Modular reasoning in object-oriented programming

David A. Naumann[⋆]

Department of Computer Science, Stevens Institute of Technology
naumann@cs.stevens.edu

Abstract. Difficulties in reasoning about functional correctness and relational properties of object-oriented programs are reviewed. An approach using auxiliary state is briefly described, with emphasis on the author's work. Some near term challenges are sketched.

Formal verification depends on scientific theories of programming, which answer questions such as these: What are good models of computational behavior? What behavioral properties of components are needed for modular reasoning about a composed system? How can such properties be specified and a component be verified, or even derived from its specification? How can a program and justification of its correctness be revised in accord with small revision of its specification? Such questions have well developed answers that are adequate for small programs under strong simplifying assumptions. But many useful programs are quite large and built from complicated components that violate simplifying assumptions.

The longstanding challenge of compositional reasoning remains substantially unsolved. Object-oriented programs pose several challenges that are the focus of my recent research, in which auxiliary state is being used to specify encapsulation boundaries and disciplined interdependence. Section 2, explains the approach, accomplishments, and challenges in terms of invariants for shared mutable objects. Section 3 addresses relational properties including data refinement and secure information flow. This line of research has been carried out for Java-like programming languages; I argue in Section 1 for the importance of such languages. Some additional challenges pertinent to object-oriented programming, but not tied to the main theme, are discussed in Section 4. A detailed tutorial on the state-based approach to encapsulation advocated here appears elsewhere [33].

Several near-term challenges (1–5 years) are presented here in the setting of sequential object-oriented programs. Because the approach taken here is based on the use of assertions, it is also quite relevant to verification of concurrent object-oriented programs and low level imperative code.

1 Why Java-like language?

In order to develop theory for modular reasoning about large programs, we need a corpus of large programs and automated support for experiments. Since I would like to do

[⋆] Partially supported by the National Science Foundation under grants CCR-0208984 and CCF-0429894 and by Microsoft Research.

science that contributes to human good through improved engineering, the primary objects of study should be representative examples of large programs that are significantly deployed and used. This means confronting programs written in notations like C, Java, and C#—though not necessarily handling all of their features without restriction. Aside from obvious pragmatic reasons for interest in Java-like languages, there are technical reasons why such a language is a good point in the language design space.

– The language is sufficiently rich to express higher order design patterns which are needed for well structured programs and used in common practice.
– Despite the preceding item, the language is essentially "defunctionalized" [42, 4] owing to the binding of methods to classes rather than to instances. Thus relatively simple semantic models are adequate, at least for large fragments of the language. For example, my work discussed in Sections 2 and 3 has been done using a straightforward Scott-Strachey denotational semantics, for a fragment of Java including recursive types, inheritance, mutable objects, and other features without restriction; this model has been encoded in PVS [34]. Nipkow's group and others have obtained strong results using straightforward operational models [26].
– The module system (packages, generic classes, public/private/protected visibility) embodies most of what current theory offers for scope-based encapsulation.
– The Java type system is name-based; named types provide a convenient hook on which to hang specifications and encapsulation boundaries. In particular, it helps deal with inheritance, which is widely used if problemmatic.
– Pointer arithmetic is absent. Parameter passing is by value and identifiers cannot alias. Method declarations are not nested, avoiding the semantic complexity of reference to variables in enclosing scopes other than global scope.[1]

These features are not without cost. Java programs make much use of global variables ("statics")—global in that they are in outermost scopes; this is mitigated in that the scope of visibility may be a single class or package. Reflection, at least in full generality, is a feature I see as a very difficult and long-term challenge for verification. This is exacerbated in that reflection, like threads and permission-based access control, appears in the form of special libraries rather than being distinguished with separate syntax.

Perhaps the highest cost is the ubiquity of aliasing in the sense of shared references to mutable objects in the heap.

2 Heap encapsulation using auxiliary state

For modular reasoning in object-oriented programming there are several challenges.

1. Non-hierarchical control flow due to callbacks leads, even in sequential programs, to interference like that in concurrent programs.
2. The conventional notion of layered abstraction is also subverted by non-hierarchical control flow due to inheritance and method overriding.

[1] Compare the complexity of Idealized Algol models [44] with Modula-3 and Oberon, where non-local references are restricted for those procedures that are passed as arguments or stored in variables [32].

3. Design patterns that are essentially higher order are often used, but unlike in functional programming the encapsulation aspects are not explicit in the program text, owing to data representation based on shared heap objects.
4. Functional aspects of such patterns are also not specified formally, for lack of good models (compare "map" in functional programming with the "Visitor" pattern).

The second challenge is addressed by the notion of behavioral subtyping which is well understood [29, 20] except that the extant theories do not fully deal with the first and third challenges.

For the fourth challenge, which we discuss in Section 4, one might argue that at best we should aim for verifying simple safety properties. Indeed, in his VSTTE talk Bart Jacobs said that full functional verification of nontrivial Java programs is impractical. But for realistically complex systems, attempts to verify simple safety properties lead to the need for more general properties, especially object invariants.

For the first and third challenges, progress is being made using auxiliary state to express encapsulation using assertions. That is the topic of this section, which focuses on object invariants. More extensive discussions and citations on these topics can be found in Müller's VSTTE paper [31] and my survey paper [33].

Non-hierarchical control flow. As an example of the first challenge, consider a sensor playing the role of Subject in the Subject/Observer pattern [22]. The sensor maintains a set of registered Views: when the sensor value reaches the threshhold $v.\mathit{thresh}$ of a given view v, the sensor invokes method $v.\mathit{notify}$ and removes v from the set. This description is in terms of a set, part of the abstraction offered by the Subject; the implementation might store views in an array ordered by *thresh* values. The pattern cannot be seen simply as a client using an abstraction, because *notify* is what is known as an *upcall* to the client. The difficulty is that *v.notify* may make a *reentrant callback* to the sensor. Some callbacks are quite sensible, e.g., the view could query the sensor value. But trouble is likely if *v.notify* invokes a method to enumerate the current set of views. While notifications are under way, the array may be in an inconsistent state—is v in the set? in the array?—yet the enumeration method may assume as precondition the sensor's invariant. Non-hierarchical control flow renders naive reasoning about object invariants unsound.

The problem is similar to interference in shared-variable concurrency, for which there are several established and well understood solutions. For the reentrant callback problem, which already occurs in sequential code, the situation is less settled, although the probem is a frequent cause of insidious bugs. Various solutions have been proposed:

- Establish caller's invariant before *every* method call. But this is impractical in many cases: most calls do not result in reentrant callbacks and good use of abstraction in design leads to many calls to substructures while a super-structure's invariant is temporarily violated.
- Use concurrency locks. But this leads to deadlocks in the sequential case.
- Use temporal specification of allowed calling sequences. This can be heavy handed and violates abstraction by making method calls visible. Moreover, verification of such properties requires the whole program in general.

A more promising approach begins by making the invariant an explicit precondition on those methods that assume it, like the enumerator in the example. This precondition cannot be established by client v attempting a reentrant callback, unless in fact the sensor restores its invariant before invoking *v.notify*.

An object invariant $\mathcal{I}$ ought not appear in the precondition of a public method, as that could expose the internal representation. Various techniques have been proposed to hide information, e.g., treating $\mathcal{I}$ in a precondition as an opaque predicate [14, 15], a typestate [19], a call to a pure method, or a model field [30, 25].

We advocate the approach of Leino *et al* [8], known as the *Boogie methodology* or the inv/own discipline. We give a simplified account sufficient for discussion. The discipline uses a *ghost* (auxiliary) field[2] inv of type boolean which represents whether the invariant of o is in force, just as a programmer might do using an ordinary field. There are several associated proof obligations; together they embody a discipline that ensures the following is a *program invariant*, i.e., it holds in all reachable states:

$$(\forall\, o \mid o.inv \Rightarrow \mathcal{I}(o)\,) \tag{1}$$

Informally: for each allocated object o, the object's invariant holds if $o.inv = true$. Thus within the body of a method with precondition inv, one can exploit the invariant $\mathcal{I}$ while exposing to clients not the predicate $\mathcal{I}$ but only the boolean field inv.

Heap encapsulation. Besides its own fields, an object may depend on some objects that serve as its internal representation. This can be represented using another auxiliary field by which an object points to its direct *owner*, if any. An object's invariant is allowed to depend only on objects it transitively owns. An associated program invariant is that $o.inv$ implies $p.inv$ for every object p owned by o. If an object is in a consistent state then so are its representation objects. This invariant is maintained owing to a proof obligation: update of a field of an object p has as precondition that $p.inv = false$. So, if an object p is susceptible to update then not only may $\mathcal{I}(p)$ be temporarily violated but also if p is part of the representation of some object o then also $o.inv = false$ and $\mathcal{I}(o)$ may be temporarily violated.

Ownership imposes a forest structure on the heap, separating encapsulated data from clients. Ownership types [18, 2] embody this idea and an account of the resulting encapsulation has been given in terms of the theory of representation independence [5]. But it has proved difficult to find an ownership type system that admits common design patterns and also enforces encapsulation sufficiently strong for modular reasoning about object invariants. In particular, many examples call for the transfer of ownership (e.g., in resource management) and this does not sit well with types.

An alternative to types is separation logic [45, 39]. In separation logic, owning an object p has been equated with having a precondition dependent on p. A modest challenge is how to scale the logic up to classes (instantiable abstractions) instead of single-instance modules. A bigger challenge is how to cope with the fact that in object-oriented

[2] For our purposes, a *model field* is an auxiliary field, the value of which is defined as a function of other state, whereas the value of a *ghost field* must be updated by explicit auxiliary assignments.

languages, the object is the unit of addressability but some fields are inherited and others (to be added in subclasses) are not known to the modular reasoner. Parkinson and Bierman [41, 14] have taken initial steps and their treatment of encapsulation has been given an acount in terms of higher order separation logic [13, 15]. By contrast with separation logic, the approach described here is compatibility with standard logics and specification notions, which can leverage existing tools and programmer expertise.

One advantage of encoding ownership with a ghost field is that transfer is straightforward; the field is mutable. In combination with the invariant-tracking field inv, the discipline [8, 28] expresses very directly the flow of control in and out of hierarchical encapsulation boundaries even as those boundaries are mutated.

The most exciting advantage of the approch is that it generalizes to more elaborate patterns. Ownership is concerned with a single object and its representation. Already the pattern of iterators is problemmatic, in that an iterator needs access to the representation objects of its associated collection but a collection is not owned by its iterators. There are many situations where several publically-accessible objects cooperate to provide an abstraction, so their individual invariants need to depend on non-owned objects. Just as the $owner$ field records a dependence that can be taken into account in reasoning, one can use a ghost field to record the dependence between peer objects.

This idea has been developed in the simple case of one object's invariant depending on another: the "friendship" discipline [37, 10] imposes modular obligations on both dependee and dependant, so (1) is maintained even when an invariant $\mathcal{I}$ depends on non-owned objects. A field $deps$ is used so that $p.deps$ is a set of object references that includes all o that could have an invariant currently dependent on p that is not licensed by ownership.

The friendship discipline has been successfully applied to several design patterns including iterators [38] and Subject/View [10], but it does not seem likely that there is a single such discipline sufficiently general to handle every situation. I believe that by using auxiliary state to record encapsulation boundaries for heap structure, we can formalize a number of generally applicable *specification patterns*. Interactive theorem proving or just pencil and paper can be used to show that the associated global invariant is a consequent of the pattern's stipulated annotation discipline. Automated first-order provers may then be used to discharge the assertions in particular instances of the pattern, treating program invariants like (1) as axiom schemes.

For patterns that can be specified using just ownership, the Spec# system implements the Boogie methodology using a first-order prover as discussed in the VSTTE paper of Barnett *et al* [9]. Ownership can also be encoded in the JML specification language which is being used in a number of verification systems, as discussed in the VSTTE paper of Leavens and Clifton [27]. There is impressive agreement about syntax but the semantics is neither formalized nor entirely consistent between projects. Within a 5-year time frame it should be possible to provide a foundational logic for JML, encompassing encapsulation (via scope and via auxiliary state), reentrancy, and behavioral subtyping. This would serve to integrate and assess, in particular helping to ensure that the axiomatic semantics embodied in some tools is sound with respect to an (idealized) operational semantics. Concurrency specification is less developed but a sound treatment using strong atomicity assumptions should be within reach [46].

3 Relational properties

By relational property I mean notions like simulation, where a pair of programs preserve some relation. The most important relational property is preservation of a simulation between implementations of an abstract data type —yielding modular proof of program equivalence or data refinement. Another is noninterference in the sense of secure information flow and dependency analyis [1, 47], which in turn can be used to justify use of impure method calls in specifications [36]. Sampaio *et al* developed a refinement calculus for a subset of Java [16] and implemented a tool that applies general refactoring transformations that are validated on the basis of a theory of data refinement [17].

The latter theory is only sound in the absence of heap sharing. Anindya Banerjee and I have adapted the inv/own discipline to support representation independence [7], i.e., soundness of simulations for proof of program equivalence with heap sharing. In five years it should be possible to integrate these theories to encompass refinement and shared heap objects, allowing as units of encapsulations multiple classes and more importantly small configurations of cooperating objects (e.g., a set and its enumerators). An associated milestone would be a refactoring tool, say for Java's Eclipse developments environment, that applies semantically validated transformations.

Benton [12] and Yang [50] propose relational Hoare logics in which the basic correctness condition takes the form

$$\{R\} \genfrac{}{}{0pt}{}{S}{S'} \{P\} \tag{2}$$

where S and S' are commands, R and Q are relations. The meaning is that running S in parallel with S' on a pair of R-related states yields P-related final states. Amtoft *et al* [3] axiomatize a relational Hoare logic for the special case of noninterference, using special syntax in assertions to specify relations between heap regions for precise reasoning about sharing. These logics merit further development and machine support.

Relational properties can be proved using extant tools for ordinary correctness. The idea is make two copies of the state space and somehow compose the related programs together in such a way that relations are predicates and the relational property is reduced to a Hoare triple [21, 43, 23, 49, 11]. Essentially, (2) is reduced to $\{R\}S; S'\{P\}$. This has been called the *auxiliary variable* technique, Reynolds' method, and *pair composition* among others. Making two copies of a state given by explicit variables is easy; just make a renamed copy of the variables. For the heap, the relevant relations typically involve a partial bijection on addresses [5, 6] and this needs to be encoded in a single heap. I have recently used ghost variables to encode heap-based relations and thereby adapt the pair composition technique to Java programs [35]. This technique can leverage existing verification tools; experiments using ESC/Java2 and Spec# have been promising.

There is a difficulty with this technique. It already appears in the special case of noninterference, where in (2) S' is a renamed copy of S. Terauchi and Aiken [49] experimented with this case, called *self composition*, and found that even when R, P are very weak, a strong assertion is needed at the semicolon in $\{R\}S; S'\{P\}$. They also show how type-based analysis that conservatively approximates noninterference can facilitate automation of self composition technique, by justifying transformation of $S; S'$ into a better, interleaved form. Similar transformations are especially useful for the ghost assignments needed to use the technique with the heap.[3] As I point out in [35], such transformations are exactly the kind Benton aims to account for [12].

It is debatable whether a specialized relational property like noninterference merits much attention in the Program Verifier project. But the importance of data refinement seems clear. While the pair composition technique is attractive in that it can encode relations in an ordinary specification logic like JML, such logics are not so expressive in terms of high level mathematical abstractions. For my small experiments [35], I used ad hoc specifications but in general what is needed is to express the pair encoding of heaps using something like friendship invariants.

Within five years we should be able to develop a theory of relational Hoare logic encompassing the heap and inheritance that is complete and which supports the noninterference transformations as derived laws. We should also be able to extend the theory and implementations of verifiers like ESC/Java2 and Spec# to support a sufficiently expressive specification language for pair composition. This would avoid the need to build tools specific to relational Hoare logic.

4 Design patterns, higher order logic, and refinement

Regions of the heap, such as a small configuration of objects and their transitively owned representations, are often the focus of reasoning. Why are heap regions second class? In separation logic, quantification over predicates is needed for interesting specifications, in part because patterns of heap structure are expressed using separation at the level of predicates. Moreover, sound reasoning about invariants depends on them being supported by a definite region of the heap [40]. In the inv/own discipline, relevant sets of objects are determined by $owner$ paths. In neither case are regions directly manipulated. Why not expressions describing regions? Reynolds [45] mentions ghost variables ranging over heaps, but this is not available in extant work on higher order separation logic [15, 13].

Kassios introduced something akin to expressions for regions [25]. He uses model fields to express encapsulation in a way somewhat different from the Boogie approach. Whereas the latter protects $\mathcal{I}(o)$ by restricting it to depend on objects p that record the dependence in an auxiliary field of p (i.e., $p.own$ or $p.deps$), Kassios uses a field of o to hold the refs to all objects on which $\mathcal{I}(o)$ currently depends. The fact that this field conservatively approximates the current footprint of $\mathcal{I}(o)$ can itself be expressed in assertions.[4] Kassios' methodology is quite flexible, in a way reminiscent of separation logic, and it elegantly handles some of the leading examples for the Boogie/friendship discipline. But the development is at an early stage.

In five years it should be possible to specify and verify programs such as application level resource managers by directly describing the heap regions on which they act —

[3] A suitable type-based analysis for Java-like programs was developed in [6].

[4] The construct "$f\ frames\ E$" says that the heap objects on which expression E depends are contained in object set f. This is a second order condition, but there appear to be adequate first order laws for reasoning with this as an uninterpreted predicate.

thus making transparent their frame properties. Better still, comparative case studies would serve to assess the alernative approaches we have mentioned.

Region notation would be especially useful for describing configurations of objects in design patterns, now expressed informally with various diagrams. I am aware of no convincing functional specifications for basic design patterns such as Visitor or Observer. Are there useful first-order specifications? Higher order? Absent a general functional specification, how can an instance of the pattern be specified in order to verify "structural integrity" of a system [24] and even functional correctness of the particular instance?

In five years it should at least be possible to verify absence of runtime errors in a 10Kloc Java application making use of inheritance and design patterns such as these.

An interesting aspect of the popularity of design patterns is that software engineers are increasingly familiar with the distinction between abstraction and modular structure in design versus in coding. Java, for example, offers classes and packages but no specialized construct for the visitor pattern or for the iterator pattern. Furthermore, "model driven development" emphasizes the construction of multiple linked artifacts, where again some high level structure need not be manifest in the lower level artifacts such as source code. This trend is hardly surprising to formal methods researchers and indeed it was emphasized long ago by Parnas. It offers some hope in moving away from rigid attachment to feature-rich monolithic languages (see also Abrial's VSTTE paper).

Since object-oriented design patterns show how to embody, in a conventional language, abstractions that are not directly expressed, one can hope for formal specification of a pattern as a refinement. This could provide an alternative to annotations as means to move software engineers towards writing formal specifications.

References

[1] M. Abadi, A. Banerjee, N. Heintze, and J. G. Riecke. A core calculus of dependency. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, 1999.

[2] J. Aldrich and C. Chambers. Ownership domains: Separating aliasing policy from mechanism. In *European Conference on Object-Oriented Programming (ECOOP)*, 1–25, 2004.

[3] T. Amtoft, S. Bandhakavi, and A. Banerjee. A logic for information flow in object-oriented programs. In *POPL*, 2006. Extended version available as KSU CIS-TR-2005-1.

[4] A. Banerjee, N. Heintze, and J. G. Riecke. Design and correctness of program transformations based on control-flow analysis. In *Intl. Symp. on Theoretical Aspects of Computer Software*, 420–447, Oct. 2001. LNCS 2215.

[5] A. Banerjee and D. A. Naumann. Ownership confinement ensures representation independence for object-oriented programs. *Journal of the ACM*, 52(6):894–960, Nov. 2005.

[6] A. Banerjee and D. A. Naumann. Stack-based access control for secure information flow. *Journal of Functional Programming*, 15(2):131–177, 2005.

[7] A. Banerjee and D. A. Naumann. State based ownership, reentrance, and encapsulation. In *European Conference on Object-Oriented Programming (ECOOP)*, 387–411, 2005.

[8] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004. Special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs.

[9] M. Barnett, R. DeLine, B. Jacobs, M. Fhndrich, K. R. M. Leino, W. Schulte, and H. Venter. The Spec# programming system: Challenges and directions. In B. Meyer and J. C. P. Woodcock, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2005.

[10] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In D. Kozen and C. Shankland, editors, *Mathematics of Program Construction*, volume 3125 of *LNCS*, 54–84, 2004.

[11] G. Barthe, P. R. D'Argenio, and T. Rezk. Secure information flow by self-composition. In *Proceedings of the 17th IEEE Computer Security Foundations Workshop (CSFW'04)*, 100–114, 2004.

[12] N. Benton. Simple relational correctness proofs for static analyses and program transformations. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, 14–25, 2004.

[13] B. Biering, L. Birkedal, and N. Torp-Smith. BI hyperdoctrines and higher-order separation logic. In *European Symposium on Programming (ESOP)*, volume 3444 of *LNCS*, 233–247, 2005.

[14] G. Bierman and M. Parkinson. Separation logic and abstraction. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, 247–258, 2005.

[15] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *IEEE Symp. on Logic in Computer Science (LICS)*, 260–269, 2005.

[16] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic reasoning for object-oriented programming. *Sci. Comput. Programming*, 52(1-3):53–100, 2004.

[17] A. L. C. Cavalcanti and D. A. Naumann. Forward simulation for data refinement of classes. In L. Eriksson and P. A. Lindsay, editors, *Formal Methods Europe*, volume 2391 of *LNCS*, 471–490, 2002.

[18] D. Clarke. Object ownership and containment. Dissertation, Computer Science and Engineering, University of New South Wales, Australia, 2001.

[19] R. DeLine and M. Fähndrich. The Fugue protocol checker: Is your software baroque? Available from http://research.microsoft.com/˜maf/papers.html, 2003.

[20] K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany*, 258–267. IEEE Computer Society Press, Mar. 1996.

[21] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[22] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[23] D. Gries. Data refinement and the tranform. In M. Broy, editor, *Program Design Calculi*. Springer, 1993. International Summer School at Marktoberdorf.

[24] T. Hoare and J. Misra. Verified software: Theories, tools, experiments. In B. Meyer and J. C. P. Woodcock, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2005.

[25] I. T. Kassios. Dynamic framing: Support for framing, dependencies and sharing without restriction. In J. Misra, T. Nipkow, and E. Sekerinski, eds., *Formal Methods*, volume 4085 of *LNCS*, 268–283, 2006.

[26] G. Klein and T. Nipkow. A machine-checked model for a Java-like language, virtual machine and compiler. *ACM Trans. Prog. Lang. Syst.*, 2006.

[27] G. T. Leavens and C. Clifton. Lessons from the JML project. In B. Meyer and J. C. P. Woodcock, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2005.

[28] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming (ECOOP)*, 491–516, 2004.

[29] B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. Prog. Lang. Syst.*, 16(6), 1994.

[30] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *LNCS*. Springer-Verlag, 2002.

[31] P. Müller. Reasoning about object structures using ownership. In B. Meyer and J. C. P. Woodcock, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2005.

[32] D. A. Naumann. Predicate transformer semantics of a higher order imperative language with record subtyping. *Sci. Comput. Programming*, 41(1):1–51, 2001.

[33] D. A. Naumann. Assertion-based encapsulation, object invariants and simulations. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *Post-proceedings, Formal Methods for Components and Objects (FMCO 2004)*, volume 3657 of *LNCS*, 251–273, 2005.

[34] D. A. Naumann. Verifying a secure information flow analyzer. In J. Hurd and T. Melham, editors, *18th International Conference on Theorem Proving in Higher Order Logics TPHOLS*, volume 3603 of *Lecture Notes in Computer Science*, 211–226. Springer, 2005.

[35] D. A. Naumann. From coupling relations to mated invariants for secure information flow. In *European Symposium on Research in Computer Security (ESORICS)*, volume 4189 of *LNCS*, 279–296, 2006.

[36] D. A. Naumann. Observational purity and encapsulation. *Theoretical Comput. Sci.*, 2006. To appear.

[37] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In *IEEE Symp. on Logic in Computer Science (LICS)*, 313–323, 2004.

[38] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state. *Theoretical Comput. Sci.*, (365) 143–168, 2006. Extended version of [37].

[39] P. O'Hearn. Scalable specification and reasoning: Technical challenges for program logic. In B. Meyer and J. C. P. Woodcock, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2005.

[40] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *ACM Symp. on Princ. of Program. Lang. (POPL)*, 268–280, 2004.

[41] M. J. Parkinson. Local reasoning for Java. Technical Report 654, University of Cambridge Computer Laboratory, Nov. 2005. Dissertation.

[42] J. C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of the ACM Annual Conference*, volume 2, 717–740, New York, 1972. ACM.

[43] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.

[44] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*. North-Holland, 1981.

[45] J. C. Reynolds. An overview of separation logic. In B. Meyer and J. C. P. Woodcock, editors, *Verified Software: Theories, Tools, and Experiments (VSTTE)*, 2005.

[46] E. Rodríguez, M. Dwyer, C. Flanagan, J. Hatcliff, G. T. Leavens, and F. Robby. Extending JML for modular specification and verification of multi-threaded programs. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.

[47] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[48] Q. Sun, A. Banerjee, and D. A. Naumann. Modular and constraint-based information flow inference for an object-oriented language. In R. Giacobazzi, editor, *Static Analysis Symposium (SAS)*, volume 3148 of *LNCS*, 84–99. Springer-Verlag, 2004.

[49] T. Terauchi and A. Aiken. Secure information flow as a safety problem. In *12th International Static Analysis Symposium (SAS)*, volume 3672 of *LNCS*, 352–367, 2005.

[50] H. Yang. Relational separation logic. *Theoretical Comput. Sci.*, 2004. To appear.