Maintaining Invariants Through Object Coupling Mechanisms

Eric Kerfoot   Steve McKeever
Oxford University Computing Laboratory
{eric.kerfoot, steve.mckeever}@comlab.ox.ac.uk

Abstract

Object invariants are critical components to the specification of object-oriented systems, which define valid states for objects and how they may be interrelated. A complex problem is created when an invariant relies on objects that are externally aliased and modified, since the invariant's class cannot ensure that modification to these objects preserves the invariant. This paper informally introduces a method of coupling objects called the Colleague Technique, which creates strong relationships between objects whose invariants rely on one another and defines additional conditions to ensure these invariants. The technique builds on the classical technique by providing a method of ensuring object-dependent invariants are maintained by the operations of an object-oriented system. We demonstrate our technique using the Java programming language and the JML specification language.

1. Introduction

An object's invariant is a predicate stating conditions for its members which defines the valid states of the object. This predicate is expected to be maintained by the object's operations if their contracts are met, and by clients if members are modified directly. This leads to the expectation that well-formed conditions would ensure this soundness property in the Design-by-Contract (DbC) technique [14]. However sources of unsoundness are present even with well-defined conditions, one primary cause being the situation where an invariant relies on an object for its condition that is aliased outside the invariant's object. The implication of this situation is that this dependee object could be modified by another in a valid way, but which may still invalidate the invariant that depends upon it.

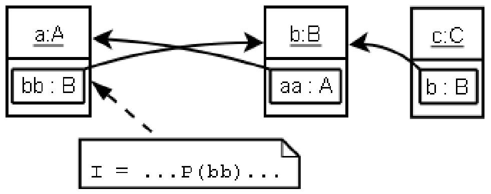

Figure 1. The Indirect Invariant Effect

To illustrate the example, consider Figure 1 that illustrates the object *a* whose invariant depends on object *b*. If *b* were modified by object *c* this could invalidate *a*'s invariant without violating any of *a*'s method contracts. The problem was first identified in [14] where it is described as the Indirect Invariant Effect. Invariants that experience the effect are dependent on instances of another object type, which are said to be vulnerable to such an invariant.

Although the problem is simply stated, it is found in many common design patterns and idioms in object-oriented systems where it can be a significant source of error. In these situations the assumption of soundness, which is that an invariant will be satisfied if method contracts are met, no longer holds. This is a result of the fact that objects can be modified in ways that satisfy their contracts, and so also their invariants, but break the invariants of other objects that depend on them.

For example, a set of iterators depend on the collection over which they iterate for their invariant conditions, such that if the collection were to have too many objects removed, an iterator may refer to a position in the collection that no longer exists. The problem also occurs in self-referential classes whose invariants rely on instances of themselves. An example is a person class with a spouse attribute and an invariant which states that the spouse's spouse must be the current `this` object. In this case it may occur that one spouse is assigned a new spouse and so breaks the old spouse's invariant. Any additional invariant that such a class may have would rely on the assumption of marriage being an exclusive bidirectional relationship between two objects.

The effect is addressed in [11] which presents a solution as an axiomatic verification methodology. The verification methodology is stated in terms of Hoare logic and concludes with a scheme of proof obligations for invariants and method conditions. The added obligations are complex and cumbersome, requiring a degree of global reasoning. This is a result of the need to globally verify that all vulnerable objects do not at any point invalidate invariants that depend on them.

Another solution to the problem of objects being vulnerable to an invariant is found in confined type and object ownership models [2, 4, 5, 10], where the type system prevents objects from being aliased outside of their creator object. Such a method may result in runtime systems organized into hierarchical series of references with upper level objects owning those below. An object's invariant may only depend on objects that are owned, which are safe from third part modification, thus preventing the Indirect Invariant Effect. This condition on invariants and the required confined/owned properties can be statically checked, such as in the Universe type system [7, 15] that encodes object ownership as special reference types.

What ownership requires is that the invariant of an object can only rely on owned objects, which only the invariant's object may reference and directly modify. This ensures that an object's invariant cannot be broken when objects it depends on are modified, since the conditions of the object's methods ensure that any modification is always valid.

The hierarchical nature of object organization that ownership creates has certain limitations in how objects may be related. For example, straight-forward ownership disallows iterators whose invariants depend on data structures that they do not own, recursive data structures such as linked lists, or recursive types like the person class where a person cannot own its spouse. Different ownership techniques address these issues, such as the visibility technique [16] that weakens the ownership requirements at the cost of greater proof obligations, but which again add to the complexity of verifying correctness.

The solution [3] used in the Boogie methodology is quite similar to the proposed solution in this paper. Using special language constructs, objects can relate themselves to "friend" objects that share responsibility for their friend's invariant. This builds on the Boogie methodology described in [17] that partially addresses the issue with a form of ownership. However this methodology relies on these specialized constructs, additional auxiliary variables, and specialized assertion statements, thus is more difficult to apply in a more general DbC approach. A more preferable approach would define a method that can be used with existing DbC analysis and verification approaches.

The root problem with the Indirect Invariant Effect is that invariants reliant on other objects create dependency relationships that are weakly represented, and so a method of defining these relationships more concretely would lead to a solution. Our Colleague Technique addresses this dependency problem by providing a mechanism of coupling objects whose invariants rely on one another, and defining additional invariant conditions which ensure that no modification to either object invalidates the other's invariant. The disadvantage of this method is reduced software reuse that's a consequence of close coupling, however this is outweighed by the ability to soundly predicate invariants on external objects. What the technique does not provide is an encapsulation mechanism, which can be provided using an ownership methodology that prevents the internal representation of an object from being externally aliased.

Collegiality is defined as an additional technique that is used with classical DbC methods, such that if a specification is correct classically then applying the Colleague Technique will result in a correct specification. This resulting specification will also use invariants and conditions as defined in the classical technique, and so allows existing analysis, verification, and code generation techniques to be applied in conjunction with collegiality. The technique is described using Java [9] and JML [12] as the specification language which it extends with a new annotation. Thus existing tools and analysis techniques developed for JML can be used in conjunction with the technique.

This section has discussed the Indirect Invariant Effect and its consequences. The remainder of this paper will discuss the Colleague Technique as a solution to this problem. Firstly, the technique will be defined as an additional concept to classical DbC constructs. Object types that are suitable as colleague types must meet certain requirements that are discussed next. This is followed by a description of how additional invariant conditions are formulated which protect invariants from being invalidated by operations on dependee objects. Finally the technique will be applied to the Iterator and Person examples discussed in this section.

2. The Colleague Technique

The previous section has outlined the Indirect Invariant Effect problem and how it introduces unsoundness in the classical invariant technique. This section will describe the Colleague Technique and how it creates strong relationships between objects whose invariants rely on one another. These relationships are used to define additional conditions on the invariants of both objects such that modification to one will not invalidate the invariant of the other.

2.1 Definition

The relationship between objects used by the Colleague Technique is defined by stating that each object type has an attribute that refers to an instance of the other type, or is a set of such references. These two attributes are declared as being each other's colleague, and the types they reside in as colleague types. The Colleague Technique is thus an additional specification concept with specific semantic requirements that solves the problem of predicating invariants on other objects.

The Indirect Invariant Effect is caused when an invariant is allowed to rely on any arbitrary object. With the Colleague Technique an invariant may rely on an object if it is referred to by one of the object's colleague attributes. This restriction limits which objects an invariant may rely on, and adds the knowledge to the specification of a colleague type that its instances may be relied upon by the invariants of other objects. With this knowledge the invariant of a colleague type can be augmented with additional conditions that prevent modifications which would invalidate their colleague's invariant.

Definition *Two object types* `A` *and* `B` *are colleague types if* `A` *has an attribute* `bb` *which is a single or a set of* `B` *references, and* `B` *has an attribute* `aa` *which is a single or a set of* `A` *references, and* `aa` *is defined as being collegial with* `A.bb` *and* `bb` *as collegial with* `B.aa`*. The invariants of* `A` *and* `B` *may only rely on objects referred to in these attributes, owned objects, and primitive values.*

An instance of `A`*,* `a`*, and one of* `B`*,* `b`*, are collegial if a reference to* `B` *is stored in* `a.bb` *and a reference to* `a` *is stored in* `b.aa`*. This is illustrated in Figure 2. Both* `a` *and* `b` *are responsible for maintaining that the relationship holds by ensuring the cross referencing and removing references when the relationship is established between them or when either object is removed from the system.*

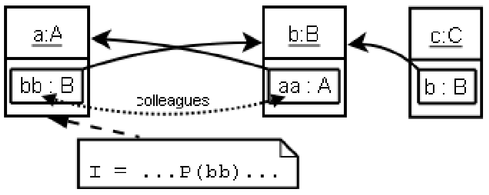

Figure 2. The Collegial Solution

This concept of explicit relationships between the attributes of two object types is similar to that used by the object-based Booster specification language [6]. Booster is descendent from Z [18] and the B Method [1], but is a domain-specific language that targets database systems. The explicit relationships are used as a means of maintaining associations between data, and through the use of Weakest Precondition [8] methods to auto-generate conditions that ensure the relationship.

This bidirectional binding between objects creates ad-hoc contexts [5] that are similar to ownership contexts, except that there is no owner/owned relationship, but a partnership between objects. If a collegial attribute is a reference value then that object may have only one colleague of that type, and if it is a set then it can have multiple colleagues.

The purpose of having this bidirectional relationship is so that an assumption about responsibility can be made by both collegial partners. If an object has a reference to a colleague object, then it can be assumed that the colleague will reference it as well. Thus an object's specification can assume that these colleagues will not allow any modification to themselves that breaks the invariant conditions that depend on them.

Figure 2 illustrates how a depends on b since its invariant includes the predicate $P(bb)$ and bb stores a reference to b. The specification for type A can safely define such an invariant since the assumption exists that b's attribute aa will alias a, and that B's invariant will include a condition on aa that prevents modifications that would invalidate P. Thus the bidirectional property of the relationship is critical to the technique, and so certain requirements and facilities must be present to ensure that the relationship is created and broken correctly. These requirements are discussed in the next section.

2.2 Requirements

The colleague relationship imposes requirements on the invariants and methods of object types that must be met for them to be used as colleague types. These requirements are necessary for the technique to correctly guarantee invariants:

- A colleague object type can only have one collegial attribute for any colleague object type. For example, an object type *A* cannot define two attributes to both be collegial with attributes of object type *B*, even if the attributes are different. Otherwise an object type can be defined to be collegial with any number of other object types.

  This restriction prevents situations where a contradiction may arise between two conjuncts of an object's invariant. If an object type *A* were allowed to have two attributes that were collegial with two attributes of *B*, then instances of *A* and *B* could become double collegial through these two means of association. It would then be possible for invariant conditions to be placed on the collegial attributes that would be contradictory if they were asserted for the same object. This also eliminates the possibility of circular dependency of invariants between two objects.

- Each invariant is responsible in maintaining its colleagues' invariant, and so colleagues must be defined in a way such that they and parts of their invariants are visible to one another. This may require that the invariants be declared as publicly visible and only rely on public members.

  This is needed since an invariant of an object will be augmented with added conditions that reflect the invariants of colleagues. If the colleague objects' invariants were not public, that is they were not publicly visible or relied on non-public members, then these added conditions could not be formulated or would be required to access non-public members.

- Both invariants may only rely on objects that are colleagues of their objects or `this`.

- They may also only rely on the members of these objects – but not members of members – which do not evaluate to regular reference types nor depend on regular reference types for their values. This restricts how complex invariants can be, but any degree of complexity can be created by classes providing appropriate methods that return useful information. Such methods, for example, may calculate values that can be used in specifications that requires other objects.

  The purpose of this restriction is to limit which objects are dependees of an invariant. If an invariant were allowed to state a condition dependent on regular objects, even if they were referenced by attributes of a colleague object, then this would reintroduce the Indirect Invariant Effect. This prevents the situation where, if an invariant includes a predicate of the form "$P(this.x.y.foo())$" where the value of *foo* is of interest, both objects *x* and *y* become dependees. In this situation *x* will not have sufficient invariant conditions to prevent invalidating modifications to *y*, since *y* is accessed directly and not through a method of *x*, which would be able to perform an invariant check which would prevent invalidating modifications to *y*.

- If a method is used in an invariant, this requires that the method's return value depend only on colleague objects, primitive values, or other methods of same object that are similarly restricted. These methods must also be pure, that is they are side-effect free.

This technique has limitations in that each colleague must be pre-defined to be part of a collegial relationship, as opposed to ownership where any arbitrary object can be owned by another. Greater coupling between objects reduces reusability, however invariants that rely on other objects already create this coupling, which the Colleague Technique formalizes.

Collegiality provides a method to closely couple two object types, whose instances may be aliased in different parts of a system, in a way that wouldn't be permitted with ownership. From the perspective of a software module, it allows an object that relies on the module's internal state to be passed over the module's public interface boundary to the client. This object is used to provide some functionality of the module, but since it relies on the internal structure it must be defined in a way that does not adversely affect this state but also allows it to be aliased by the modules's client objects. The Colleague Technique aims to provide a method of specifying such objects so that this can be achieved.

Constructing and breaking the collegial relationship is important since the cross referencing must be maintained. If one object was collegial with another, then it relies on that colleague object to alias it and so prevent operations that would break its invariant. If the relationship between two objects was malformed in that it became unidirectional, then the assumption about invariant responsibility breaks down.

What this implies is that creating and breaking the relationship are specific operations that the code of an object type should not be responsible for. Although the technique can be defined purely as a specification, it is helpful to describe these operations in terms of helper methods that define the criteria for determining when two objects are collegial and managing the relationship between colleagues:

- To access collegial references, an accessor is defined for each collegial attribute that returns the reference value if the type is a singleton or an iterator if the type is a set. This accessor is called '`getY()`' for an attribute named `Y`, eg. an attribute named '`foo`' is accessed by '`getfoo()`'.

- A colleague type must have a boolean-returning method '`isAssociated`' for every type *X* that it is collegial with, which takes a reference of type *X* and determines if it is an object that is a colleague of the current object.

- Determining if an association relationship can be formed is performed by a method called '`isAssociable`' that accepts the colleague candidate as an argument.

- A colleague type must have a void-returning method '`associate`' for every type *X* that it is collegial with. This method takes as the single argument a reference of type *X* which it adds to the collegial attribute. The method '`associate`' is then called on the argument object, passing `this` as the argument.

- A fourth method for a collegial type called '`disassociate`' is defined for every collegial type *X* which has the corollary effect of disengaging two objects from a collegial relationship, by assigning `null` to singleton types or removing the given reference from the collegial set type.

Thus a set of requirements are defined that an object type must meet so as to be useable as a colleague type, and a set of helper methods are described which are essential to the operation of the technique. These methods need not be concrete but may be abstract methods in a specification, however if colleague relationships need to be concrete in the implementation of the system then these methods would need to be as well. The next section will build on the collegial relationship and discuss how this is used to construct new invariant conditions that ensure invariant soundness.

2.3 Invariant Conditions

The purpose of entering two objects into a collegial relationship is to allow one or both to predicate their invariants on the other, such that each object's specification has the information to ensure the object's methods do not violate the others invariant. This is achieved by adding extra conditions to an object's invariant that ensure the properties its colleagues require of it.

These extra conditions are derived from the part of the object's invariant predicated on the colleague attribute, which are then expressed in terms of the colleague object itself. Taking a condition placed on a member of a colleague attribute and replacing the name of the attribute with `this` restates the condition from the perspective of the colleague object itself. This new condition, which states the same property but from the perspective of the other colleague object, can then be used as the needed additional condition.

Given the object types A and B from the above discussion and their respective invariants I_A and I_B, the part of I_A predicated on `bb` is denoted by P which must be in a form where every member access must explicitly begin with '`this.`' (called normal form in this context):

$I_A = ...P(bb)...$ – if `bb` is a singleton type
$I_A = ... \forall\, i : bb \mid S \bullet P(i)...$ – if `bb` is a set type, given S
$I_A = ... \exists\, i : bb \mid S \bullet P(i)...$ – if `bb` is a set type, given S

These three forms of the invariant for A relate members of `bb`, either attributes or values returned from pure method calls, to members of A or constant values. P is stated in the perspective from A to B, and so to reverse the perspective and produce an invariant for B, the roles of `this` and `bb` must be reversed. This takes P and produces a mirror P_m stated in terms of `aa`.

If `aa` is a singleton attribute then there are two forms of the mirror P_m predicate:

$$P_m(aa) == P[this, this.aa/this.bb, this]$$

– if `bb` is a singleton

$$P_m(aa) == S[this, this.aa/i, this] \Rightarrow P[this, this.aa/i, this]$$

– if `bb` is a set

If `aa` is a set attribute then the two forms are quantified over the elements of the set:

$$P_m(aa) == \forall\, i : this.aa \bullet P[this, i/this.bb, this]$$

– if `bb` is a singleton

$$P_m(aa) == \forall\, i : this.aa \mid S[this, i/i, this] \bullet P[this, i/i, this]$$

– if `bb` is a set

Therefore the invariant of B has the additional requirement of maintaining the predicate $P_m(aa)$:

$$I_B = ... \wedge P_m(aa)$$

The predicate P states relationships between the members of the classes A and B, and P_m states the same relationships but from the perspective of the other colleague type. This has the effect of swapping collegial references with `this` wherever they occur in P and reverses the direction of the predicate.

If, for example, P represented the expression '`this.bb.m<10`', then the mirror P_m would equal '`this.m<10`', which would ensure that the required property of `m` would be maintained. For a more complex example take P to represent '`this.bb.m==this.n()`' for some method `n`, then the mirror P_m is '`this.m==this.aa.n()`'.

There is another possible original form of the invariant other than the three given above. If bb is a singleton which may be set to `null` (that is it is nullable in JML terms) then the P predicate would be false when this occurs, thus an implication relation is used to guard against this possibility:

$$I_A = ...bb \neq \texttt{null} \Rightarrow P(bb)...$$

The mirror invariant of this form is derived by taking P and applying the above transformation. If aa is a set type then this P_m becomes the resulting invariant, but if it is a singleton that is nullable then a guard implication is used in this instance as well:

$$I_B = ...aa \neq \texttt{null} \Rightarrow P_m(aa)...$$

In the presence of inheritance where an object type can inherit or implement a colleague type, it is not difficult to see that behaviour subtyping [13] is necessary for the technique to work. If this were not the case then an object type may inherit from a collegial type and not be responsible for the inherited mirror invariant, thus even if it remains internally consistent the invariants of dependent objects may be invalidated.

2.4 Results

The Colleague Technique as described is used to make explicit the relationships created by invariant dependencies. The purpose in doing so is to develop a means of preventing the Indirect Invariant Effect from allowing invariants to become invalidated without contractual violations. The additional conditions that are added to the invariants of colleague types achieve this, and are dependent upon the fact that only their colleague types will depend on them for their invariants.

To understand how the technique provides this guarantee, consider the conditions that the classical DbC technique places on a method call. The precondition and invariant of an object must hold before a method begins, and since the mirror invariant must also be asserted here then the object is guaranteed to be in a state that does not break the invariant of another. When a method exits, the postcondition and invariant is asserted which again makes the valid state guarantee. By encoding the reciprocal responsibilities that collegial objects have to one another as invariant conditions, the Colleague Technique uses existing DbC methodologies to safeguard object-dependent invariants. Thus the Colleague Technique does not require additional proof obligations in addition to those used in a pre-existing verification methodology.

An implementation, in Java and using JML, of the iterator problem discussed previously demonstrates how the technique prevents an instance of unsoundness in the classical DbC approach. An additional "collegial" annotation is used to declare those attributes that are collegial with which other object type, and with what attribute.

Figure 3 lists the code for this example. It states that the attribute `iterators` of *List* is collegial with *List* or *ListIterator*. The invariant of *ListIterator* that relies on the instance of *List* it iterators over states that its size must not be less than what it was when the iterator was instantiated:

```
this.list.size()>=this.last
```

To ensure that this does not happen, `List` must have an invariant that ensures its size is never less than the `last` attribute of any associated iterators:

```
(\forall ListIterator i; this.iterators.contains(i);
                this.size()>=i.last)
```

Since the `associate` method constructs the relationship correctly, this additional invariant prevents the removal of enough elements from an instance of *List* to break an associated iterator's invariant. To allow the removal of elements again from a *List* instance, it would be required to disassociate collegial iterators, which occurs when they are no longer needed and are removed from the system. Thus collegiality forces coordination between iterators and collections, which is implicitly required by the fact that iterators are dependent on their collection's state.

This invariant was generated using a prototype Java tool that has been successfully used with this example and the Person example in described below. The tool analyzes the invariants of input Java classes, generates mirror invariants using the methodology outlined in this paper, and outputs the classes again with the mirror invariants and helper methods added. The resulting classes can be compiled into standard Java using the Common JML Tools[1], which adds runtime assertion checks to the compiled bytecode. The resultant classes have been analyzed through testing and successfully provide runtime checks that prevent the Indirect Invariant Effect.

This tool demonstrates how the described technique can be used in conjunction with existing DbC techniques to close the unsoundness gap created by object-dependent invariants. With only the additional `collegial` annotation augmenting standard JML, the tool produces resulting code that has only standard JML annotations and standard Java code, such that other tools that analyze and transform JML-annotated Java code can be subsequently used. A more sophisticated tool may be able to identify attributes of classes that need to be collegial without the additional `collegial` annotation, thus without adding significantly new specification constructs or methodologies that other solutions require, the Colleague Technique effectively addresses the problem of object-dependent invariants and can be employed with a relatively simple code-generating tool.

The second discussed example involved self-referential types, such as the spouse example in Figure 4. The invariant of the class requires that one's spouse be married to one's self. The method by which `associate` operates, which ensures the cross-referencing of colleague objects, would guarantee that this invariant would always be true if the `spouse` parameter was collegial as the code defines. Invariants that state conditions on members of colleagues can also be used in this instance, but would still require additional conditions stating the same property for the local attributes.

3. Conclusion

This paper has described the Colleague Technique, and its associated ownership technique, that is stated as a solution for the Indirect Invariant Effect. The effect is a critical problem with the classical DbC invariant technique since many common design patterns and programming idioms rely on the aliasing of objects within a system.

This technique defines a method of correctly constructing a relationship between objects whose invariants depend on one another, and how additional conditions ensure that operations on either will not invalidate the other's invariant. This discussion has been done in terms of concrete Java methods and attributes, however the technique can be defined in terms of abstract model variables entirely in some cases and without the concrete helper methods. Either as a concrete or abstract component of a specification, the purpose of the technique is to make explicit the relationship between objects that are created when an invariant relies on other objects for its conditions. What the technique does not provide is a method of guaranteed encapsulation, which is best accomplished using a lightweight method of ownership.

The net result of this technique is to close the unsoundness gap created in the classical DbC technique cause by invariants

[1] http://sourceforge.net/projects/jmlspecs/

```
class List {
    private /*@ spec_public @*/
            ArrayList items = new ArrayList();
    private /*@ collegial ListIterator.list; @*/
            Set iterators = new LinkedHashSet();

    // The mirror invariant derived from ListIterator
    //@ invariant (\forall ListIterator i;
    //@   this.iterators.contains(i); this.size()>=i.last);

    //@ requires o != null;
    //@ ensures this.items.contains(o);
    public void add(Object o) { this.items.add(o); }


    //@ requires i>=0 && i<this.size();
    //@ ensures \result == this.items.get(i);
    public /*@ pure @*/ Object get(int i)
            { return this.items.get(i); }

    //@ ensures: \result == this.items.length();
    public /*@ pure @*/ int size()
            { return this.items.length(); }

    //@ requires i >= 0 && i < this.size();
    //@ ensures !this.items.contains(
    //@         \old(this.items.get(i)));
    public void remove(int i) { this.items.remove(i); }

    public ListIterator iterator()
        { return new ListIterator(this);}
}

class ListIterator {
    private /*@ nullable collegial List.iterators; @*/
            List list;
    private /*@ spec_public @*/ int position=0, last;

    //@ invariant this.position<=this.last;

    // The invariant dependent on the colleague object
    //@ invariant this.list!=null ==>
    //@         this.list.size()>=this.last;

    //@ requires this.isAssociable(l);
    //@ ensures this.isAssociated(l);
    public ListIterator(List l){ this.associate(l);
            this.last=l.size(); }

    //@ requires this.list != null;
    //@ ensures \result == this.position<this.last;
    public /*@ pure @*/ boolean hasNext()
            { return position<=last; }

    //@ requires this.hasNext();
    //@ ensures this.position==\old(this.position)+1;
    //@ ensures \result==this.list.get(\old(this.position));
    public Object next(){ this.position++;
            return this.list.get(this.position-1); }

    protected void finalize()
            { this.disassociate(this.list); }
}
```

Figure 3. Iterator Collegial Example

```
class Person {
    private /*@ nullable collegial Person.spouse @*/
            Person spouse;

    //@ invariant this.spouse != null ==>
    //@     this.spouse.spouse == this;
    ...
}
```

Figure 4. Marriage Example

relying on externally aliased objects. Allowing invariants to be predicated on objects is an important component when specifying complex layered object structures, and so a method that ensures the soundness of the technique, i.e. if the conditions of operations are met then the invariants will remain valid, contributes significantly to the correctness and applicability of this formal method to real-world complex software engineering challenges.

This paper has informally defined and discussed the Colleague Technique by examining the problem and the proposed solution. Future work with the technique will elaborate on how it can be integrated with JML and be used as an abstract or concrete specification technique. Proving the property that colleague specifications derived from correct specifications are themselves correct, and that it does solve the problem of soundness with object-dependent invariants, is also part of future research with the technique. Through induction on the method of creating mirror invariants from original invariants, the proof must show that the mirror invariants are well-formed, well-typed, and do not represent new restrictions on the system. To prove that the technique does solve the soundness problem, it will be necessary to formally express an invariant's dependence on objects, and use this to demonstrate that the technique correctly guards against invalidating modifications. The development of tool support is also planned, whose objective is to analyze and possibly prove the correctness of programs that use the technique.

References

[1] Jean-Raymond Abrial. *The B Book: Assigning Programs to Meanings*. Cambridge University Press, August 1992.

[2] Paulo Sérgio Almeida. Balloon types: Controlling sharing of state in data types. *Lecture Notes in Computer Science*, 1241:32, 1997.

[3] Mike Barnett and David Naumann. Friends need a bit more: Maintaining invariants over shared state. In Dexter Kozen and Carron Shankland, editors, *Mathematics of Program Construction, 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004, Proceedings*, volume 3125 of *Lecture Notes in Computer Science*, pages 54–84. Springer, 2004.

[4] Boris Bokowski and Jan Vitek. Confined types. Technical report, 1999.

[5] David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *Proceedings of the 13th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA-98)*, volume 33:10 of *ACM SIGPLAN Notices*, pages 48–64, New York, October 18–22 1998. ACM Press.

[6] Jim Davies, Charles Crichton, Edward Crichton, David Neilson, and Ib Holm Sørensen. Formality, evolution, and model-driven software engineering. In *Proceedings of the Brazilian Symposium on Formal Methods (SBMF 2004)*, volume 130 of *Electronic Notes in Theoretical Computer Science*, pages 39–55, May 2005.

[7] W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, October 2005.

[8] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[9] James Gosling et al. *The Java Language Specification*. GOTOP Information Inc.

[10] John Hogg. Islands: aliasing protection in object-oriented languages. In *OOPSLA '91: Conference proceedings on Object-oriented programming systems, languages, and applications*, pages 271–285, New York, NY, USA, 1991. ACM Press.

[11] Kees Huizing and Ruurd Kuiper. Verification of object oriented programs using class invariants. In *FASE '00: Proceedings of the Third Internationsl Conference on Fundamental Approaches to Software Engineering*, pages 208–221, London, UK, 2000. Springer-Verlag.

[12] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. JML: A notation for detailed design. In Haim Kilov, Bernhard Rumpe, and Ian Simmonds, editors, *Behavioral Specifications of Businesses and Systems*, pages 175–188. Kluwer Academic Publishers, 1999.

[13] B. Liskov and J. Wing. Behavioral subtyping using invariants and constraints, 1999.

[14] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, second edition, 1997.

[15] P. Müller and A. Poetzsch-Heffter. Universes: A type system for alias and dependency control. Technical Report 279, Fernuniversität Hagen, 2001.

[16] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. Technical Report 424, ETH Zurich, March 2005.

[17] K. Rustan, M. Leino, and P. uller. Object invariants in dynamic contexts, 2004.

[18] J. M. Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science. Prentice Hall, 2 edition, 1992.