Permissions to Specify the Composite Design Pattern

Kevin Bierhoff Jonathan Aldrich
Institute for Software Research, Carnegie Mellon University
5000 Forbes Avenue, Pittsburgh, PA 15213, USA
{kevin.bierhoff,jonathan.aldrich} @ cs.cmu.edu

ABSTRACT

The Composite design pattern is a well-known implementation of whole-part relationships with trees of Composite objects. This paper presents a permission-based specification of the Composite pattern that allows nodes in an object hierarchy to depend on invariants over their children while permitting clients to add new children to any node in the hierarchy at any time. Permissions can capture the circular dependencies between nodes and their children that arise in this context. The paper also discusses verifying a Composite implementation and known limitations of the presented specification.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Requirements/Specification—*Languages*; D.2.2 [**Software Engineering**]: Design Tools and Techniques—*Modules and interfaces*; D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Design, languages, verification.

Keywords

Typestate, invariants, implementation verification.

1. INTRODUCTION

The Composite design pattern is a well-known implementation of whole-part relationships with trees of Composite objects [7]. If nodes depend on invariants over their children then it becomes challenging to verify that adding a child to a node correctly notifies the node's parents of changes [9]. In particular, these circular dependencies between nodes are hard to capture with verification approaches based on ownership [1] or uniqueness [5].

This paper presents a permission-based specification of the Composite design pattern that allows nodes in an object hierarchy to depend on invariants over their children while permitting clients to add new children to any node in the hierarchy at any time (section 3). Permissions can capture the circular dependencies between nodes and their children that arise in this context. Section 4 outlines how the presented specification can be used for verifying a simple Composite implementation.

The approach is based on the authors' work on sound reasoning about typestates in object-oriented programs [3] (briefly introduced in section 2). We therefore use a typestate-based invariant in our presentation. Section 5 discusses how this and other limitations of the presented specification can be remedied before section 6 concludes.

7th International Workshop on Specification and Verification of Component-Based Systems (SAVCBS 2008), November 9–10, 2008, Atlanta, GA, USA.

2. PERMISSIONS

This section gives a brief introduction to the approach used to specify the Composite pattern in the following section. The approach was originally developed for sound reasoning about typestates in object-oriented programs with aliasing [3] and is based, in part, on previous work on typestates for objects [5].

In our approach, developers can associate objects with a *hierarchy* of typestates, similar to Statecharts [8]. For example, we will use typestates to indicate whether a Composite node's subtree has an even or odd number of nodes.

Methods correspond to state transitions and are specified with *access permissions* that describe not only the states required and ensured by a method but also how the method will access the references passed into the method. We distinguish exclusive (unique), exclusive modifying (full), read-only (pure), immutable, and shared access (table 1). Furthermore, permissions will specify the *data group* [10] they give access to. Data groups represent orthogonal (logically independent) parts of an object's state. Thus, we can track permissions separately for each data group. We associate a set of mutually exclusive typestates with each data group and therefore will often refer to data groups as *state dimensions*. We use sans-serif all-uppercase words for data groups and all-lowercase words for states. Permissions can optionally include the state the data group is known to be in.

Permissions can only co-exist if they do not violate each other's assumptions. Thus, the following aliasing situations can occur for a given object: a single reference (unique), a distinguished writer reference (full) with many readers (pure), many writers (share) and many readers (pure), and no writers and only readers (immutable and pure).

Permissions are linear in order to preserve this invariant. But unlike linear type systems [11], they allow aliasing. This is because permissions can be *split* when aliases are intro-

Access through	Current permission has ...	
other permissions	Read/write access	Read-only access
None	unique	unique
Read-only	full	immutable
Read/write	share	pure

Table 1: Access permission taxonomy

duced. For example, we can split a unique permission into a full and a pure permission to introduce a read-only alias. Using *fractions* [4] we can also *merge* previously split permissions when aliases disappear (e.g., when a method returns). This allows recovering a more powerful permission.

Fractions are conceptually rational numbers between zero and one. In previous work, fractions below one make objects immutable; in our approach, they can alternatively indicate shared modifying access. Splitting a permission into two means to replace it with two new permissions whose fractions sum up to the fractions in the permission being replaced. Merging two permissions does the opposite. We will sometimes use permissions such as $\mathsf{immutable}(x, \mathsf{WEIGHT}, 1/2)$, which represents a permission with exactly a half fraction. Merging two of these permissions yields a $\mathsf{full}(x, \mathsf{WEIGHT})$, which gives exclusive modifying access to the WEIGHT data group but still permits $\mathsf{pure}(x, \mathsf{WEIGHT})$ permissions to the same object at the same time.

To specify invariants and method pre- and post-conditions we combine permissions (and other atomic predicates such as a variable being **true**) with linear logic operators. We will use multiplicative conjunction ($\otimes$) when two predicates must hold at the same time. Additive conjunction (&) allows internal choice between two predicates, while disjunction ($\oplus$) represents external choice. Linear implication ($\multimap$) will be used when one predicate indicates another.

3. COMPOSITE SPECIFICATION

We now turn to our sample Composite class, shown in figure 1, which is a simple implementation of the Composite pattern [7]. Every node in a Composite object tree will be represented by a Composite object. The following subsections summarize goals and assumptions before we discuss the class's invariants and method specifications.

3.1 Specification Goals

- Allow clients to add children to any node in a tree.
- Allow nodes to depend on their children in invariants.
- Ensure that adding children to a node does not violate its parents' invariants.

3.2 Assumptions

In order to keep the presentation manageable we use a simplified implementation. We assume that every node in the tree can only have up to two children. We also restrict our discussion to an extremely simple invariant: every node in the tree tracks whether its subtree contains an even or odd number of nodes (including the node itself). This allows our specification to remain in the realm of typestate. Furthermore, notice that Composite is the only type of object allowed in a Composite tree. Leafs in the tree are simply Composite objects with no children. This lets us ignore problems with inheritance for now. Finally, this specification assumes single-threaded execution.

We discuss extensions to more children and more sophisticated invariants in section 5. Our approach can be extended to include inheritance [3] and multi-threaded programs [2], but these extensions are beyond the scope of this paper.

3.3 Invariants

The focus of our Composite specification is the definition of internal invariants that allow verifying that all nodes in a Composite tree remain consistent when children are added to a node in the tree.

We define 4 data groups [10] and distinguish 2 states in each data group using the **states** keyword.

- The WEIGHT data group defines states that reflect the invariant tracked by our Composite objects: whether the number of nodes in the subtree is even or odd.
- The LEFT (RIGHT) data group each define states that indicate whether the left (right, respectively) subtree contains an even or odd number of nodes (excluding the current node).
- The PARENT data group distinguishes whether the node is an orphan (no parent) or not.

Each of our 4 data groups holds one of the fields defined in the Composite class. A permission for the WEIGHT data group, for example, therefore permits access to the *odd* field that it contains, but not the other fields. A full permission gives exclusive write access to the fields in the data group, while an immutable permission gives read-only access with the guarantee that the field will not be (silently) modified.

Unfortunately, however, the fields in the Composite class are interdependent in certain ways. In particular, the WEIGHT dimension depends on the objects referenced by the *left* and *right* fields. More precisely, it depends on whether the left and right subtrees contain an odd or even number of nodes. We will model these dependencies as permissions to a data group held by another data group. Figure 2 illustrates the permissions between a node, its parent, left child, and hypothetical client, following the specification in figure 1.

Our intuition for defining invariants is now to use immutable (or full) permissions in an invariant whenever it *depends* on the object or data group referenced with the permission. For example, the WEIGHT data group holds immutable permissions to the LEFT and RIGHT data groups in order to depend on those data groups' states. LEFT and RIGHT, in turn, hold immutable permissions to their children's WEIGHT data groups, which allows LEFT and RIGHT to depend on the children being even or odd.

Based on this structure we can define invariants separately for each data group and their states, which are marked with the keyword **invariant** and the name of the data group or state. Invariants for data groups must always hold, while an invariant for a state defines the condition under which the object is in that state. For readability, we sometimes define multiple **invariant** clauses for the same data group. All invariants defined for a data group must hold at the same time.

Following our intuition, invariants for a data group or state should only mention fields and states that are (transitively) reachable through immutable permissions from the

```
final class Composite {
states PARENT = { orphan, hasParent }
states WEIGHT = { even, odd }
states LEFT = { lefteven, leftodd }
states RIGHT = { righteven, rightodd }

boolean odd; in WEIGHT;
Composite parent; in PARENT;
Composite left; in LEFT;
Composite right; in RIGHT;

invariant PARENT: immutable(this, WEIGHT, 1/2) ⊗ parent ≠ this;
invariant PARENT: (parent = null ⊸ immutable(this, WEIGHT, 1/2)) &
                  (parent ≠ null ⊸ (share(parent, PARENT) ⊗ (immutable(parent, LEFT, 1/2) ⊗ parent.left = this) ⊕
                                                            (immutable(parent, RIGHT, 1/2) ⊗ parent.right = this)));
invariant orphan: parent = null;
invariant hasParent: parent ≠ null;
invariant WEIGHT: immutable(this, LEFT, 1/2) ⊗ immutable(this, RIGHT, 1/2);
invariant WEIGHT: (odd = false ⊸ ((this in leftodd ⊗ this in righteven) ⊕ (this in lefteven ⊗ this in rightodd))) &
                  (odd = true ⊸ ((this in lefteven ⊗ this in righteven) ⊕ (this in leftodd ⊗ this in rightodd)));
invariant even: odd = false;
invariant odd: odd = true;
invariant LEFT: left ≠ null ⊸ immutable(left, WEIGHT, 1/2);
invariant lefteven: left = null ⊕ (left ≠ null ⊗ left in even);
invariant leftodd: left ≠ null ⊗ left in odd;
invariant RIGHT: right ≠ null ⊸ immutable(right, WEIGHT, 1/2);
invariant righteven: right = null ⊕ (right ≠ null ⊗ right in even);
invariant rightodd: right ≠ null ⊗ right in odd;

Composite()
   ensures full(this, PARENT) in orphan ⊗ pure(this, WEIGHT) in odd ⊗
      immutable(this, LEFT, 1/2) ⊗ immutable(this, RIGHT, 1/2);
{ odd = true; parent = null; left = null; right = null; }

void setLeft(Composite c)
   requires share(this, PARENT) ⊗ immutable(this, LEFT, 1/2) ⊗ share(c, PARENT) in orphan ⊗ c ≠ null ⊗ c ≠ this;
   ensures share(c, PARENT) in hasParent ⊗ c.parent = this;
{
   c.parent = this;
   left = c;
   if(c.odd) {
      odd = ! odd;
      Composite p = parent;
      while(p != null) {
         p.odd = ! p.odd;
         p = p.parent;
      }
   }
}

void setRight(Composite c)
   requires share(this, PARENT) ⊗ immutable(this, RIGHT, 1/2) ⊗ share(c, PARENT) in orphan ⊗ c ≠ null ⊗ c ≠ this;
   ensures share(c, PARENT) in hasParent ⊗ c.parent = this;
{ c.parent = this; right = c; if(c.odd) { ... } }

boolean odd()
   requires pure(this, WEIGHT);
   ensures pure(this, WEIGHT) ⊗ ((result = true ⊸ this in odd) & (result = false ⊸ this in even));
{ return odd; }
}
```

Figure 1: Simple `Composite` class with invariants and method specifications

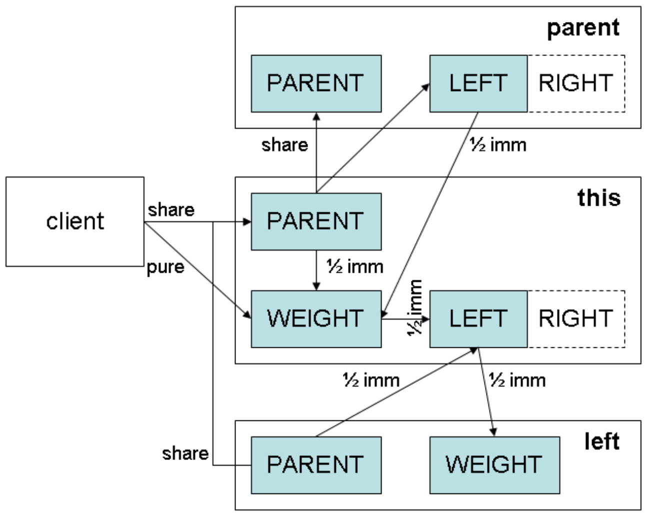

Figure 2: A sample Composite object with parent, left child, and a client that references it. Arrows are labeled with the permissions they represent. Shaded boxes represent data groups inside objects. Only relevant data groups of parent and child are shown.

data group being defined. That ensures that the needed state or field value cannot change without the knowledge of the data group mentioning the state or field in its invariant. For instance, the invariant for the lefteven state includes the *left* field, which is part of lefteven's data group, LEFT, and the state of the WEIGHT data group in the object referenced through the *left* field, for which LEFT holds an immutable permission.

We frequently use internal choice (&) between linear implications ($\multimap$) to encode situations where an *indicator* predicate implies additional facts. For example, the WEIGHT dimension uses the *odd* field as an indicator for states of other dimensions. Internal choice captures the intuition that only one of the indicating predicates can be true at the same time. For example, the *odd* flag cannot be **true** and **false** at the same time.[1] States inside data groups frequently just assert the truth of an indicating predicate.

The PARENT dimension is intended to be used by clients for adding children to nodes. Therefore, we will give out share permissions to this dimension, which allow free modification from multiple places. But in order to add children and modify the *odd* flag, the PARENT dimension holds a permission to the WEIGHT dimension (which in turn references LEFT and RIGHT). It also references the node's parent, if any. The invariants declared for the PARENT dimension are mostly for verification purposes and will be discussed in section 4.

3.4 Method Specifications

The specifications for the `Composite` methods follow from the invariants we discussed above as well as which data groups are accessed in each method. We define the method pre- and post-condition with the **requires** and **ensures** keywords, respectively.

[1]In Boolean logic, internal choice would be expressed as a regular conjunction.

The constructor creates a brand-new `Composite` object without children or parent. The absence of a parent is indicated with the state orphan in the PARENT dimension. The ability to add children comes from the returned immutable permissions for LEFT and RIGHT, which are consumed (i.e., required but not ensured) by the methods for setting the left and right child. Additional immutable permissions for LEFT and RIGHT are kept in the invariant for WEIGHT, as discussed in the previous section. We could also define a method for removing a child, which would return the respective immutable permission to the client. We chose to keep permissions for adding children with the client in order not to have to track them with more `Composite` invariants. Finally, we return a pure permission for the WEIGHT from the constructor, which clients can use to query the *odd* flag with the `odd` method. (Notice that the constructor starts out with full permissions to all data groups, some of which are immediately consumed to satisfy the new object's invariants, resulting in the declared post-condition.)

Setting the left or right child (with `setLeft` and `setRight`, respectively) requires the respective immutable permission for the LEFT or RIGHT dimension in addition to a share permission for the receiver's PARENT dimension. It also requires a share permission for the child's PARENT dimension. The child is required to be an orphan. The invariant for that state linearly implies an immutable permission for the child's WEIGHT dimension, which will be given to the new parent's LEFT (or RIGHT) dimension. In return, the child will capture the given permissions for the receiver in its invariant for hasParent, its ensured state.

Finally, the `odd` method can be used by clients to query whether a node is even or odd. We use a pure permission in the specification of this method. The post-condition uses a linear implication in a way similar to what we discussed for invariants in the previous section: the return value indicates the state of the receiver. The pure permission used in the specification for `odd` implies that the receiver's state can change without the client noticing it. (This is in contrast to immutable permissions, which exclude this possibility.)

One disadvantage of this specification is that once a node has children, the client only has a share permission to the node's PARENT dimension. This is because the initial full that is ensured by the constructor will have to be split into share permissions in order to give some of them to the node's children, as discussed above. Afterwards, clients will loose track of whether a node is an orphan or not. Therefore, our `Composite` probably should have a method `isOrphan` that can be called to test whether a node is in the orphan state. Alternatively, it might be possible to specify the class with an additional dimension that children can use internally to access their parents.

4. IMPLEMENTATION VERIFICATION

This section outlines how the specification presented above can be used to verify the implementation of the `setLeft` method for setting a node's left child. Our verification approach relies on a packing/unpacking methodology which we adapted from existing work [5, 1]. Unpacking a data group releases permissions guaranteed by invariants; packing will consume permissions required by invariants.

Figure 3 shows the `setLeft` method from figure 1 with **pack** and **unpack** commands inserted. Our unpacking *focuses* [6] on the unpacked data group [3] and makes the

```
void setLeft(Composite c)
    requires $\mathsf{share}(\mathbf{this}, \mathsf{PARENT}) \otimes \mathsf{immutable}(\mathbf{this}, \mathsf{LEFT}, 1/2) \otimes \mathsf{share}(c, \mathsf{PARENT})$ in orphan $\otimes c \neq \mathbf{null} \otimes c \neq \mathbf{this}$;
    ensures $\mathsf{share}(c, \mathsf{PARENT})$ in hasParent $\otimes c.parent = \mathbf{this}$;
{
    unpack(c, PARENT);
    c.parent = this;
    unpack(this, PARENT);
    if(parent != null)
    { unpack(parent, PARENT); unpack(parent, WEIGHT); unpack(parent, LEFT); unpack(parent, RIGHT); }
    unpack(this, WEIGHT); unpack(this, LEFT);
    left = c;
    pack(this, LEFT); unpack(c, WEIGHT);
    if(c.odd) {
        pack(c, WEIGHT); pack(c, PARENT);
        odd = ! odd;
        pack(this, WEIGHT); if(parent != null) { pack(parent, LEFT); pack(parent, RIGHT); } pack(this, PARENT);
        Composite p = parent;
        while(p != null) {
            if(p.parent != null)
            { unpack(p.parent, PARENT); unpack(p.parent, WEIGHT); unpack(p.parent, LEFT); unpack(p.parent, RIGHT); }
            p.odd = ! p.odd;
            pack(p, WEIGHT); if(p.parent != null) { pack(p.parent, LEFT); pack(p.parent, RIGHT); } pack(p, PARENT);
            p = p.parent;
        }
    } else {
        pack(c, WEIGHT); pack(c, PARENT); pack(this, WEIGHT);
        if(parent != null) { pack(parent, LEFT); pack(parent, RIGHT); pack(parent, WEIGHT); pack(parent, PARENT); }
        pack(this, PARENT);
    }
}
```

Figure 3: Verification of the `setLeft` method from figure 1

invariant of the unpacked data group available as-is even when unpacking a share permission. This means that in order to be sound, we cannot unpack a data group of an object if it is already unpacked. Unpacking the same data group of two references x and y is only allowed when $x \neq y$. This is why we require nodes to be different from their children in the Composite specification (figure 1).[2]

In the `setLeft` method we first unpack the new child, c. Since c is an orphan, we get a full permission to its WEIGHT dimension. Assigning **this** as c's parent will later require the immutable receiver permission from the method pre-condition when packing c. If the receiver has a parent then we need to unpack the permissions we have for the parent[3] in order to gain a full permission for the receiver's WEIGHT dimension (if the receiver is an orphan then that permission is part of its own PARENT invariant). At this point it is crucial that the parent points back to the receiver with its *left* or *right* field: this lets us combine the receiver's own immutable permission to its WEIGHT dimension with the one held by the parent.

Unpacking the full WEIGHT permission for the receiver yields a permission for LEFT, which we also unpack in order to assign c as the receiver's left child. Re-packing LEFT consumes an immutable permission for c's WEIGHT dimension. That leaves another immutable permission for packing c, which we can do right after testing c's *odd* flag. We can update the receiver's *odd* flag—which may have to be changed due to the new child—because we unpacked a full permission for the receiver's WEIGHT dimension, as discussed above.

After updating the receiver's *odd* flag we have to loop through its (transitive) parents to update their flags. Notice that only two objects are unpacked at a time: the object pointed to by p and its immediate parent. We can unpack both of them because p's invariants guarantee that it is distinct from its parent. We do *not* prevent cycles in the Composite tree with our specification, but since we only unpack two objects at a time we can still verify partial correctness.

Updating the parents proceeds similarly to updating the receiver (without assigning *left* or *right*). One interesting issue is that when we unpack *p.parent*, we only have an $\mathsf{immutable}(p.parent, \mathsf{WEIGHT}, 1/2)$ permission available. Later on, in the next loop iteration, *its* parent is unpacked (or we discover that is has no parent), which yields a second immutable permission, giving us $\mathsf{full}(p, \mathsf{WEIGHT})$. That allows us to to update the *odd* flag and re-pack.

5. FUTURE WORK

This section discusses limitations of the Composite specification presented in section 3 and how they can be overcome.

[2]We previously only allowed one data group in one object to be unpacked at a time [3], but we believe that the more permissive rule described here preserves soundness.

[3]We unpack both the parent's LEFT and RIGHT dimension because a child does not know if it is the left or right child.

Non-typestate invariants. Because our approach focuses on typestates we have chosen an invariant, even vs. odd, that can be expressed with typestates. But we believe that the presented specification can be adapted for other invariants specified in the WEIGHT dimension. For example, if nodes wanted to track the number of nodes in their subtree, they could declare a field *weight* and define the invariant $weight = lw() + rw() + 1$, where lw and rw are functions that return the number of nodes in the left and right subtree, respectively. Similar to subtrees remaining even or odd, these numbers are guaranteed to remain valid because WEIGHT is relying on them using immutable permissions. Notice that an invariant based on the number of nodes would shorten our specification substantially because we would not need to define the meaning of "even" and "odd" explicitly. Verifying such properties may require a theorem prover to reason about integers, which we believe could be added to our approach.

Many children. An arbitrary number of children can for example be achieved with a list or array. In order to specify invariants over this list, we will need to describe the invariant to the child held in each list element or array cell. Moreover, we will need an invariant for the children that guarantees the parent to point back to them. It appears that one could put each list element or array cell into a separate data group (with separate permissions), similar to the LEFT and RIGHT groups, and we are working on a way of supporting this in specifications.

Method calls while unpacked. We put the code for adding a child into a single method that contains a loop to iterate through the receiver's parents (see figure 3). It would be nicer to, for example, call a method on the new child to set its parent, and to call a method on the parent to update its invariant. The presented implementation was chosen because objects involved in these calls would be unpacked at the call sites. Moreover, it is harder to guarantee that no data group of any object is unpacked more than once when multiple methods may unpack objects at once. We believe that a specification language similar to Spec# [1], which can specify objects to be unpacked at method boundaries, could remove this restriction.

Precise effects. As discussed, the pure permission used for specifying the odd method (figure 1) reflects that state changes in the WEIGHT dimension can happen without the client noticing. But while our approach will "forget" whether a node was even or odd upon any effectful operation, more precise tracking of effects may enable forgetting this information only if nodes are added to the node's subtree (which is when the node's state actually changes).

Overhead reduction. The specification overhead in figure 1 is arguably high. About half of the invariants relate to the particular property we are tracking, even vs. odd. The rest represents a pattern for encoding circular dependencies between objects with immutable permissions. We believe that a developer could reuse this pattern to track the property of interest on top of it. This suggests introducing a specification construct for defining circular dependencies, which would internally be translated into the invariants shown, to reduce specification size.

6. CONCLUSIONS

This paper presents a specification of the Composite design pattern with permissions that allows nodes to depend on their children in invariants and allows clients to add children to any node in a Composite tree at any time. Permissions can express the circular dependencies between nodes that are needed to guarantee that adding a child to a node correctly updates the parents' invariants. We discuss how shortcomings of the presented specification can be overcome with a richer specification language than the one used in this paper. In particular, we believe that our permission based approach can be extended from typestate-based to more interesting invariants and to arrays or lists of objects.

Acknowledgments. We thank Nels Beckman and the anonymous reviewers for their helpful feedback on earlier versions of this paper. This work was supported in part by the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems", DARPA contract HR00110710019, and NSF grant CCF-0811592.

7. REFERENCES

[1] 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, June 2004.

[2] N. E. Beckman, K. Bierhoff, and J. Aldrich. Verifying correct usage of Atomic blocks and typestate. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, Oct. 2008. To appear.

[3] K. Bierhoff and J. Aldrich. Modular typestate checking of aliased objects. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 301–320, Oct. 2007.

[4] J. Boyland. Checking interference with fractional permissions. In *International Symposium on Static Analysis*, pages 55–72. Springer, 2003.

[5] R. DeLine and M. Fähndrich. Typestates for objects. In *European Conference on Object-Oriented Programming*, pages 465–490. Springer, 2004.

[6] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *ACM Conference on Programming Language Design and Implementation*, pages 13–24, June 2002.

[7] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[8] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.

[9] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, submitted for publication.

[10] K. R. M. Leino. Data groups: Specifying the modification of extended state. In *ACM Conference on Object-Oriented Programming, Systems, Languages & Applications*, pages 144–153, Oct. 1998.

[11] P. Wadler. Linear types can change the world! In *Working Conference on Programming Concepts and Methods*, pages 347–359. North Holland, 1990.