Verifying Multi-object Invariants with Relationships

Stephanie Balzer and Thomas R. Gross

ETH Zurich

Abstract. Relationships capture the interplay between classes in object-oriented programs, and various extensions of object-oriented programming languages allow the programmer to explicitly express relationships. This paper discusses how relationships facilitate the verification of multi-object invariants. We develop a visible states verification technique for Rumer, a relationship-based programming language, and demonstrate our technique on the Composite pattern. The verification technique leverages the "Matryoshka Principle" embodied in the Rumer language: relationships impose a stratification of classes and relationships (with corresponding restrictions on writes to fields, the expression of invariants, and method invocations). The Matryoshka Principle guarantees the absence of transitive call-backs and restores a visible states semantics for multi-object invariants. As a consequence, the modular verification of multi-object invariants is possible.

1 Introduction

Invariants provide a foundation for verifying programs [1], and various object-oriented programming and specification languages [2–4] have adopted invariants for objects. An *object invariant* captures the properties of an object that the object exhibits in its consistent states. Object invariants are central to a wealth of object-oriented verification techniques [5–12]. A key issue for any practical verification technique for an object-oriented language is the ability to modularly verify a program so that modules (i.e., classes) can be verified independently from each other.

Modular verification is straightforward as long as an object invariant constrains only the state of the current object and provided that an object's fields can be written to only by the object's own methods. Unfortunately, single-object invariants rarely express the constraints of real-world software, which typically asks for multi-object invariants. A *multi-object invariant* relates several objects and constrains not only the state of the current object but also the state(s) of the object(s) it refers to. The reasoning about a multi-object invariant, however, is no longer modular. For instance, if there are aliases to the referenced objects, the referenced objects may be altered in ways compromising the invariant.

Multi-object invariants compromise also the adoption of a visible states semantics for invariants. A *visible states verification technique* [9] requires an object to meet its invariant in the initial and final states of method executions

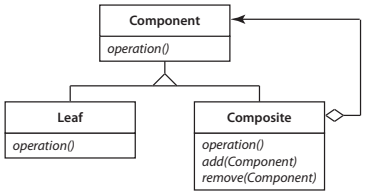
M. Mezini (Ed.): ECOOP 2011, LNCS 6813, pp. 359–383, 2011.

(i.e., the visible states) but allows an object to temporarily break its invariant during the execution of a method. A visible states semantics for invariants facilitates data type induction [1, 13, 14] as a proof technique: each method may assume that the invariant holds in the method's initial state, provided that each method ensures that the invariant holds in the method's final state. To facilitate data type induction, a verification technique needs to guard re-entrant method invocations (*call-backs*). A call-back occurs if a method, executing on an object o, invokes a method $n()$ either directly or transitively (by further method invocations) on the original object o. Since $n()$ is invoked in a state when o's invariant may be temporarily broken, $n()$ should not be allowed to assume o's invariant in its initial state. Data type induction can be restored by imposing additional obligations on the caller of a method. Namely, a calling method $m()$ is required to re-establish the invariants of those objects O that are vulnerable to $m()$'s execution, provided that the objects O are possible receivers of the direct or transitive method invocation $n()$. This obligation can be easily implemented for single-object invariants by requiring a method to re-establish its receiver's invariant before invoking a method. However, imposing the same obligation for multi-object invariants is *(i)* generally infeasible since the transitive receivers of method invocations are statically unknown, and also *(ii)* too limiting since a method of an invariant-declaring class may need to invoke methods on the objects related by the invariant to re-establish the invariant.

Existing techniques [5–12] for verifying multi-object invariants differ in how they address the challenges mentioned above as well as in the range of verification problems they can handle. Given the challenges that real-world, object-oriented programs pose for invariant-based verification, it has been questioned whether the object invariant is the correct foundation for verifying object-oriented programs [15]. In this paper, we demonstrate that an object-oriented programming language with explicit support for *relationships* enjoys properties that facilitate the verification of real-world programs with invariants. Relationship-based programming languages [16–24] complement object-oriented programming languages with the programming language abstraction of a relationship to capture the interplay between objects. We introduce a visible states verification technique for Rumer, a simple relationship-based programming language developed to explore relationships [20]. The verification technique leverages the particular modularization properties of the Rumer language that we summarize as the "Matryoshka Principle". The principle relies on a *stratification* of classes and relationships and stipulates restrictions on writes to fields, the expression of invariants, and method invocations. It translates into a stratification of invariants and guarantees the absence of transitive call-backs and, as a consequence, restores a visible states semantics for invariants. To facilitate modular reasoning about shared state, the verification technique leverages *member interposition* [20, 25] and *extent ownership*, two orthogonal language features supported in Rumer. We demonstrate our verification technique on the Composite pattern.

2 Running Example

The Composite pattern has been suggested recently as a verification challenge [12, 26] and served as the "Challenge Problem" for the 7th International Workshop on Specification and Verification of Component-Based Systems (SAVCBS). Figure 1(a) shows the class diagram of the Composite pattern. As indicated by the UML aggregation, the pattern *"composes objects into tree structures to represent part-whole hierarchies and lets clients treat leaf objects and composite objects uniformly"* [27]. Figure 1(b) shows a slightly modified version of the Composite pattern implementation presented in [26]. In accordance with the UML class diagram, the implementation distinguishes the classes `Component` and `Composite` and represents the UML aggregation by means of a `parent` reference and a `children` collection in `Component` and `Composite`, respectively. The implementation further establishes a field `total` (line 3 in Fig. 1(b)) which indicates the total number of children components that can be reached from a component.



(a)

```
1 class Component {
2   protected Composite parent;
3   protected int total = 0;
4 }
5
6 class Composite extends Component {
7   private Collection<Component> children;
8
9   public Composite() {
10    children = new Vector<Component>();
11  }
12
13  public void addComponent(Component c) {
14    children.add(c);
15    c.parent = this;
16    addToTotal(c.total + 1);
17  }
18
19  private void addToTotal(int incr) {
20    total = total + incr;
21    if (parent != null) {
22      parent.addToTotal(incr);
23    }
24  }
25 }
```

(b)

Fig. 1. Object-oriented implementation of the Composite pattern. (a) UML class diagram of the Composite pattern. (b) Implementation of the Composite pattern in a Java-like language as suggested in [26].

The Composite pattern gives raise to a number of interesting invariants. The SAVCBS 2008 challenge problem, in particular, stipulates the invariant that the value of a component object's `total` field must be equal to the number of children components contained within the sub-tree rooted at the component object. This invariant is an instance of a multi-object invariant since it is violated

by any addition or removal of a component to or from a composite's sub-tree. Method `addComponent()` accounts for this violation and triggers a bottom-up traversal of the composite tree to update the total field of a composite and of all its parent composites. The actual update is achieved by the recursive method `addToTotal()`. Once this method terminates, the invariant of the composite on which `addComponent()` is invoked as well as the invariants of all its transitive parent composites will be re-established. However, for the duration of the recursive invocations of `addToTotal()`, those invariants are broken. Since these invocations (re-)enter inconsistent objects, method `addToTotal()` cannot assume that the invariant of its current receiver object holds in the initial state of the method.

The verification of the Composite pattern is challenging since it features a multi-object invariant and disallows an naive adoption of a visible states semantics. This restriction rules out, for instance, the Classical Technique [9] for verifying the Composite pattern since it can neither accommodate multi-object invariants nor re-entrant method invocations. A number of proposals to address the challenges of verifying the Composite pattern have been suggested. Verification techniques based on *ownership* [5, 7, 9] allow the specification and verification of the Composite pattern by leveraging the heap topology enforced by ownership types. However, an ownership-based specification of the Composite pattern prevents direct modifications of a composite's components. Other proposals typically employ a relaxed visible states semantics for invariants. Summers and Drossopoulou [12], for instance, introduce Considerate Reasoning, a verification technique that is based on a visible states semantics for invariants but allows distinguished invariants to be broken in the initial states of method executions provided that the methods re-establish the invariant in the final state. In addition, techniques have been presented that do not employ a visible states semantics for invariants. Bierhoff and Aldrich [28], for instance, leverage typestate-based permissions to verify a simpler invariant for binary Composite tree structures and Jacobs et al. [29] leverage separation logic to verify the SAVCBS invariant also for binary Composite tree structures.

In this paper, we show how first-class relationships allow for a precise specification of the Composite pattern. Our specification captures not only the SAVCBS invariant regarding a composite's `total` field but also gives a precise definition of a composite's tree properties. Using higher-level programming language abstractions and their stratification, we can encapsulate the multi-object invariant of the Composite pattern in a relationship and restore a visible state semantics for invariants.

3 First-Class Relationships

This section introduces the specification of the Composite pattern in Rumer and discusses the language features that are important for modular program verification. Rumer is a relationship-based programming language with Design-by-Contract-style [2] assertions. To gain practical experience with first-class relationships, we designed and implemented a prototype compiler that supports

the features shown in this paper and offers run-time checking of Design-by-Contract-style assertions. The Rumer compiler has been the basis for various student projects at ETH Zurich.

3.1 Language Principles

This section provides an overview of Rumer's basic language features. In the subsequent sections, we introduce each language feature in turn, based on the Rumer implementation of the Composite pattern shown in Fig. 2. The assertion language of Rumer is covered in Sect. 3.2.

Programmer-Definable Types. Figure 2 shows the implementation of the Composite pattern in Rumer. The program consists of three type declarations: the entity declaration `Component` and the relationship declarations `Parent` and `Composite`. An *entity* abstracts the state and behavior that a number of objects have in common. A *relationship* abstracts the state and behavior that a number of related objects have in common. Both language abstractions can be instantiated; we use the terms *entity instance* or *object* to denote instances of type entity and the term *relationship instance* to denote instances of type relationship. An entity resembles a class in a pure object-oriented language as it can define fields and methods. The existence of first-class relationships, however, fundamentally changes the position an entity takes in a relationship-based programming language. Using the abstraction of a relationship, a programmer can factor out the description of how objects relate into a relationship, rendering the need to establish references in an entity unnecessary. In Rumer, we build on this observation and prohibit the declaration of references in entities, requiring the declaration of how abstractions and their instances relate to happen exclusively in relationships. This language requirement results in a *stratification* of entities and relationships as only relationships know about their participating entities, but not vice versa. In Sect. 4 we discuss the benefits of the resulting stratification for program verification.

Simple Relationship Declaration. Relationships declare the types of instances they relate in their `participants` clause. For example, relationship `Parent` relates entity instances of type `Component` (line 3 in Fig. 2). Rumer allows programmers to associate an identifier with each type declaration in a participants clause to denote the *role* an instance of the type plays in the relationship. In case of relationship `Parent`, we use the role names `child` and `parent` to represent the hierarchical structure of the Composite pattern. Figure 3(a) provides a graphical illustration of a snapshot of a Rumer program heap comprising `Parent` relationship instances. The figure represents entity instances as dark gray circles and relationship instances as light gray ellipses. The heap snapshot consists of six `Component` entity instances and five `Parent` relationship instances. Each `Parent` relationship instance has a `Component` entity instance as `child` participant and a `Component` entity instance as `parent` participant. As indicated by the arrows in the figure, the role identifiers `child` and `parent` denote references from a `Parent` relationship instance to its participating `Component` objects.

```
1 entity Component {...}
2
3 relationship Parent participants (Component child, Component parent) {
4   int >parent total; // interposed instance field
5
6   extent void append(Component c, Component p) {
7     these.add(new Parent(c, p));
8     foreach (x isElementOf these.transitiveClosure().select(c_p:
9               c_p.child == c).parent)
10    { x.total = x.total + 1; }
11  }}
12
13 // A Composite relationship instance owns its tree Parent extent
14 relationship Composite participants (Component root, owned Extent<Parent>tree) {
15
16  extent void createComposite(Component c)
17  { these.add(new Composite(c, new owned Extent<Parent>())); }
18
19  void appendComponent(Component c, Component p)
20  { this.tree.append(c, p);}
21
22  void appendSubComposite(query Set<Parent> c, Component p) {
23    foreach (c_p isElementOf c.select(x: x.parent == p)) {
24      this.appendComponent(c_p.child, c_p.parent);
25      this.appendSubComposite(c.select(x: x.child isElementOf
26        c.transitiveClosure().select(y: y.parent == c_p.child).child),
27        c_p.child);
28    }}
29
30  void appendComposite(Composite c, Component p) {
31    this.appendComponent(c.root, p);
32    this.appendSubComposite(c.tree, c.root);
33  }}
```

Fig. 2. Relationship-based implementation of the Composite pattern in Rumer

Member Interposition. Relationships (like entities) can declare *instance members* (i.e., instance fields and instance methods). For example, relationship `Parent` declares an instance field `total` (line 4 in Fig. 2). Unlike entity instance members, relationship instance members can either be associated with the relationship instance or with one of the relationship's participant instances. The latter is achieved using the Rumer language mechanism *member interposition* [20, 25]. Member interposition allows a participant instance of a relationship instance to be "decorated" with additional fields and methods. Interposed relationship members are declared using the '>' sign, which precedes the role identifier of the participant into which the member is interposed. In the example, the field `total` is interposed into the `Component` entity instance that acts as a `parent` in a `Parent` relationship instance. The field `total` is conceptually equivalent to the `total` field of the SAVCBS 2008 challenge problem (line 3 in Fig. 1(b)) as it allows a parent component to store the number of all children components it is directly or indirectly related to. In Fig. 3(a), the interposed relationship instance field `total` is displayed in the light gray arc that is attached to the `Component` entity instance to which the relationship instance refers by the `parent` reference. A non-interposed relationship instance field, on the other hand, would be displayed in the light gray ellipse representing the relationship instance. Like non-interposed relationship instance fields, interposed relationship

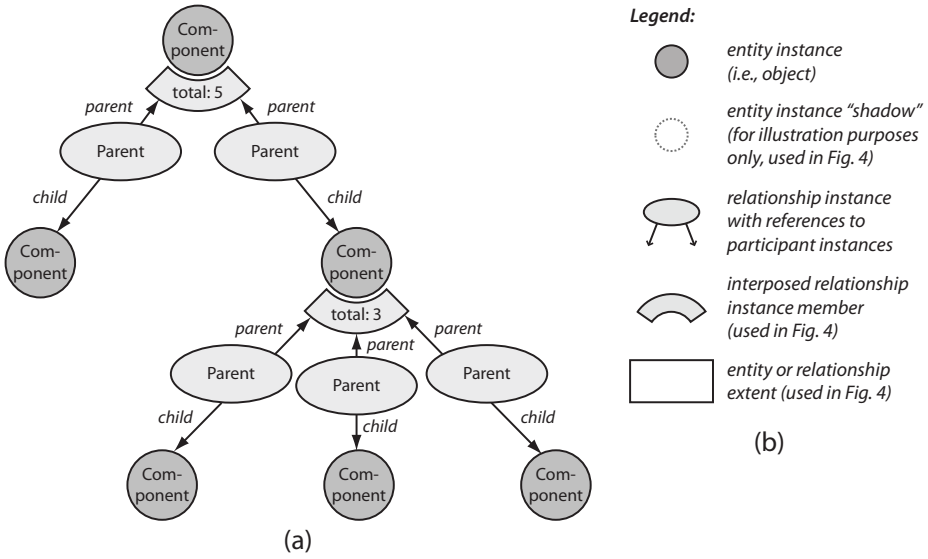


Fig. 3. (a) Schematic illustration of the Rumer program heap for the `Parent` relationship declared in Fig. 2. (b) Legend for Fig. 3(a) and Fig. 4.

instance fields are fully encapsulated in the relationship that declares the field. As a result, interposed relationship instance fields are only accessible from instances of the declaring relationship but not from the participant instances into which the fields are interposed.

Nested Relationship Declaration and Extent Ownership. The declaration of the relationship `Composite` illustrates that relationships can have other relationships as participants. A `Composite` relationship instance relates a `Component` entity instance to an extent of the `Parent` relationship. Every Rumer entity or relationship declaration T has a corresponding *extent* type Extent⟨T⟩. An instance of an entity or relationship extent comprises a set of entity or relationship instances, respectively. Extents are explicitly instantiated as well as populated and depopulated with instances by the programmer. For example, a `Parent` extent is instantiated on line 17 in Fig. 2 (`new owned Extent<Parent>()`) and populated by invoking the built-in `add()` method on lines 7 and 17 in Fig. 2. The keyword `owned` in the `participants` clause establishes ownership of a `Composite` instance of the participating `Parent` extent. The ownership declaration requires the `Parent` extent to be instantiated and populated within the relationship and not to escape the relationship. As a result, the `Composite` instance becomes the unique owner of its associated `Parent` extent.

Figure 4 shows an extended version of the Rumer heap snapshot shown in Fig. 3 and displays all instances of the types declared in Fig. 2. The figure represents extents by rectangular boxes. The heap snapshot consists of one `Composite` extent, two `Parent` extents, and one `Component` extent. Each

extent comprises a number of instances. For example, the `Composite` extent comprises two `Composite` relationship instances, and the `Component` extent comprises four `Component` entity instances. `Component` instances can participate in the `Composite` relationship as well as in the `Parent` relationship. To keep the graphical layout well-arranged, Fig. 4 uses "shadow" instances. A shadow instance is depicted by a dotted circle and is a graphical copy of an actual instance to which it is connected by a dotted line. Each `Composite` relationship instance relates a `Component` entity instance to a `Parent` extent. The former is referred to by the role identifier `root` and the latter by the role identifier `tree`. In a relationship-based implementation a Composite is thus represented by a tuple that consists of a `root` component and a set of hierarchically structured components that represent the `tree` rooted at the root component. The `tree` of a composite may denote the empty set (if the composite only consists of one (leaf) component). The dotted line between a composite's `root` component and the component at the top of the composite's `tree` `Parent` extent indicate that the two components are indeed the same. The two dotted lines converging in the third `Component` instance in the `Component` extent illustrate that `Parent` instances of different extents can share `Component` instances. The sharing of `Component` instances among different `Parent` extents does not compromise the ownership declared for `Composite`. The ownership only encompasses a `Parent` extent but not the participating `Component` entity instances. This property distinguishes extent ownership from "traditional" ownership established by ownership types [30, 31] and Universe Types [5, 32] and distinguishes the Rumer Composite implementation from one based on the ownership technique [5, 9].

Instance and Extent Members. To populate and depopulate their extents, entities and relationships declare extent methods. An *extent method* has an implicit target, which denotes the extent on which the method is called. The keyword `these` refers to the implicit target extent. Extent methods are distinguished by the `extent` keyword, which precedes the method's return type declaration. The Composite pattern implementation in Fig. 2 declares the relationship extent methods `append()` and `createComposite()`. The methods `appendComponent()`, `appendSubComposite()`, and `appendComposite()` are non-interposed relationship instance methods. In addition to extent methods, Rumer supports the declaration of extent fields (not used in Fig. 2). An extent field denotes the state of a whole extent, as opposed to an instance field, which denotes the state of an individual entity or relationship instance comprised in an extent.

Queries. Rumer provides *query expressions* (similar to LINQ heap queries [33]) to allow access to the instances contained in an extent. For example, method `append()` declares a query expression on line 8 (see Fig. 2) that makes use of the built-in query operators `transitiveClosure()` and `select()`. A query expression evaluates to a set that is constructed by invoking a query operator on a target extent or set. Whereas extents are explicitly instantiated and populated by programmers, sets can only be generated by querying extents or sets. In the example, the `transitiveClosure()` operator is invoked on the receiver extent of

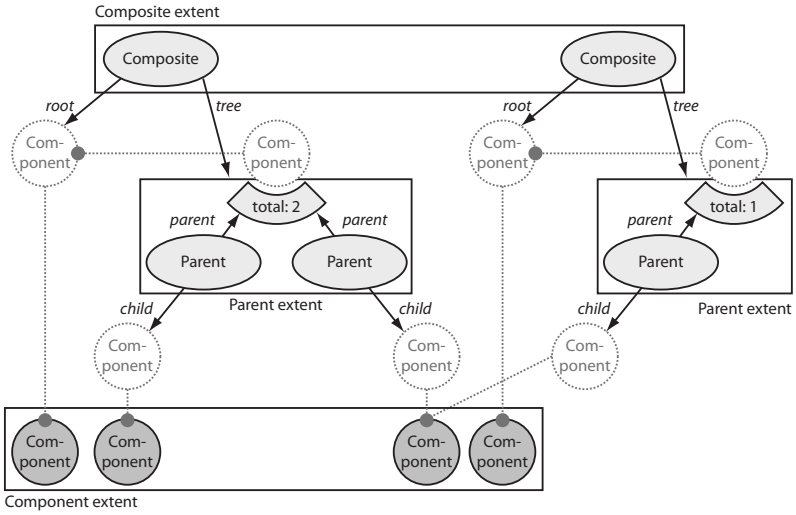


Fig. 4. Schematic illustration of the Rumer program heap for the implementation of the Composite pattern in Fig. 2 (see legend in Fig. 3(b))

the method `append()` and the `select()` operator is invoked on the set returned by the `transitiveClosure()` operator. The keyword `these` refers to the current receiver extent of an extent method invocation. The `transitiveClosure()` operator returns the transitive closure of its target set and the `select()` operator returns the subset of its target set that contains all the elements that satisfy the specified selection criterion. Like LINQ queries, the `select()` operator leverages lambda expressions to specify its selection criterion (i.e., `c_p: c_p.child == c`). As opposed to LINQ queries, Rumer queries are side-effect free. Side-effect freedom guarantees that the target sets of query operators are not altered in the course of the query evaluation and that query expressions become predicates over their target sets.

Implementation Details. Next we provide a brief overview of the individual method declarations in Fig. 2. These explanations are helpful to understand the details of the Composite pattern implementation in Rumer, however, are not a prerequisite to understanding the remainder of this paper. The impatient reader may continue with Sect. 3.2 and refer to this section as needed.

The extent method `append()` of relationship `Parent` appends the argument component `c` as child of the argument component `p`. To this end, the method instantiates a new `Parent` relationship instance with references to the components `c` and `p` and adds the new instance to the receiver extent of the method (line 7). The method `add()` is a language built-in method that adds the argument instance to the extent on which the method is invoked. The loop on line 8 increments the `total` field of all transitive parent components of the child component `c`. The loop header declaration uses built-in query operators, as discussed in the previous section.

The extent method `createComposite()` of relationship `Composite` instantiates a new `Composite` relationship instance and adds it to the current receiver extent of the method. The new instance has the component `c` as a `root` participant and an empty `Parent` extent as a `tree` participant.

The instance method `appendComponent()` of relationship `Composite` invokes the extent method `append()` with the argument components `c` and `p` on the current receiver relationship instance's `tree` extent. As a result, the component `c` is appended as a child of the component `p` in the composite's `tree` extent.

The instance method `appendSubComposite()` of relationship `Composite` appends the sub-composite denoted by the query expression `c` to the target composite as a child of component `p`. The method is implemented recursively to append the sub-composite in a depth-first traversal fashion. In each recursive invocation, one child component of the sub-composite is appended to its corresponding parent component in the target composite's tree extent. Recursion stops whenever the sub-composite `c` denotes the empty set. This is the case whenever a leaf component has been inserted in the preceding recursive invocation.

The instance method `appendComposite()` of relationship `Composite` appends the composite `c` to the target composite as a child of component `p`. The method first appends the `root` component of composite `c` to the target composite as a child of `p` and then invokes the method `appendSubComposite()` on the target composite to append the sub-composite rooted at `c`'s `root` component as a child of the previously inserted `root` component.

3.2 Assertion Language

The Rumer assertion language includes method *preconditions* and *postconditions*, *assert* statements, and *invariants*. Assertions can range over all abstractions available in the Rumer programming language. Invariants, in particular, can be declared both for type instances and type extents, giving rise to the following four invariant categories[1]:

- **Entity instance invariant:** Property that must hold for each entity instance of the entity that declares the invariant.
- **Entity extent invariant:** Property that must hold for each extent instance of the entity that declares the invariant.
- **Relationship instance invariant:** Property that must hold for each relationship instance of the relationship that declares the invariant.
- **Relationship extent invariant:** Property that must hold for each extent instance of the relationship that declares the invariant.

Figure 5 lists the invariant declarations for the Composite pattern implementation in Fig. 2. The declaration consists of a relationship extent invariant for relationship `Parent` (line 3) and a relationship instance invariant for relationship `Composite` (line 12). Extent invariants are distinguished from instance invariants by the keyword `extent`. All invariant declarations in Fig. 5 adhere to the admissibility criteria defined in Sect. 4.2.

[1] These invariant categories refine the categories introduced in earlier work [20]

```
1 relationship Parent participants (Component child, Component parent) {
2   ... // See Fig. 2
3   extent invariant
4     these.isPartialFunction() &&
5     these.transitiveClosure().isIrreflexive() &&
6     forAll(p isElementOf these.parent: p.total ==
7       these.transitiveClosure().select(c_p: c_p.parent == p).count());
8 }
9
10 relationship Composite participants (Component root, owned Extent<Parent>tree) {
11   ... // See Fig. 2
12   invariant
13     !(this.root isElementOf this.tree.child) &&
14     (!this.tree.isEmpty() => this.root isElementOf this.tree.parent) &&
15     this.tree.child == this.tree.transitiveClosure().select(c_p: c_p.parent ==
16        this.root).child;
17 }
```

Fig. 5. Invariant declarations for the Composite program in Fig. 2. See method preconditions and postconditions in Appendix A.

The extent invariant of relationship `Parent` leverages Rumer queries (see Sect. 3.1) and guarantees the following properties: *(i)* that every child component is related to at most one parent component (line 4), *(ii)* that the graph described by the `Parent` relationship is acyclic (line 5), and *(iii)* that the value of a parent component's `total` field is equal to the number of children components to which the parent component is transitively related (line 6). Property *(iii)* satisfies the invariant of the SAVCBS 2008 challenge problem regarding a composite's `total` field. Properties *(i)* and *(ii)* guarantee that the graph described by a `Parent` extent forms a forest of trees.

The instance invariant of relationship `Composite` restricts a composite's `Parent` extent from a forest of trees to a tree by enforcing the following properties: *(i)* that a composite's root component never appears as a child component in the graph described by the `Parent` extent (line 13), *(ii)* that a composite's root component appears as a parent component in the graph described by the `Parent` extent unless the graph is empty (line 14), and *(iii)* that a composite's root component is the parent component of all children components of the graph described by the `Parent` extent (line 15). These properties guarantee that a composite has a unique root and that a composite's root component is the same as the one at the top of a composite's tree. The heap snapshot shown in Fig. 4 represents a valid instantiation of the Composite pattern specification of Fig. 2 and Fig. 5. The shown `Composite` instances have unique roots and form trees. Note that the fact that different composites may share components (as indicated by the third shadow component in the `Component` extent) does not compromise the extent invariant of `Parent`. An extent invariant must hold for each extent instance but not for the union of all extent instances.

To guarantee the invariants declared in Fig. 5, the methods of the Composite pattern implementation (see Fig. 2) define preconditions and postconditions. The complete list of preconditions and postconditions for all methods is given in Appendix A. In the following, we highlight those preconditions and postconditions that are particularly interesting.

By appending the argument component c as child of the argument component p, method append() of relationship Parent may compromise the extent invariant of Parent. To prevent introducing cycles and relating a child component to several parent components, the method establishes the following precondition:

```
c != p && !(c isElementOf these.child union these.parent)
```

Furthermore, the method updates the total field of all transitive parent components of the child component c and thus ensures the following postcondition:

```
forAll(x isElementOf these.transitiveClosure().select(c_p: c_p.child == c).parent:
  x.total == old(x.total) + 1)
```

Method appendComposite() of relationship Composite appends the argument composite c to the target composite as a child of the argument component p. To prevent introducing cycles in the altered target composite, the method establishes the following precondition:

```
p != c.root && !(p isElementOf c.tree.child union c.tree.parent) &&
(this.tree.isEmpty() => p == this.root) &&
(!this.tree.isEmpty() => p isElementOf this.tree.child union this.tree.parent) &&
!(c.root isElementOf this.tree.child union this.tree.parent) &&
((this.tree.child union this.tree.parent) intersection
(c.tree.child union c.tree.parent)).isEmpty();
```

The above precondition would prevent us from appending the left composite instance of Fig. 4 to the right composite instance, or vice versa, since the last conjunct of the precondition could not be satisfied. The instance method appendComposite() invokes the instance method appendSubComposite() of relationship Composite for the actual insertion of the sub-tree rooted at c's root component. To guarantee that the sub-tree c passed as argument indeed forms a tree with the root component p, method appendSubComposite() establishes the following precondition:

```
!(p isElementOf c.child) &&
(!c.isEmpty() => p isElementOf c.parent) &&
(c.child == c.transitiveClosure().select(c_p: c_p.parent == p).child) &&
```

The above precondition is equivalent to a composite's invariant. The postcondition of appendSubComposite()

```
this.tree == old(this.tree) union c;
```

precisely captures the "invariant" of the recursive implementation of the method that inserts the sub-tree c in a depth-first traversal fashion.

4 The Matryoshka Principle

The modularization discipline embodied in the Rumer programming language follows the *"Matryoshka Principle"*. We use the metaphor of the Russian nesting dolls to refer to the inherent *stratification* of programming language abstractions in Rumer. Based on this stratification, the principle defines admissibility criteria that stipulate restrictions on writes to locations, invariant declarations, and method invocations. We first introduce the stratification of the programming language abstractions. Then, we discuss the admissibility criteria.

4.1 Stratification

Figure 6 provides a schematic illustration of how programming language abstractions are stratified in Rumer. The figure shows an extended version of the Composite heap snapshot shown in Fig. 4 and depicts each programmer-defined type of the Composite program in addition to the extents and entity and relationship instances shown in Fig. 4. The stratification of the Rumer language abstractions is determined by the `participants` clauses of relationship declarations. Since only relationships can refer to their participants, but the participants not to their relationship, the `participants` clauses give rise to a *strict partial order* between relationships and participants. Figure 6 represents the resulting ordering of relationships and participants by placing relationships (relatively) above their participants. As indicated by the vertical arrows, the ordering makes a relationship become an *upper* layer of its participants, and conversely, the participants become a *lower* layer of their relationships.

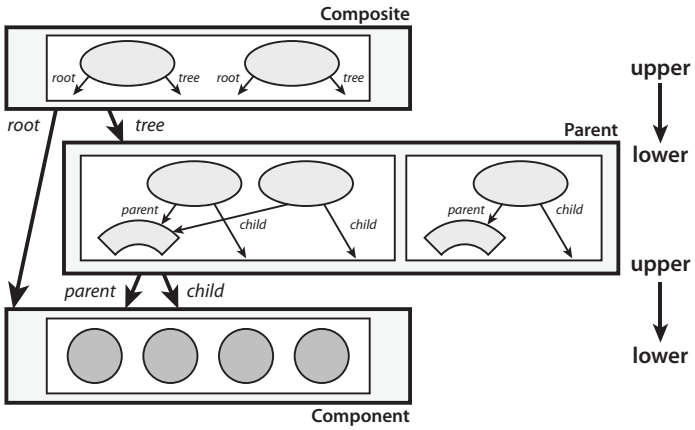


Fig. 6. Schematic illustration of the stratification of Rumer language abstractions (based on Fig. 4)

4.2 Admissibility Criteria

The admissibility criteria rely on the stratification of language abstractions shown in Fig. 6. The criteria stipulate restrictions that define *(i)* which methods are allowed to write to which locations, *(ii)* which invariants are allowed to depend on which locations, and *(iii)* on which instances a currently executing method can invoke another method.

In Rumer, a *location* can be an instance field, an extent field, or a whole extent. Table 1 lists all possible Rumer locations and indicates for each location by what program statement it can be written to. The admissibility criteria are formulated relative to the set of locations that can be reached by an instance. This set of locations is determined by the `participants` clauses of relationship

declarations which indicate which other instances a particular instance may reach. The general direction of "reachability" between instances conforms to the ordering of relationships and participants indicated by the arrows in Fig. 6. This ordering allows a relationship extent to reach itself and any of its participating extents and allows a relationship instance to reach itself and any of its participating instances. Orthogonal to the direction of reachability determined by `participants` clauses, an extent can reach any of the instances it comprises by formulating an appropriate query expression.

Table 1. Rumer locations and program statements that can write to them

Location			Write
Entity		**instance field**	Assignment to field.
		extent field	Assignment to field.
		extent	Invocation of `add()` or `remove()` on extent.
Relationship	**interposed**	**instance field**	Assignment to field.
	non-interposed	**instance field**	Assignment to field.
		extent field	Assignment to field.
		extent	Invocation of `add()` or `remove()` on extent.

Admissible Writes. To allow for modular verification, the Matryoshka Principle requires that a method writes only to a location that is reachable from the current receiver and that is declared by the same type as the method. This requirement guarantees that all Rumer locations are encapsulated in the types that declare the locations.

The assignment to the interposed instance field `total` of relationship `Parent` on line 10 in Fig. 2, for example, is admissible since `x` refers to a `parent` component of a `Parent` relationship instance residing in `these` and since the assignment occurs in a method declared by the same relationship (i.e., `Parent`) as the field `total`. The admissibility of the assignment also relies on the fact that interposed instance fields are treated as fields of the relationship instance even though they describe properties of relationship participants. In previous work [25], we have shown how member interposition facilitates modular reasoning over shared state at the example of the Observer pattern. The invocations of the built-in method `add()` on line 7 and line 17 in Fig. 2 represent admissible writes as well since they write to the current receiver extent and since they occur in extent methods of the types that declare the current receiver extent.

Admissible Invariants. To allow for modular verification, the Matryoshka Principle requires that an invariant depends only on those locations *(a)* that are encapsulated in the type that declares the invariant, or alternatively, on those locations *(b)* that are declared by a type that is owned by the type that declares the invariant.

Both invariant declarations in Fig. 5 are admissible. The relationship extent invariant of `Parent` depends on the current receiver `Parent` extent as well as on `Parent`'s interposed relationship instance field `total`. These locations are

encapsulated by the relationship `Parent`. The relationship instance invariant of `Composite` is admissible due to a `Composite`'s instance ownership of its `Parent` extent.

Admissible Method Invocations. To allow for a visible states semantics and inductive reasoning, a verification technique must either be guaranteed that call-backs do not occur or be in the position to statically identify those invocations that may result in a call-back. Prohibiting call-backs in general is not feasible since it would also prevent direct call-backs. A *direct call-back* occurs if an executing method invokes a method on the same receiver instance as the one of the executing method. Recursive method invocations are special instances of direct call-backs. Prohibiting recursive method invocations would be too limiting a restriction. Moreover, direct call-backs can be statically identified and guarded with the proof obligation to re-establish the invariant of the current receiver instance before the call. Transitive call-backs, on the other hand, cannot be statically determined but can only be over-approximated. A *transitive call-back* occurs if an executing method invokes a method on a different receiver instance as the one of the executing method and if the invoked method or any of the methods it transitively invokes calls back into the original receiver instance. In a pure object-oriented setting, call-backs are essential for re-establishing a multi-object invariant. In a relationship-based language, however, multi-object invariants can be expressed at the right level of abstraction, relieving the need of a call-back to re-establish a multi-object invariant.

To guarantee a visible states semantics for invariants, the Matryoshka Principle requires that a currently executing method can only invoke a method *(a)* on the same receiver instance as the currently executing method or *(b)* on a receiver instance that is of a lower type than the declaring type of the currently executing method. These requirements guarantee that method invocations are either recursive, propagate downwards, or are dispatched over an instance contained in an extent. As a result, transitive call-backs are prevented and a visible state semantics for invariants can be maintained.

All method invocations occurring in the Composite pattern implementation in Fig. 2 are admissible since they either constitute direct call-backs or invocations on a receiver instance that resides in a lower stratification layer or in the extent of the calling method.

5 Verification Technique

This section introduces the visible states verification technique for Rumer. Section 5.1 briefly introduces the unified framework for visible states verification techniques [34], which we use for presenting our technique. Section 5.2 details the proof obligations of our technique. We proved our verification technique to be sound in [35].

5.1 Background

To describe our verification technique, we use the unified framework[2] for visible states verification techniques introduced by Drossopoulou et al. [34]. The framework captures a verification technique in terms of seven *parameters*. These parameters are:

- $\mathbb{X}$ invariants expected to hold in initial and final states of a method execution (i.e., visible states).
- $\mathbb{V}$ invariants vulnerable to a method execution, i.e., which may be broken while the method executes.
- $\mathbb{B}$ invariants that must be proven to hold before a method call.
- $\mathbb{E}$ invariants that must be proven to hold in the final state (i.e., at the end) of a method execution.
- $\mathbb{U}$ permitted receivers of field updates.
- $\mathbb{D}$ invariants that may depend on a given heap location (and indirectly locations on which an invariant may depend).
- $\mathbb{C}$ permitted receivers of method calls.

Figure 7 illustrates the meaning of the framework parameters for the method `appendComponent()` on line 19 in Fig. 2. As demonstrated by the figure, $\mathbb{X}$ can be assumed to hold in the initial and final states of method `appendComponent()`. In between these visible states only $\mathbb{X} \setminus \mathbb{V}$ can be assumed to hold since some invariants may be temporarily broken by the execution of the method. For field updates and method calls, the receiver instances must be checked to be in $\mathbb{U}$ and $\mathbb{C}$, respectively. For example, before the invocation of method `append()` on the receiver `this.tree` it must be checked that the instance referred to by `this.tree` is in $\mathbb{C}$. In the pre-state of a method call (i.e., `append()`), $\mathbb{B}$ must be proven, and in the final state of the method execution (i.e., `appendComponent()`), $\mathbb{E}$ must be proven. For assignments (not shown in Fig. 7), lastly, it must be checked that at most the invariants in $\mathbb{D}$ are influenced.

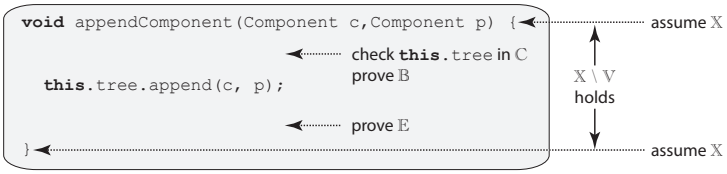


Fig. 7. Illustration of the verification technique framework parameters (based on [34])

[2] The framework has been introduced to capture visible states verification techniques for object invariants and to prove their soundness. Like Summers et al. [36], we use the framework for illustration purposes only.

5.2 Proof Obligations

Our verification technique for Rumer is a *visible states verification technique* [9]. It supports the complete assertion language discussed in Sect. 3.2. The technique is *modular* and allows entities and relationships to be verified independently from each other.

In this section, we describe our verification technique in terms of the seven parameters of the unified framework introduced previously. Since the parameters $\mathbb{U}$, $\mathbb{D}$, and $\mathbb{C}$ are defined by the admissibility criteria introduced in Sect. 4.2, this section defines only the remaining parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$. We refer to the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ as the *invariant parameters* since they specify sets of invariants, and to the parameters $\mathbb{U}$, $\mathbb{D}$, and $\mathbb{C}$ as the *admissibility parameters* since they are captured by the admissibility criteria of the Matryoshka Principle. The verification technique determines the invariant parameters for all kinds of programmer-definable methods (i.e., entity instance method, entity extent method, interposed relationship instance method, non-interposed relationship instance method, and relationship extent method) as well as for constructors and the non-pure built-in methods `add()` and `remove()` (see Sect. 3.1). Built-in query operators (see Sect. 3.1) do not need to be considered by the verification technique since they are side-effect free.

Table 2 and Table 3 specify the invariant parameters for programmer-definable methods. Although there are variations between the invariant parameters for the different kinds of methods, there is a general schema that can be observed: The invariant parameters $\mathbb{V}$ and $\mathbb{X}$ are determined by the admissibility parameters $\mathbb{U}$ ("Admissible Writes") and $\mathbb{D}$ ("Admissible Invariants") and by the stratification of the programming language abstractions, respectively. The admissibility parameter $\mathbb{U}$ guarantees that only the locations of the current receiver of a method can be written to. This admissibility parameter guarantees in turn that, while a method executes, at most the locations to which the method is allowed to write can change. The set of "vulnerable" locations indirectly determines the set of invariants $\mathbb{V}$ that are vulnerable to a method: it is the set of invariants that may depend on those locations that may change during the execution of a method. The set of locations that an invariant may depend on is determined by the admissibility parameter $\mathbb{D}$. This set of locations is different for an invariant that is based on ownership compared to an invariant that is not based on ownership. In case of an ownership-based invariant, the set of locations that an invariant is allowed to depend on includes the locations of those lower-level types for which ownership is declared. In case of an invariant that is not based on ownership, the set of locations that an invariant is allowed to depend on includes only the locations of the invariant declaring type. The set of invariants $\mathbb{X}$ that are expected to hold in the visible states of a method, on the other hand, is determined by the parameter $\mathbb{V}$ and the stratification of the Rumer programming language. Irrespective of whether ownership is declared, a method can always expect all invariants of lower-level types to hold in its visible states. If the declaring type

Table 2. Invariant parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for entity instance method, entity extent methods, interposed relationship instance methods, and relationship extent methods

Method	Parameter
Entity instance	$\mathbb{X}$: Entity instance invariant of current receiver. Invariants of all types for which the entity is not a transitive participant.
	$\mathbb{V}$: Entity instance invariant of current receiver. Entity extent invariant of current receiver's extent. Ownership-based invariants of upper-level types.
	$\mathbb{B}$: Entity instance invariant of current receiver if direct call-back.
	$\mathbb{E}$: Entity instance invariant of current receiver. Preservation of entity extent invariant of current receiver's extent.
Entity extent	$\mathbb{X}$: All instance and extent invariants of the entity. Invariants of all types for which the entity is not a transitive participant.
	$\mathbb{V}$: Entity instance invariants of all instances in current receiver extent. Entity extent invariant of current receiver extent. Ownership-based invariants of upper-level types.
	$\mathbb{B}$: Entity instance invariants of all instances in current receiver extent and entity extent invariant of current receiver extent if direct call-back. Entity instance invariant of callee if entity instance method is called.
	$\mathbb{E}$: Entity instance invariants of all instances in current receiver extent. Entity extent invariant of current receiver extent.
Interposed relationship instance	$\mathbb{X}$: Invariants of all types for which the relationship is not a transitive participant.
	$\mathbb{V}$: Relationship instance invariants of all relationship instances that have current receiver as participant. Relationship extent invariant of current receiver's extent. Ownership-based invariants of upper-level types.
	$\mathbb{B}$: -
	$\mathbb{E}$: Preservation of relationship instance invariants of all relationship instances that have current receiver as participant. Preservation of relationship extent invariant of current receiver's extent.
Relationship extent	$\mathbb{X}$: All instance and extent invariants of the relationship. Invariants of all types for which the relationship is not a transitive participant.
	$\mathbb{V}$: Relationship instance invariants of all instances in current receiver extent. Relationship extent invariant of current receiver extent. Ownership-based invariants of upper-level types.
	$\mathbb{B}$: Relationship instance invariants of all instances in current receiver extent and relationship extent invariant of current receiver extent if direct call-back. Relationship instance invariant of callee if relationship instance method is called.
	$\mathbb{E}$: Relationship instance invariants of all instances in current receiver extent. Relationship extent invariant of current receiver extent.

of a method is owned by an upper-level type, the execution of the method may compromise any ownership-based invariant of the owning type. However, if the declaring type of a method is not owned by an upper-level type, the execution of the method can only compromise invariants of its declaring type.

We illustrate the verification technique based on the Rumer implementation of the Composite pattern (see Fig. 2 and Fig. 5). Method `append()` of relationship `Parent` is a relationship extent method. According to Table 2, the parameter $\mathbb{V}$ for a relationship extent method comprises the relationship instance invariants of all relationship instances in the current receiver extent as well as the relationship extent invariant of the current receiver extent. Since relationship `Parent` only declares an extent invariant, the parameter $\mathbb{V}$ for method `append()` comprises the relationship extent invariant declared on line 3 in Fig. 5. This is also the invariant that the method must prove to hold in the final state of the method

Table 3. Invariant parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for non-interposed relationship instance methods

Invariant	Parameter
Benign	$\mathbb{X}$: Relationship instance invariant of current receiver. Invariants of all types for which the relationship is not a transitive participant.
	$\mathbb{V}$: Relationship instance invariant of current receiver. Relationship extent invariant of current receiver's extent. Ownership-based invariants of upper-level types.
	$\mathbb{B}$: Relationship instance invariant of current receiver if direct call-back.
	$\mathbb{E}$: Relationship instance invariant of current receiver. Preservation of relationship extent invariant of current receiver's extent.
Malign	$\mathbb{X}$: Relationship instance invariant of current receiver. Invariants of all types for which the relationship is not a transitive participant.
	$\mathbb{V}$: Relationship instance invariants of all relationship instances that have current receiver's participant(s) as participant(s). Relationship extent invariant of current receiver's extent. Ownership-based invariants of upper-level types.
	$\mathbb{B}$: Relationship instance invariant of current receiver if direct call-back.
	$\mathbb{E}$: Relationship instance invariant of current receiver. Preservation of relationship instance invariants of all relationship instances that have current receiver's participant(s) as participant(s). Preservation of relationship extent invariant of current receiver's extent.

(parameter $\mathbb{E}$). Given the preconditions of the method (see Appendix A) and the fact that the method updates the `total` field appropriately, the method is able to prove the invariant. To sustain a visible states semantics, Table 2 further requires that any method invocations on the current receiver extent are guarded with the proof obligation to re-establish the relationship extent invariant before the invocation. Method `append()` invokes the built-in `add()` method on line 7. The built-in methods `add()` and `remove()` (not shown in Tables 2 and 3) are treated differently than user-defined methods. Since no call-backs can result from built-in methods, callers do not have to re-establish their invariants before invoking a built-in method. As a result, method `append()` can invoke method `add()` without re-establishing its relationship extent invariant. As indicated by Table 2, method `append()` can expect the following invariants to hold in its visible states (parameter $\mathbb{X}$): all instance invariants and extent invariants of `Component` instances and `Component` extents, respectively, as well as all instance invariants and extent invariants of `Parent` instances and `Parent` extents, respectively. However, `append()` cannot expect the invariants of its upper-level type `Composite` to hold in its visible states.

Relationship `Composite` declares an extent method as well as non-interposed instance methods. The reasoning regarding the verification of the extent method is analogous to the one employed for method `append()` of relationship `Parent`. We highlight the important aspects of verifying non-interposed relationship instance methods. The invariant parameters for non-interposed relationship instance methods are defined in Table 3. The table distinguishes two kinds of non-interposed relationship instance methods: *malign* versus *bening*. The differentiation is due to the occurrence of interposed relationship instance fields in a relationship instance invariant declaration. A relationship instance invariant may relate interposed fields to non-interposed fields or relate interposed fields of different participants (category "malign" in Table 3). Alternatively, a relationship

instance invariant may only depend on non-interposed fields (category "benign" in Table 3). The relationship instance invariant of relationship `Composite` is a benign invariant since there are no interposed fields declared by the relationship. The set of vulnerable invariants $\mathbb{V}$ for an non-interposed relationship instance method of `Composite` consists only of `Composite`' instance invariant. If `Composite` declared an extent invariant, that invariant would be vulnerable as well as the invariant may depend on instance fields. In its final state, a non-interposed relationship instance method must prove that the instance invariant of its current receiver holds and that it preserves the relationship extent invariant of its current receiver's extent. The proof obligation of "invariant preservation" has been introduced in the context of object and class invariants in [11] and [36], respectively. It represents a weaker proof obligation as it does not require a method to *assert* that an invariant holds but to show that it does *not break* the invariant (provided that it held initially). Both for entity instance methods and relationship instance methods our verification technique requires the method to prove preservation of the extent invariant. This proof obligation accounts for the fact that an instance method may break an extent invariant, but may not be in the position to re-establish that invariant. If an instance method cannot show to preserve an extent invariant, its corresponding code must be captured in an extent method.

The invariant parameters for malign non-interposed relationship instance methods account for the fact that modifications of interposed relationship instance fields may compromise not only the invariant of the current receiver instance but also the invariants of all those relationship instances that have participants in common with the current receiver instance. This can be the case whenever an invariant relates an interposed relationship instance field with an interposed relationship instance field of another participant or relates an interposed relationship instance field with a non-interposed relationship instance field. The invariant parameters for such malign invariants are slightly different. Most importantly, their set of vulnerable invariants $\mathbb{V}$ contains also the invariants of all the relationship instances that have participants in common with the current receiver instance. These invariants are also the ones that must be shown to be preserved in the final state of the non-interposed relationship instance method.

6 Discussion and Related Work

Our work builds on the visible states verification techniques developed for object invariants [5, 9, 11, 12] and introduces a verification technique for a relationship-based language that supports invariants for entities and relationships both at the instance and the extent level. The ownership technique [5, 9] is the visible states verification technique for object invariants that is most closely related to our work. Our verification technique resembles the ownership technique in two aspects: *(i)* it leverages heap stratification to prevent transitive call-backs and *(ii)* it facilitates modular reasoning about multi-object invariants. However, whereas the ownership technique is only composed of a single "ingredient" (i.e., Universe

Types [5, 32]), our verification technique unifies three orthogonal "ingredients" that can be combined in multiple ways. This customization of ingredients allows a programmer to "trade" imposed restrictions and supported guarantees.

The basic ingredient of our verification technique is the Matryoshka Principle. It enforces a stratification of language abstractions and guarantees the absence of transitive call-backs. As opposed to the ownership technique, the absence of transitive call-backs does not come at the price of a single ownership restriction! In a Rumer program conforming to the Matryoshka Principle, a participant of a relationship may be a participant of several relationships.

The Matryoshka Principle can be overlaid with member interposition to facilitate the modular verification of multi-object invariants. Similarly to the ownership technique, member interposition mitigates the adverse affect of aliases to shared state, but in a less restrictive way. As opposed to the ownership technique, member interposition does not prevent an instance from participating in other relationships but only prevents the interposed field from being accessible outside the relationship. Member interposition entails furthermore a slightly different semantics in terms of proof obligations than the ownership technique: Whereas ownership allows the declaration of invariant-compromising methods in owned objects as long as the owner does not invoke these methods, member interposition prevents the declaration of a method in a relationship that compromises the relationship invariant.

The Matryoshka Principle can also be overlaid with extent ownership to facilitate the modular verification of multi-object invariants. Extent ownership allows an owning type to impose an invariant on the owned extent. Similarly to the ownership technique, extent ownership enforces single ownership of the owned extent. However, as opposed to the ownership technique, extent ownership only encompasses an extent but not any (transitive) participant instances. As a result, those (transitive) participant instances can be modified by an arbitrary instance, including the extent owner. Extent ownership is thus more "lightweight" than "traditional" ownership since it relies on the Matryoshka Principle to prevent transitive call-backs.

The Matryoshka Principle can finally be overlaid both with member interposition and extent ownership. This setup was chosen for the Composite pattern. In the Rumer implementation of the Composite pattern, member interposition facilitates the verification of the `Parent` invariant, which includes the SAVCBS 2008 challenge problem invariant regarding a composite's `total` field. The implementation leverages extent ownership, on the other hand, to verify the `Composite` invariant, which imposes a tree structure on a composite's `parent` extent.

A number of techniques address the issue of object-oriented program verification in the presence of shared mutable state by leveraging heap partitioning. Parkinson and Bierman [15, 37, 38] introduce the ideas of separation logic [39] to Java. An alternative expression of separation is used in works on dynamic frames [40–42] where pure methods or ghost fields denote a set of locations. Assertions on the disjointness of such dynamic frames then facilitates heap-local reasoning. Parkinson's and Bierman's abstract predicates bear resemblance with

the invariants of our work. Similarly to a relationship invariant, an abstract predicate imposes consistency conditions on the object structures in the heap. However, abstract predicates do not entail an invariant semantics. This offers some flexibility to the programmer who does not need to adhere to a discipline, but sacrifices data type induction.

More distantly related is also the work on relationship-based programming languages [16–24]. For a summary of other approaches to relationship-based programming we refer to an earlier paper [20]. However, Rumer differs importantly from other relationship-based programming languages by its inherent stratification and by its support for Design-by-Contract-style assertions and invariants.

7 Conclusions

The verification of object-oriented programs remains a research issue. In this paper we discuss how relationships facilitate the verification of programs with multi-object invariants. Relationships impose a stratification of programming abstractions and consequently allow for local reasoning about multi-object invariants so that a modular verification of multi-object invariants is possible. The key concepts that allow for such local reasoning are *(i)* "member interposition" — properties (or fields) of a relationship participant that belong logically to the participant yet are encapsulated in the relationship instance, *(ii)* "extent ownership" — lightweight ownership of a relationship instance of its participant extent, and *(iii)* the Matryoshka Principle.

The"Matryoshka Principle" exploits the stratification layers of a program's abstractions and defines admissibility criteria for writes to locations, invariant declarations, and method invocations. Programs that obey this principle can be verified using the simple approach outlined here. The programming language Rumer, which we use for illustration in this paper, adheres to this principle by design. However, the principle is not tied to this particular programming language. Programs in other programming languages can incorporate the principle as well (and could then be verified using this approach), but the responsibility to make the program obey the principle would fall either upon a programmer or some program development tool.

This paper reports on the benefits of including relationships in a programming language for program verification — as interest in tools and techniques to verify programs increases, we expect the idea of "relationships" as a way to express the interplay between objects to deserve serious consideration in mainstream programming languages.

Acknowledgments. We are grateful to: Sophia Drossopoulou for discussions on relationships and program verification; Gavin Bierman and Matthew Parkinson for discussions on the "relationship" between relationships and abstract predicates; Alexander J. Summers for exchange on the semantics of invariant preservation; James Noble for discussions on the "relationship" between extent ownership and "traditional" ownership; Reto Conconi, Nicholas D. Matsakis, and Albert Noll for their feedback; and the anonymous reviewers for their valuable comments.

References

1. Hoare, C.: Proof of correctness of data representations. Acta Inf. 1(4), 271–281 (1972)
2. Meyer, B.: Object-Oriented Software Construction. Prentice-Hall, Englewood Cliffs (1997)
3. Barnett, M., M. Leino, K.R., Schulte, W.: The spec# programming system: An overview. In: Barthe, G., Burdy, L., Huisman, M., Lanet, J.-L., Muntean, T. (eds.) CASSIS 2004. LNCS, vol. 3362, pp. 49–69. Springer, Heidelberg (2005)
4. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev29, Iowa State University (2006)
5. Müller, P.: Modular Specification and Verification of Object-Oriented Programs. LNCS, vol. 2262. Springer, Heidelberg (2002)
6. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Leino, K 3(6), 27–56 (2004)
7. Leino, K.R. M., Müller, P.: Object invariants in dynamic contexts. In: Odersky, M. (ed.) ECOOP 2004. LNCS, vol. 3086, pp. 491–515. Springer, Heidelberg (2004)
8. Barnett, M., Naumann, J.D.A.: Friends need a bit more: Maintaining invariants over shared state. In: Kozen, D. (ed.) MPC 2004. LNCS, vol. 3125, pp. 54–84. Springer, Heidelberg (2004)
9. Müller, P., Poetzsch-He_ter, A., Leavens, G.T.: Modular invariants for layered object structures. Sci. Comput. Program 62(3), 253–286 (2006)
10. Leino, K.R.M., Schulte, W.: Using history invariants to verify observers. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 80–94. Springer, Heidelberg (2007)
11. Middelkoop, R., Huizing, C., Kuiper, R., Luit, E.J.: Invariants for non-hierarchical object structures. Electr. Notes Theor. Comput. Sci. 195, 211–229 (2008)
12. Summers, A.J., Drossopoulou, S.: Considerate Reasoning and the Composite Design Pattern. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 328–344. Springer, Heidelberg (2010)
13. Spitzen, J.M., Wegbreit, B.: The verification and synthesis of data structures. Acta Inf. 4(2), 27–144 (1975)
14. Guttag, J.V.: Notes on type abstraction (version 2). IEEE Trans. Software Eng. 6(1), 13–23 (1980)
15. Parkinson, M.J.: Class invariants: The end of the road? In: IWACO (2007)
16. Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. In: OOPSLA, vol. 481, pp. 466–481. ACM, New York (1987)
17. Albano, A., Ghelli, G., Orsini, R.: A relationship mechanism for a strongly typed object-oriented database programming language. In: VLDB, pp. 565–575. Morgan Kaufmann, San Francisco (1991)
18. Bierman, G.M., Wren, A.: First-class relationships in an object-oriented language. In: Black, A.P. (ed.) ECOOP 2005. LNCS, vol. 3586, pp. 262–286. Springer, Heidelberg (2005)
19. Pearce, D.J., Noble, J.: Relationship aspects. In: AOSD, pp. 75–86. ACM, New York (2006)
20. Balzer, S., Gross, T.R., Eugster, P.T.: A relational model of object collaborations and its use in reasoning about relationships. In: Ernst, E. (ed.) ECOOP 2007. LNCS, vol. 4609, pp. 323–346. Springer, Heidelberg (2007)

21. Wren, A.: Relationships for Object-oriented Programming Languages. PhD thesis, University of Cambridge (November 2007)
22. Østerbye, K.: Design of a class library for association relationships. In: LCSD (2007)
23. Bodden, E., Shaikh, R., Hendren, L.: Relational aspects as tracematches. In: AOSD, pp. 84–95. ACM, New York (2008)
24. Nelson, S., Pearce, D.J., Noble, J.: First class relationships for OO languages. In: MPOOL (2008)
25. Balzer, S., Gross, T.R.: Modular reasoning about invariants over shared state with interposed data members. In: PLPV, pp. 49–56. ACM, New York (2010)
26. Leavens, G.T., Leino, K.R.M., Müller, P.: Specification and verification challenges for sequential object-oriented programs. Formal Asp. Comput. 19(2), 159–189 (2007)
27. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
28. Bierhoff, K., Aldrich, J.: Permissions to specify the Composite design patterns. In: SAVCBS, pp. 89–94 (2008)
29. Jacobs, B., Smans, J., Piessens, F.: Verifying the Composite pattern using separation logic. In: SAVCBS, pp. 83–88 (2008)
30. Noble, J., Vitek, J., Potter, J.: Flexible alias protection. In: Jul, E. (ed.) ECOOP 1998. LNCS, vol. 1445, pp. 158–185. Springer, Heidelberg (1998)
31. Clarke, D.G., Potter, J.M., Noble, J.: Ownership types for exible alias protection. In: OOPSLA, pp. 48–64. ACM, New York (1998)
32. Dietl, W.: Universe Types Topology, Encapsulation, Genericity, and Tools. PhD thesis, ETH Zurich, 18522 (2009)
33. Bierman, G.M., Meijer, E., Torgersen, M.: Lost in translation: Formalizing proposed extensions to C#. In: OOPSLA, pp. 479–498. ACM, New York (2007)
34. Drossopoulou, S., Francalanza, A., Müller, P., Summers, A.: A Unified Framework for Verification Techniques for Object Invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 412–437. Springer, Heidelberg (2008)
35. Balzer, S.: A relationship-based programming language and its value for program verification. Technical report, ETH Zurich (2011)
36. Summers, A.J., Drossopoulou, S., Müller, P.: Universe-type-based verification techniques for mutable static fields and methods. JOT 8(4), 85–125 (2009)
37. Parkinson, M.J.: Local Reasoning for Java. PhD thesis, University of Cambridge (2005)
38. Parkinson, M.J., Bierman, G.M.: Separation logic and abstraction. In: POPL, pp. 247–258. ACM, New York (2005)
39. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
40. Kassios, I.T.: Dynamic frames: Support for framing, dependencies and sharing without restrictions. In: Misra, J., Nipkow, T., Karakostas, G. (eds.) FM 2006. LNCS, vol. 4085, pp. 268–283. Springer, Heidelberg (2006)
41. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008)
42. Smans, J., Jacobs, B., Piessens, F.: Implicit dynamic frames: Combining dynamic frames and separation logic. In: Drossopoulou, S. (ed.) ECOOP 2009. LNCS, vol. 5653, pp. 148–172. Springer, Heidelberg (2009)

A Pre- and Postconditions for **Composite** Specification

Below table indicates, for every method declared in Fig. 2, its preconditions and postconditions.

Precondition of Parent.append():

```
c != null && p != null && c != p &&
!(c isElementOf these.child union these.parent) &&
(!these.isEmpty() => p isElementOf these.child union these.parent);
```

Postcondition of Parent.append():

```
these.count() == old(these.count()) + 1 &&
these.select(c_p: c_p.child == c).count() == 1 &&
forAll(x isElementOf these.select(c_p: c_p.child == c): x.parent == p) &&
forAll(x isElementOf these.transitiveClosure().select(c_p: c_p.child == c).parent:
x.total == old(x.total) + 1);
```

Precondition of Composite.createComposite():

```
c != null;
```

Postcondition of Composite.createComposite():

```
these.count() == old(these.count()) + 1 &&
thereExists(x isElementOf these: x.root == c && x.tree.isEmpty());
```

Precondition of Composite.appendComponent():

```
c != null && p != null && c != p &&
(this.tree.isEmpty() => this.root == p) &&
(!this.tree.isEmpty() => p isElementOf this.tree.child union this.tree.parent &&
!(c isElementOf this.tree.child union this.tree.parent));
```

Postcondition of Composite.appendComponent():

```
this.tree.count() == old(this.tree.count()) + 1 &&
this.tree.select(c_p: c_p.child == c).count() == 1 &&
forAll(x isElementOf this.tree.select(c_p: c_p.child == c): x.parent == p);
```

Precondition of Composite.appendSubComposite():

```
p != null && !(p isElementOf c.child) &&
(!c.isEmpty() => p isElementOf c.parent) &&
(c.child == c.transitiveClosure().select(c_p: c_p.parent == p).child) &&
(this.tree.isEmpty() => this.root == p) &&
(!this.tree.isEmpty() => p isElementOf this.tree.child union this.tree.parent) &&
(c.child intersection (this.tree.child union this.tree.parent)).isEmpty();
```

Postcondition of Composite.appendSubComposite():

```
this.tree.count() == old(this.tree.count()) + c.count() &&
this.tree == old(this.tree) union c;
```

Precondition of Composite.appendComposite():

```
c != null && p != null &&
p != c.root && !(p isElementOf c.tree.child union c.tree.parent) &&
(this.tree.isEmpty() => p == this.root) &&
(!this.tree.isEmpty() => p isElementOf this.tree.child union this.tree.parent) &&
!(c.root isElementOf this.tree.child union this.tree.parent) &&
(!c.tree.isEmpty() =>
((this.tree.child union this.tree.parent) intersection (c.tree.child union c.tree.parent)).isEmpty();
```

Postcondition of Composite.appendComposite():

```
this.tree.count() == old(this.tree.count()) + 1 + c.tree.count() &&
this.tree == old(this.tree) union {(c.root, p)} union c.tree;
```