Modular reasoning in object-oriented programming

David A. Naumann*

Department of Computer Science, Stevens Institute of Technology

This paper responds to the solicitation of position papers for the IFIP Working Conference on Verified Software: Theories, Tools, Experiments (http://vstte.ethz.ch). I outline my recent research in verification, listing accomplishments and challenges. I do not articulate suitably grand challenges or milestones.

My primary interest is in scientific theories of programming: 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? 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. The main part of this paper, Section 2, explains the approach, accomplishments, and challenges.

My work has focused on Java-like programming languages; I argue in Section 1 for the importance of such languages. Section 4 lists some related projects in which I am involved.

1 Why Java-like language?

In order to develop theory for modular reasoning about large programs, one needs a corpus of large programs and automated support for experiments. Since I would like to do 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, there are several ways in which Java-like languages are 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" [26, 2] 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.[1]

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

[1] For example, my work discussed in Section 2 has been done using a machine-checked Scott-Strachey denotational semantics, for a fragment of Java including recursive types, inheritance,

- 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 (almost), avoiding the semantic complexity of reference to variables in enclosing scopes other than global scope.[2]

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.
3. Design patterns that are essentially higher order are often used, but unlike in functional programming the encapsulation aspects are not expressed, 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).

For the fourth challenge, one might argue that at best we should aim for verifying simple safety properties, but for realistically complex systems this quickly leads to the need for more general properties, especially for object invariants. The second and fourth items are left to Section 3 as open challenges. For the other two, progress is being made using auxiliary state as described in this section. A benefit of the approach is compatibility with standard logics and specification notions, so it leverages existing tools and programmer expertise.

As an example of the first challenge, consider a sensor playing the role of Subject in the Subject/Observer pattern. 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

mutable objects, and other features without restriction. Nipkow's group and others have obtained strong results using operational models.

[2] Compare the complexity of Idealized Algol models [28] with Modula-3 and Oberon, where non-local references are restricted for those procedures that are passed as arguments or stored in variables [20].

method v.*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 then 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. Trouble is likely if view 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.

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 flawed solutions have been proposed:

- Establish caller's invariant before *every* method call (impractical in many cases, and most calls do not result in reentrant callbacks).
- Use concurrency locks (leads to deadlocks in the sequential case).
- Use temporal specification of allowed calling sequences (heavy handed and violates abstraction by making method calls visible; verification requires whole program).

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 [10, 11], as a typestate [16], or a call to a pure method.

The most promising approach is that of Leino et al [7]. In a simplified account sufficient for discussion, they use a *ghost* (auxiliary) field inv of type boolean which represents whether the invariant of o is in force, just as a programmer might do using an ordinary field. Several associated proof obligations embody a discipline that ensures the following is a *Program invariant*, i.e., it holds in all reachable states:

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

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.

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.

Ownership imposes a forest structure on the heap, separating encapsulated data from clients. Ownership types [15, 1] embody this idea and the encapsulation has been assessed in terms of the standard theory of representation independence [3]. 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.

Exciting progress has been made using separation logic, where owning an object o has been equated with having a precondition dependent on o. 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 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.

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 [7, 19] 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 [24, 8] imposes modular obligations on both dependee and dependant, so (1) is maintained even when an invariant $\mathcal{I}$ depends on non-owned objects. This discipline has been successfully applied to several design patterns including iterators and Subject/View, 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 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.

3 Some challenges and milestones

The JML specification language is being used in a number of verification systems. There is impressive agreement about syntax but the semantics is neither formalized nor 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. Concurrency specification is less developed but a sound treatment using strong atomicity assumptions should be within reach [29].

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 [25]—why not *expressions* describing regions?

I am aware of no convincing functional specifications for basic design patterns such as the Visitor and Observer pattern. Are there useful first-order specifications? Higher order? Absent a general functional specification, how can an instance of the pattern be specified? In five years it should be possible to verify absence of runtime errors in a 10Kloc Java application making use of inheritance and design patterns such as these.

4 Related projects

The ownership/inv discipline is part of the Boogie methodology being implemented in the Spec# verifying compiler. My work has been done in ongoing collaboration with the Spec# team. We also worked out a theory and static analysis to soundly allow specifications to include calls to methods that have side effects that are not observable in the context of the specification [9, 23].

Anindya Banerjee and I have adapted the discipline to support representation independence [6], i.e., soundness of simulations for proof of program equivalence. (See also the tutorial [21].) Although such proofs are commonplace, I am not aware of language based verification tools that support relational properties. I recently showed how to use ghost fields to encode relational properties as ordinary verification conditions for suitable composition of a pair of programs [22], extending previous work [17, 27, 18] to encompass Java-like programs.

With NSF funding, Gary Leavens and I are working on semantics for core features of JML and aim to justify its rules on behavioral subtyping. We also hope to renew my collaboration with Augusto Sampaio and his colleagues at University of Pernambuco, Brazil, in which we developed a refinement calculus for a subset of Java [12, 13, 14]. They have implemented a refactoring tool on this basis.

Anindya Banerjee and I are also working on secure information flow analysis for Java, attempting to achieve more flexible policy by taking into account access control [5]. I used PVS to verify correctness of the analysis algorithm. With my student Qi Sun we developed an algorithm for modular inference of security levels [30]. Qi Sun is implementing a prototype and using the analysis for checking observational purity [9]. I'm working with Gilles Barthe and Tamara Rezk, INRIA Sophia-Antipolis, on security-type preserving compilation.

References

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

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

[3] A. Banerjee and D. A. Naumann. Ownership confi nement ensures representation independence for object-oriented programs. *Journal of the ACM*, 2002. Accepted, revision pending. Extended version of [4].

[4] A. Banerjee and D. A. Naumann. Representation independence, confi nement and access control. In *POPL*, 166–177, 2002.

[5] A. Banerjee and D. A. Naumann. Stack-based access control for secure information fbw. *Journal of Functional Programming*, 15(2):131–177, 2003.

[6] A. Banerjee and D. A. Naumann. State based ownership, reentrance, and encapsulation. In *European Conference on Object-Oriented Programming (ECOOP)*, 2005.
[7] 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.
[8] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In D. Kozen, editor, *Mathematics of Program Construction*, 54–84, 2004.
[9] M. Barnett, D. A. Naumann, W. Schulte, and Q. Sun. 99.44% pure: Useful abstractions in specifications. In *ECOOP workshop on Formal Techniques for Java-like Programs (FTfJP)*, 2004. Technical Report NIII-R0426, University of Nijmegen; journal version submitted.
[10] G. Bierman and M. Parkinson. Separation logic and abstraction. In *POPL*, 247–258, 2005.
[11] L. Birkedal, N. Torp-Smith, and H. Yang. Semantics of separation-logic typing and higher-order frame rules. In *LICS*, 260–269, 2005.
[12] P. Borba, A. Sampaio, A. Cavalcanti, and M. Cornélio. Algebraic reasoning for object-oriented programming. *Sci. Comput. Programming*, 52(1-3):53–100, 2004.
[13] P. H. M. Borba, A. C. A. Sampaio, and M. L. Cornélio. A refinement algebra for object-oriented programming. In L. Cardelli, editor, *European Conference on Object-oriented Programming (ECOOP)*, number 2743 in LNCS, 457–482, 2003.
[14] 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.
[15] D. Clarke. Object ownership and containment. Dissertation, Computer Science and Engineering, University of New South Wales, Australia, 2001.
[16] R. DeLine and M. Fähndrich. The Fugue protocol checker: Is your software baroque? Available from http://research.microsoft.com/~maf/papers.html, 2003.
[17] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.
[18] D. Gries. Data refinement and the tranform. In M. Broy, editor, *Program Design Calculi*. Springer, 1993. International Summer School at Marktoberdorf.
[19] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *European Conference on Object-Oriented Programming (ECOOP)*, 491–516, 2004.
[20] D. A. Naumann. Predicate transformer semantics of a higher order imperative language with record subtyping. *Sci. Comput. Programming*, 41(1):1–51, 2001.
[21] D. A. Naumann. Assertion-based encapsulation, object invariants and simulations. In *FMCO post-proceedings*, 2005.
[22] D. A. Naumann. From coupling relations to mated invariants for secure information flow and data abstraction. Submitted for publication, July 2005.
[23] D. A. Naumann. Observational purity and encapsulation. In *Fundamental Aspects of Software Engineering (FASE)*, 190–204, 2005.
[24] D. A. Naumann and M. Barnett. Towards imperative modules: Reasoning about invariants and sharing of mutable state (extended abstract). In *LICS*, 313–323, 2004.
[25] P. O'Hearn, H. Yang, and J. Reynolds. Separation and information hiding. In *POPL*, 268–280, 2004.
[26] 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.
[27] J. C. Reynolds. *The Craft of Programming*. Prentice-Hall, 1981.
[28] J. C. Reynolds. The essence of Algol. In J. W. de Bakker and J. C. van Vliet, editors, *Algorithmic Languages*. North-Holland, 1981.
[29] 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 *ECOOP* 2005.
[30] 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.