi and L. Lamport. Composing specifications. In *ACM TOPLAS 15(1)*, pp73-132, 1993.
2. T. Ball, B. Cook, V. Levin, and S.K. Rajamani. SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft. In *4th IFM*, LNCS 2999, pp1–20, 2004.
3. A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *J.ACM*, to appear, 2005.
4. M. Barnett, R. DeLine, M. Fahndrich, K.R.M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.
5. J. Berdine, C. Calcagno, and P.W. O'Hearn. Smallfoot: Automatic modular assertion checking with separation logic. In *4th FMCO*, pp115-137, 2006.
6. J. Berdine, A. Chawdhary, B. Cook, D. Distefano, and P. W. O'Hearn. Variance analyses from invariance analyses. *34th POPL*, pp211-224., 2007.
7. J. Berdine, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. *19th CAV*, 2007.

8. B. Biering, L. Birkedal, and N. Torp-Smith. BI-hyperdoctrines, higher-order separation logic, and abstraction. *ACM TOPLAS*, to appear, 2007.
9. A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Transactions of Software Engineering*, 21:809–838, 1995.
10. R. Bornat, C. Calcagno, and H. Yang. Variables as resources in separation logic. In *19th MFPS*, 2005.
11. S.D. Brookes. A semantics for concurrent separation logic. Festschrift for John C. Reynolds's 70th Birthday. *Theoretical Computer Science 375(1-3), pp227-270*. Prelim version appeared in CONCUR'04, LNCS 3170., 2007.
12. C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Beyond reachability: Shape abstraction in the presence of pointer arithmetic. *13th SAS*, LNCS 4134, pp182-203, 2006.
13. 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.
14. D. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. CASSIS, pp108-128, 2004.
15. Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *13th PLDI*, 2006.
16. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 2005. To appear.
17. D. Distefano, P. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *12th TACAS*, pages 287–302, 2006.
18. X. Feng, R. Ferreira, and Z. Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *16th ESOP*, 2007.
19. C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended static checking for Java. In *9th PLDI*, 2002.
20. A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. To appear in *PLDI 2007*.
21. B. Guo, N. Vachharajani, and D. August. Shape analysis with inductive recursion synthesis. To appear in *PLDI 2007*.
22. 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.
23. C. A. R. Hoare. Towards a theory of parallel programming. In Hoare and Perrot, editors, *Operating Systems Techniques*. Academic Press, 1972.
24. C.A.R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
25. J. Hogg. Islands: aliasing protection in object-oriented languages. 6th OOPSLA, 1991.
26. S. Isthiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 36–49, London, January 2001.
27. C. B. Jones. Specification and design of (parallel) programs. *IFIP Conference*, 1983.
28. C. B. Jones. Wanted: A compositional approach to concurrency. In A. McIver and C. Morgan, editors, *Programming Methodology*, pages 1–15, 2003. Springer-Verlag.
29. G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 2007. To appear.

30. K.R.M. Leino and P. Müller. Object invariants in dynamic contexts. In *18th ECOOP*, LNCS 3086, pp491-516, 2004.
31. K.R.M. Leino and P. Müller. A verification methodology for model fields. In *15th ESOP*, LNCS 3924, pp115-130, 2006.
32. N. Marti, R. Affeldt, and A. Yonezawa. Verification of the heap manager of an operating system using separation logic. *Proceedings of the 3rd SPACE Workshop*, Charleston, 2006.
33. J. McCarthy and P. Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Machine Intelligence*, 4:463–502, 1969.
34. J. Misra and K.M. Chandy. Proofs of networks of processes. *IEEE Trans. Software Eng.*, 7(4):417–426, 1981.
35. J. C. Mitchell and G. D. Plotkin. Abstract types have existential types. *ACM Trans. Programming Languages and Systems*, 10(3):470–502, 1988.
36. D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. In *19th LICS*, pages 313–323, 2004.
37. 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.
38. P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *31st POPL*, pages 268–280, 2004.
39. P.W. O'Hearn. Resources, concurrency and local reasoning. Festschrift for John C. Reynolds's 70th Birthday. *Theoretical Computer Science 375(1-3), pp271-307*. Prelim version appeared in CONCUR'04, pp49–67, LNCS 3170., 2007.
40. M. Parkinson and G. Bierman. Separation logic and abstraction. Proceedings of POPL, 2005.
41. M. Parkinson and V. Vafaedis. A marriage of rely-guarantee and separation logic. *18th CONCUR*, to appear, 2007.
42. Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *19th LICS*, 2004.
43. A. Rabinovich. On compositionality and its limitations. In *ACM TOCL 8(1)*, pp73-132, 2007.
44. 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.
45. J. C. Reynolds. Types, abstraction and parametric polymorphism. Proceedings of IFIP, 1983.
46. J. C. Reynolds. Intuitionistic reasoning about shared mutable data structure. In Jim Davies, Bill Roscoe, and Jim Woodcock, editors, *Millennial Perspectives in Computer Science*, pages 303–321, Houndsmill, Hampshire, 2000. Palgrave.
47. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th LICS*, pp 55-74, 2002.
48. E.W. Stark. A proof technique for rely/guarantee properties. In *FSTTCS*, LNCS 206, pp369-391, 1985.
49. H. Tuch, G. Klein, and M. Norrish. Types, bytes, and separation logic. In *34th POPL*, 2007.
50. H. Yang. *Local Reasoning for Stateful Programs.* Ph.D. thesis, University of Illinois, Urbana-Champaign, 2001.
51. H. Yang and P. O'Hearn. A semantic basis for local reasoning. In *Foundations of Software Science and Computation Structures*, Springer LNCS 2303., 2002.