Modular Reasoning about Invariants over Shared State with Interposed Data Members

Stephanie Balzer Thomas R. Gross

Department of Computer Science, ETH Zurich, CH-8092 Zürich, Switzerland
{balzers, trg}@inf.ethz.ch

Abstract

Reasoning about object-oriented programs is difficult since such programs usually involve aliasing, and it is not easy to identify the ways objects can relate to each other and thus to confine a program's heap. In this paper, we address this problem in the context of a relationship-based programming language. In relationship-based programming languages, relationships are first-class citizens and allow a precise description of inter-object relationships. Relationships enforce a *modularization discipline* that is closer to the natural modularity inherent to many problem domains and that yields, as a result, program heaps that are DAGs. We further describe a mechanism, *member interposition*, that leverages the new modularization discipline and supports encapsulation of fields of shared objects. We have implemented the described modularization discipline and the mechanism of member interposition in the context of Rumer, a relationship-based programming language with support for contract specifications. We discuss the implications of member interposition for the modular verification of object invariants with an example. Relationships and interposed members provide an alternative to ownership type systems.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Languages, Verification

Keywords Invariants, Ownership type systems

1. Introduction

Object-oriented programming languages have become first choice in many application domains as they provide programmers with useful abstractions to create large software systems. Equally important as the availability of appropriate abstractions is the ability to reason *modularly* about the individual components of a large software system. Modular reasoning becomes even more important in program verification, allowing modules to be verified independently from each other with the guarantee that their verification remains valid upon module composition.

The presence of *aliases* in object-oriented programs, however, compromises modular reasoning considerably. Various ownership type systems [14, 24] have been proposed to mitigate the adverse affect of aliases to program understanding, and they have proved effective for a variety of tasks [9, 13, 15]. Ownership type systems have further been successfully applied to program verification, allowing for modular verification of object invariants [5, 18, 21].

Classical ownership type systems typically require objects to have precisely one primary owner and thus structure the heap into an ownership tree. Unfortunately, such systems increase the annotation burden on the programmer, and furthermore, conformance to a tree structure is often too restrictive and conflicts with many programming idioms. More recent approaches relax some of the classical restrictions (i.e., single ownership) [5, 11, 19] and offer more flexibility. Of course, the gain in flexibility comes at the cost of detailing the class interdependences [5, 11] and a limited applicability [19] of the approach.

The very fact that a programming language must set-up an auxiliary "machinery" to support modular reasoning clearly questions the modularization capabilities of the language. In this paper, we investigate an alternative approach: we introduce a simple sequential programming language, *Rumer*, which is a representative of a *relationship-based programming language* [8, 23, 25, 26, 28]. Relationship-based programming languages offer abstractions similar to the ones found in conceptual modelling languages (e.g., Unified Modelling Language (UML) [16], Entity-Relationship Diagrams [12]), accommodating more naturally the modularity inherent to many problem domains. More specifically, relationship-based programming languages support, similarly to class-based object-oriented languages, classes to represent real-world artifacts. In addition to classes, however, relationship-based programming languages also support the abstraction of a *relationship* to represent the possible collaborations between instances of classes. Without first-class support, the abstraction of a relationship is typically lost in the implementation as relationships can be represented only by means of reference fields and methods and thus are likely to distribute across several classes.

In contrast to other *relationship-based* programming languages, Rumer departs in various respects from established object-oriented principles. In particular, Rumer does not allow objects to reference each other. Instead, programmers must declare relationships to represent any express of interactions between objects. This treatment enforces a *modularization discipline* that requires a programmer to compose modules in a "top-down" fashion, with the resulting object structure forming a DAG. To emphasize the changed semantics, we use the term *entity* (instead of class) to denote the type constructor for objects. Our language further supports a refinement mechanism, *member interposition*, that allows refinement of entities in the context of a relationship [2]. By interposing a member into an entity, a relationship can add those features to a participating entity that are relevant to the collaboration. Interposed members are accessible only from within the relationship, *encapsulating* those entity members within the relationship.

PLPV'10 January 19, 2010, Madrid, Spain.
Copyright © 2010 ACM 978-1-60558-890-2/10/01...$10.00

This paper makes the following contributions:

- the introduction of a *modularization discipline* in the context of a relationship-based programming language that makes program heaps become DAGs.
- the use of *member interposition* as a means to *encapsulate* specific properties of otherwise shared objects within relationships.
- a discussion of the implications of member interposition for the *verification* of program *invariants*.
- a discussion of the *implementation* of member interposition in our prototype compiler for the programing language Rumer.

2. Invariants over Shared State

To illustrate the issues in the verification of program invariants over shared state, we use the example of synchronized clocks, adapted from [5], as a running example. Figure 1 shows a corresponding implementation in a Spec$^\sharp$-like language [4]. The program makes use of contract specifications — method preconditions (**requires**) and postconditions (**ensures**) as well as object invariants (**invariant**). Whereas a precondition denotes the states in which a caller is allowed to call the method and a postcondition denotes the state in which the implementation is allowed to terminate [3], an object invariant spells out what is expected to hold of each object's data fields in the steady states of the object [4]. There is no consensus among the various verification techniques what the steady states of an object are. The Spec$^\sharp$programming system [4], in particular, does not assume a *visible state semantics* [22], requiring object invariants to hold in pre-states and post-states of method executions, but introduces a block statement `expose` to delineate the non-steady states of an object. Such expose blocks offer an elegant solution to the verification of object-oriented programs in the presence of reentrant method calls [3]. Since the treatment of reentrancy is orthogonal to the issue discussed in this paper and has furthermore been tackled successfully [3], we omit the specification of expose blocks for the discussion in this paper.

The program in Figure 1 implements the idea of clock synchronization where a clock can periodically be synchronized with a master clock by copying the time of the master clock. This modus operandi should guarantee that the time of a particular clock is always less than or equal to the time of its master. The object invariant declared in class `Clock` in line 31 confirms this understanding. Unfortunately, modular reasoning about a clock's invariant is not possible without introducing further restrictions. As the object invariant of class `Clock` depends not only on the state of the current object but also on the state of the object referred to by the reference `master`, it is susceptible to invalidation by any changes done to the referred to object. For instance, changing the implementation of method `Tick()` in class `Master` to decrease rather than increase the master time will falsify the clock invariant.

At first, one might expect that ownership would solve the problem. To allow for modular verification of the clock's invariant in that case, field `master` in class `Clock` would need to be declared as being part of the representation of class `Clock`, making `Clock` the owner of `Master`. Doing so would certainly restore local reasoning, however, this setup would foil the intention of our example. Namely, it should be possible to associate a master clock with several clocks, preventing a clock from becoming the single owner of a master clock.

The crux of the example is that the field `time` of a master clock should be shared among several clocks while at the same time be "controlled" by every single clock that is synchronized with that master clock. Barnett and Naumann [4] introduce *friendship* to solve this problem. Friendship is a protocol that allows invariants expressed over shared state. In that protocol, a *granting* class can give privileges to another *friend* class, allowing the invariant of the friend class to depend on fields of the granting class. In our example, class `Master` would take the place of the granting class, whereas class `Clock` would take the place of the friend class. For the protocol to work, however, both classes must be complemented with additional specifications. For instance, a granting class must name its friend classes and list the fields these friends depend on, and further maintain some bookkeeping information about its actual friends. A friend class, in turn, needs to register with a granting class and constrain the possible updates of the fields of the granting class it depends on.

```
1  class Master {
2    int time;
3    String location;
4
5    Master(String loc)
6      ensures time == 0;
7    { time = 0; location = loc; }
8
9    Tick(int incr)
10     requires 0 <= incr;
11     ensures old(time) <= time;
12   { time = time + incr; }
13
14   invariant 0 <= time;
15 }
16
17 class Clock {
18   int time;
19   String location;
20   Master master;
21
22   Clock(Master m, String loc)
23     ensures master == m && time == 0;
24   { master = m ; location = loc; time = 0; }
25
26   Sync()
27     ensures time == master.time;
28   { time = master.time; }
29
30   invariant
31     master != null ==> 0 <= time && time <= master.time;
32 }
```

Figure 1. Simple program implementing synchronized clocks in a Spec$^\sharp$-like programming language.

3. Rumer in a Nutshell

Rumer is a simple sequential programming language that supports *relationships as first-class citizens*. In contrast to other *relationship-based* programming languages and systems [8, 23, 25, 26, 28], Rumer enforces a *modularization discipline* that requires a programmer to compose modules in a "top-down" fashion, with the resulting object structure forming a DAG. Furthermore, Rumer supports contract specifications, as present in object-oriented contract languages [4, 10, 17, 20], that are checked dynamically by the current prototype compiler.

Figure 2 shows the corresponding implementation of the running example in Rumer. Rumer supports two user-definable types: the entity and the relationship. An *entity* describes the abstract characteristics of objects of a particular kind. A *relationship*, on the other hand, describes the abstract characteristics of the collaborations between some particular objects. The program in Figure 2 declares the entity `Clock` and the relationship `SyncClocks`. Whereas the entity `Clock` defines the common properties of clocks, the relationship `SyncClocks` defines a specific collaboration between such clocks, namely the synchronization between a clock and its master. A relationship indicates in its **participants** clause the

```
1  entity Clock {
2    string location;
3
4    init initialize(string loc)
5    { location = loc; }
6  }
7
8  relationship SyncClocks
9    participants (Clock client, Clock server) {
10   int >client time;
11   int >server time;
12
13   init initialize()
14     ensures client.time == 0 & server.time == 0;
15   { client.time = 0; server.time = 0; }
16
17   void >server tick(int incr)
18      requires 0 <= incr;
19      ensures  old(time) <= time;
20   { time = time + incr; }
21
22   void sync()
23     ensures client.time == server.time;
24   { client.time = server.time; }
25
26   invariant
27     0 <= client.time & client.time <= server.time;
28 }
```

Figure 2. Simple Rumer program implementing synchronized clocks.

entity types involved in the collaboration. In the example, the collaboration occurs between clocks. Rumer allows a programmer to associate *role names* with the entities participating in a relationship. In the example, the role names `client` and `server` are used to denote a clock and its master clock, respectively. Role names can be used to refer to a participating object in the program text (e.g., lines 15 and 24). Both entities and relationships can furthermore declare fields and methods. For instance, entity `Clock` declares the field `location` as well as the initialization method `initialize()`. An initialization method is similar to a creation procedure in Eiffel [20] and is only used during instance creation. Relationship `SyncClocks`, on the other hand, declares the fields `time` — one for each role — (see Section 4) as well as an initialization method and the methods `tick()` and `sync()`. Lastly, Rumer also supports a code block (omitted in the example) that plays a role similar to the"main" function in other languages and allows the "root" combination of relationships and entities.

Entities correspond to classes in object-oriented languages but differ in an important aspect: an entity can declare only value type fields[1]. This restriction applies similarly to relationships: a relationship can only declare value type fields, yet has access to its participating entities through the declared role names. Limiting field types to value types has important consequences: *(i)* it changes the modularization of a program considerably since object structures are built declaratively through relationship declarations rather than imperatively through reference manipulations, and *(ii)*, it prevents cycles in the traversals of such object structures since entities can only be reached from relationships (but not the other way around) and since no entity nor relationship can refer to another entity or relationship, respectively.

Figure 3 shows a schematic view of the Rumer heap for the running example. The figure depicts the partitioning of the program heap due to the entity and relationship declarations. Entity instances (i.e., objects) are depicted as filled circles and relationship instances are depicted as a pair of entity instances surrounded by an ellipse. In the example, there are two partitions: one comprising all instances of entity `Clock` and one partition comprising all instances of relationship `SyncClocks`. In Rumer, these partitions are, inspired by database technology, called *extents*. Such extents are accessible to the programmer and can be queried using query expressions similar to C$^\sharp$ LINQ heap queries [7]. The arrows in Figure 3 indicate possible heap traversals. Using the roles `client` and `server` one can reach a clock and a master clock object, respectively, from a corresponding `SyncClocks` relationship instance. As indicated by the figure, the program heap forms a DAG with entities and relationships as nodes and role projection operators as edges.

4. Member-Level Encapsulation

This section introduces the mechanism of member interposition and discusses its use for enforcing member-level encapsulation as well as its implications for the verification of program invariants.

4.1 Member Interposition

The Rumer programming language supports *member interposition*, a mechanism introduced in earlier work [2] that allows refining entity instances in the context of a relationship. An interposed member (i.e., field or method) declaration appears within a relationship declaration like any other "regular" member and is denoted by the '>' symbol. In Figure 2, interposed members are declared in lines 10, 11, and 17. The '>' symbol is followed by a role name and indicates of which participating entity the declared member is a property. In the example, two interposed fields with the name `time` are declared, whereas one is interposed into the clock playing the role of a client ("clock") and the other is interposed into the clock playing the role of a server ("master clock"). To understand the semantic difference between non-interposed relationship members and interposed relationship members, let's assume that we introduce a third field called time in relationship `SyncClocks` that we declare as a non-interposed member (i.e., **`int`** `time`). This set-up would allow us to record not only the time of a particular clock and the time of its associated master clock, but also the time when the two clocks were last synchronized. In terms of Figure 3, non-interposed relationship members can be thought of as being associated with a relationship instance (ellipse), and thus with an entity instance pair, whereas interposed relationship members can be though of as being associated with an entity instance (filled circle).

Interposed members are encapsulated within their declaring relationships. As a result, interposed members are accessible only from within their defining relationships, but not from within the entities they affect. This means further, that only the properties defined in an entity declaration are "globally" visible and can be shared among all the relationships the entity participates in. Basically, member interposition introduces a subtyping relation between the entity and the role the entity plays in a relationship. This means in turn, that a role inherits the members defined by the entity it refines, whereby an interposed member shadows any inherited member with the same name. Let's assume, for instance, that we import entity `Clock` from a library and that this entity already declares a field `time`. In that case, the interposed fields `time` declared in relationship `SyncClocks` for the roles `client` and `server`, respectively, would shadow the field `time` declared in the entity `Clock`.

To understand the encapsulation capabilities offered by member interposition, let's explore whether the relationship *SyncClocks* of Figure 2 could be emulated by a class in a class-based object-oriented language. Obviously that class would need to declare references to the participants of the relationship, namely the fields `Clock client` and `Clock server`, respectively. Such a class

[1] Note that type **`string`** in Rumer is a value type and not an immutable reference type.

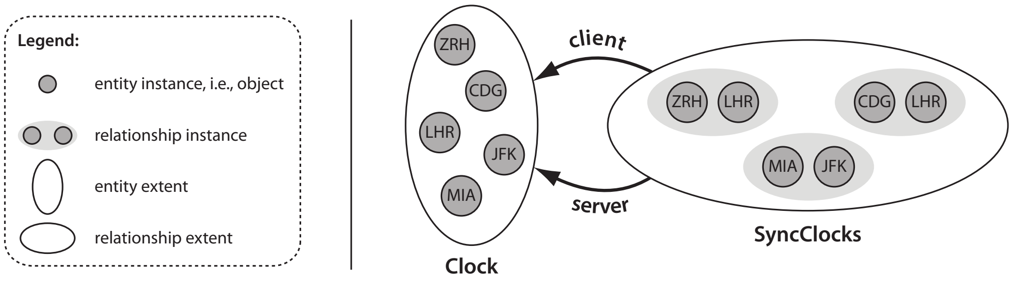

Figure 3. Schematic view of the Rumer program heap for the synchronized clocks example.

would further need to declare a method for the initialization method `initialize()` and a method for the non-interposed method `sync()`. Since the class emulating the relationship maintains references to its participants, an instance of that class represents a relationship instance. Any member declared in that emulating class thus represents a property of a relationship participant pair rather than a property of a particular relationship participant. As a result, the interposed fields `time` and the interposed method `tick()` could not be declared in the emulating class. Instead, those interposed members would need to be declared in the participant class `Clock` where they become no longer encapsulated by the class emulating the relationship.

4.2 Controlling the Visibility of Changes

Noble et al. [24] have made the observation that in an object-oriented program *"aliasing per se is not the major problem — rather, the problem is the visibility of changes caused via aliases"*. To control aliasing, it seems thus not to be necessary to categorically forbid state changes, but to limit the visibility of those state changes one cares about. Member interposition is an instrument offering remedy in exactly such situations: by interposing a member into a participant of a relationship, the visibility of that member is local to the relationship and thus any changes to the state of that member become fully encapsulated by the relationship. Member interposition is thus not only a refinement mechanism but also a form of *alias control* mechanism with regard to a participant's properties.

Using member interposition as a mechanism to control aliasing is appealing for a number of reasons: *(i)* Member interposition allows for *member-level encapsulation*. Whereas classical alias control approaches typically rely on ownership and make the owner own the *entire* object, member interposition makes the relationship only "own" a *subset* of the object, namely the members it interposes. As a result, member interposition enables the *sharing* of "owned" objects among several relationships. *(ii)* Member interposition is a *light-weight technique* to alias control. Since, on the type system level, the role that an entity plays in a relationship is represented as a separate type that is a subtype of the entity, the prevention of representation exposure can be achieved by regular type conformance checks and does not require any special-purpose typing rules. *(iii)* Member interposition is a *non-invasive* technique. Member-level encapsulation of entities can be achieved without touching the entity itself, a property that eases also the integration of library functionality.

4.3 Implications for Invariant Verification

The previous section has shown that member interposition facilitates modular reasoning about a relationship. This result suggests that member interposition may also ease the modular verification of invariants. Let's consider again the Rumer implementation of our running example (see Figure 2). The invariant declaration of relationship `SyncClocks` (line 27) asserts that a clock's time (`client.time`) is less than or equal to the time of its associated master clock (`server.time`) and that both times are non-negative. As opposed to the invariant of class `Clock` in the Spec$^\sharp$-like implementation of the running example (see Figure 1), the invariant of relationship `SyncClocks` depends only on fields declared in the relationship. The use of interposed fields in its invariant, allows the relationship `SyncClocks` to control any updates of those fields, guaranteeing that such updates do not invalidate the invariant. Given the encapsulation of the affected fields and the postconditions of methods `initialize()`, `tick()`, and `sync()`, the invariant of relationship `SyncClocks` can thus be modularly verified. In order to be amenable to modular verification, a program has thus to meet certain well-formedness conditions. This gives raise to the definition of an *admissible invariant*.

Definition 1. *An invariant is admissible if it only mentions fields that are encapsulated by the invariant declaring type.*

For a relationship invariant, Definition 1 requires that the fields occurring in the invariant declaration are either non-interposed fields or interposed fields. This restriction rules any occurrences of inherited entity fields out. Although not used in Figure 2, entities can also declare invariants. Of course Definition 1 applies likewise to entity invariants. Since an entity is, as opposed to a relationship, self-contained, any field occurrence in an entity invariant will be encapsulated by the entity and thus be admissible.

The invariants used in the code examples so far, constrain the state of objects and relationship instances by delimiting their field values. Other kinds of constraints are conceivable. In the Spec$^\sharp$-like implementation of the running example (see Figure 1), an implicit constraint on the cardinality of the synchronization collaboration exists. By declaring field `master` of class `Clock` to be of type `Master` rather than of type `LinkedList<Master>`, we implicitly enforce that a clock is associated with at most one master clock. Figure 4 shows an extended version of the Rumer implementation of the running example, taking care of this additional constraint explicitly in the invariant declaration.

The new invariant `SyncClocks.`**`isPartialFunction`**`()` declared in line 41 of relationship `SyncClocks` in Figure 4 uses a Rumer built-in predicate to test whether the target of the predicate invocation (i.e., the extent `SyncClocks`) is a partial function. The example demonstrates another consequence of Rumer's modularization discipline: the declarative nature of program construction and thereby induced heap partitioning makes the language inherently set-oriented. As a result, Rumer abstracts entity extents as sets of objects and relationship extents as relations [2]. Requiring the extent of relationship `SyncClocks` to form a partial function precisely realizes our design intention, namely that a clock can synchronize with at most one master clock.

Invariants like `SyncClocks.`**`isPartialFunction`**`()` declared in relationship `SyncClocks`, are representatives of a different kind of invariant category. As opposed to the invariant in line 40, which restricts the state of the participating entities, the new invariant restricts the state of the relationship extent. In line with the terminology introduced in [2], we call the former category *value-based invariants* as such invariants constrain the values instances may assume for their fields, and we call the latter category *structural invariants* as such invariants constrain the occurrence of instances in their respective extents. As indicate in Figure 4, structural invariants are prefixed by the keyword **`extent`**.

Structural invariants entail new requirements with regard to their modular verification. The underlying scheme to achieve modular reasoning, however, remains essentially the same for structural invariants as for value-based invariants. The main idea is basically to encapsulate any state changes within the declaring type of the invariant. For structural invariants this means consequently that any additions to entity and relationship extents as well as any removals from these extents need to be encapsulated within the entity and relationship, respectively.

To enable modular verification of its structural invariant, the extended version of the Rumer program (see Figure 4) encapsulates any such extent-changing operations, such as the creation of relationship instances, in method `createSyncClocks()`. As opposed to the methods `tick()` and `sync()`, which are instance methods, the method `createSyncClocks()` is declared to be an *extent method* and behaves analogously to a static method in class-based object-oriented programming languages. The creation of a relationship tuple and its addition to its corresponding extent happen in the body of the if statement in method `createSyncClocks()`. The **`add`**`()` operation builds the Cartesian product of the sets passed as second and third argument, initializes each resulting tuple as defined by the initialization method passed as first argument, and adds the tuple to the `SyncClocks` extent. The second and third argument of the addition operation make use of Rumer's built-in querying facility to retrieve the clocks with the indicated location. Note that method `createSyncClocks()` only creates and adds a new relationship tuple if the client clock in question has not yet been associated with a master clock (i.e., is not yet contained in the `SyncClocks` extent). Provided that the location names passed as arguments are unique (precondition), method `createSyncClocks()` guarantees in turn to maintain the structural invariant (postcondition). Given the fact that relationship `SyncClocks` encapsulates any state changes within appropriate methods, and given the fact that these methods are appropriately annotated with pre- and postconditions, we can conclude that the entire invariant of relationship `SyncClocks` can be modularly verified.

We are now in the position to update Definition 1 to include the treatment of structural invariants. Before doing so, however, we must briefly discuss two last issues with regard to the modular verification of structural invariants. The first issue relates to the removal of instances from an extent. The modular verification of a structural invariant requires not only that any removal operations are encapsulated by their declaring types, but also that no entity instance can be removed from an extent as long as the entity instance participates in a relationship. Rumer accounts for this reservation and treats entity removals as "scheduled for removal" requests, guaranteeing that an entity instance is only removed from an extent once it is no longer participating in any relationship. The second issue, lastly, relates to the infeasibility of the modular verification of certain kinds of structural invariants. More precisely, these are invariants constraining a relation to be total or surjective, respectively. To modularly reason whether a relation is total or surjective, a relationship would need to own the respective entity extent. Since entities are shared among several relationships, however, a relationship cannot become the owner of an entity extent. The final definition for an admissible invariant thus becomes:

```
entity Clock {
  string location;

  init initialize(string loc)
  { location = loc; }
}

relationship SyncClocks
  participants (Clock client, Clock server) {
  int >client time;
  int >server time;

  init initialize()
    ensures client.time == 0 & server.time == 0;
  { client.time = 0; server.time = 0; }

  void >server tick(int incr)
     requires 0 <= incr;
     ensures  old(time) <= time;
  { time = time + incr; }

  void sync()
    ensures client.time == server.time;
  { client.time = server.time; }

  extent void createSyncClocks (string clLoc, string svLoc)
    requires
      Clock.select(c: c.location == clLoc).count() <= 1 &
      Clock.select(c: c.location == cvLoc).count() <= 1;
    ensures SyncClocks.select(c_s:
            c_s.client.location == clLoc).count() <= 1;
  {
    if (!(Clock.select(c: c.location == clLoc) isSubsetOf
          SyncClocks.client)) {
      SyncClocks.add(new SyncClocks().initialize(),
        Clock.select(c: c.location == clLoc),
        Clock.select(c: c.location == svLoc));}
  }

  invariant 0 <= client.time & client.time <= server.time;
  extent invariant SyncClocks.isPartialFunction();
}
```

Figure 4. Extended version of Rumer program from Figure 2. Added code is highlighted.

Definition 2. *A value-based invariant is admissible if it only mentions fields that are encapsulated by the invariant declaring type. A structural invariant is admissible if it does not impose a totality or surjectivitiy constraint on a relationship and if all addition and removal operations are encapsulated by the invariant declaring type.*

From Definition 2 we can finally conclude that, using Rumer's enforced modularization discipline together with member-level encapsulation, all value-based invariants and most structural invariants (all but those requiring totality or surjectivitiy of a relationship) can be turned into admissible invariants and are thus amenable to modular verification.

5. Implementation

We implemented a prototype compiler for Rumer that supports member-level encapsulation through member interposition. The current prototype compiler supports furthermore invariant declarations and offers run-time monitoring of such invariants. Our compiler is a source-to-source compiler, using Java as the target language.

As the target language of the Rumer compiler is Java, the code generator must map the language constructs of Rumer onto the available constructs in Java. In particular, the compiler defines a mapping from the user-definable types in Rumer, i.e., entities and relationships, onto Java classes. This mapping relies basically on a simplified version of the role object pattern [6], providing separate classes for both entity and relationship declarations as well as for entity roles to accommodate interposed members. Furthermore, the compiler defines a set of Java classes comprising the run-time data structures necessary to execute the target program. These are in particular classes representing extents, sets, as well as bags and their corresponding built-in operations (e.g., **`add`**, **`remove`**, **`select`**, **`map`**, etc.).

The implementation of lambda functions became actually the main technical "obstacle" to overcome since Java does not provide inherent support for closures. As a result, the compiler emulates closures in the generate target code using anonymous inner classes. A well-known problem regarding the emulation of closures with anonymous inner classes, however, is the fact that anonymous inner classes do not properly close over their surrounding scope. In particular, anonymous inner classes copy the values of any primitive local variables of the surrounding scope that are accessed in the anonymous inner class, requiring the programmer in turn to declare such variables final. Even though for our purposes a lambda function actually does not need to change the content of a local variable of the surrounding scope, requiring such variables to be final in general is too restrictive as it forbids any updates of those variables. To circumvent this restriction, the Rumer compiler introduces a generic wrapper class for wrapping primitive values and replaces all reads of local primitive values in anonymous inner classes with respective getter invocations on an instance of that wrapper class.

6. Related Work

Our work intersects mainly with efforts in two separate domains: relationship-based programming languages and ownership type systems.

Relationship-based programming languages: While the importance of relationships as a means to model software systems dates back approximately 20 years [1, 27], it is only recently that the support of relationships as a separate abstraction in an object-oriented programming language has been investigated. As a result of these efforts *relationship-based programming languages* [8, 23, 25, 26, 28] have emerged. Whereas existing approaches to relationship-based programming languages rely firm on object-oriented principles and mainly extend object-oriented languages with relationship support, Rumer departs in various respects from established object-oriented principles. The most important distinguishing features of Rumer with regard to other relationship-based languages are its enforced modularization discipline and its set-orientation. Furthermore, Rumer is the only relationship-based language that supports member interposition.

Ownership type systems: The work on ownership type systems that is most closely related to our work is the research on the verification of ownership-based invariants [5, 18, 19, 21]. These efforts have contributed a methodology for the verification of class-based object-oriented programming languages equipped with contracts. Our work differs from the work on the verification of ownership-based invariants mainly in *(i)* the targeted programming language paradigm and, more importantly, in *(ii)* the actually applied alias control technique. Whereas existing approaches enforce a single owner regime where the owner owns the entire object, our work allows the sharing of owned objects where the owner only owns a subset of the object's state. The only other techniques that also drop the single ownership restriction, is the work by Barnett and Naumann [5] on friendship-based invariants and the work by Leino and Schulte [19] on history invariants. Both techniques allow the verification of the observer pattern but differ in applicability and entailed restrictions. Whereas in the friendship-based approach a granting class (subject) needs to "know" its friend (observer) classes, the history invariant-based approach drops this restriction. This flexibility, however, comes at the cost of limiting the possible evolution of the subject. Whereas in the friendship-based approach an observer can restrict how a subject can evolve over time by specifying an appropriate update guard, in the history invariant-based approach the subject is limited to evolve monotonically. Our work essentially differs from friendship and history invariants in the targeted programming language paradigm and in the fact that our approach does not require adapting the collaborating classes nor does it limit the possible evolution of the subject.

7. Conclusions

This paper introduces a simple sequential programming language with contracts, *Rumer*, that supports relationships as first-class citizens and that enforces a *modularization discipline* requiring programmers to compose modules in a "top-down" fashion. The enforced modularization discipline has a surprising consequence: it prevents cycles in the traversals of object structures and thus make the program heap become a DAG. Building on the new modularization discipline and a technique to refine relationship participants (member interposition), the paper contributes a *light-weight technique to alias control* allowing for *member-level encapsulation* of *shared* objects. Light-weight encapsulation based on member interposition furthermore facilitates modular reasoning, and the paper discusses the implications of member interposition for the verification of entity and relationship invariants. It turns out that thanks to the new modularization of a program and thanks to the use of member interposition to encapsulate the fields of an entity a relationship depends on, the major part of invariant declarations in a Rumer program are amenable to *modular verification*.

Limiting field types to only value types may seem at first overly restrictive. In developing Rumer programs so far, we have never come across a situation where a problem was not implementable in Rumer. We were rather confronted with the usual issue, that comes along with any change in programming paradigm, of finding out what the "Rumer-way" of expressing the problem would be. To finally address this question, further experience with Rumer, in particular on a larger scale, is necessary. The prospective benefits of dropping reference fields from a programming language, however, are certainly worth any investigation.

Acknowledgments

We are grateful to Jean-Raymond Abrial for his continuous support of this work. We also thank the annononymous reviewers for their detailed and insightful comments.

References

[1] A. Albano, G. Ghelli, and R. Orsini. A relationship mechanism for a strongly typed object-oriented database programming language. In *17th International Conference on Very Large Data Bases (VLDB'91)*, pages 565–575. Morgan Kaufmann Publishers Inc., 1991.

[2] S. Balzer, T. R. Gross, and P. Eugster. A relational model of object collaborations and its use in reasoning about relationships. In *21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 323–346. Springer, 2007.

[3] M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology (JOT)*, 3(6):27–56, 2004.

[4] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec$^\sharp$ programming system: An overview. In *International Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS'04)*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2004.

[5] M. Barnett and D. A. Naumann. Friends need a bit more: Maintaining invariants over shared state. In *7th International Conference on Mathematics of Program Construction (MPC'04)*, Lecture Notes in Computer Science, pages 54–84. Springer, 2004.

[6] D. Bäumer, D. Riehle, W. Siberski, and M. Wulf. The role object pattern. In *4th Conference on Pattern Languages of Programs (PLoP'97)*, 1997.

[7] G. M. Bierman, E. Meijer, and M. Torgersen. Lost in translation: Formalizing proposed extensions to Spec$^\sharp$. In *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 479–498. ACM, 2007.

[8] G. M. Bierman and A. Wren. First-class relationships in an object-oriented language. In *19th European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3586 of *Lecture Notes in Computer Science*, pages 262–286. Springer, 2005.

[9] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *17th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 211–230, New York, NY, USA, 2002. ACM.

[10] L. Burdy, Y. Cheon, D. R. Cok, M. D. Ernst, J. R. Kiniry, G. T. Leavens, K. R. M. Leino, and E. Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT'05)*, 7(3):212–232, 2005.

[11] N. R. Cameron, S. Drossopoulou, J. Noble, and M. J. Smith. Multiple ownership. In *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 441–460. ACM, 2007.

[12] P. P.-S. Chen. The entity-relationship model - toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, March 1976.

[13] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *17th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'02)*, pages 292–310. ACM, 2002.

[14] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *13th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'98)*, pages 48–64. ACM, 1998.

[15] W. Dietl, S. Drossopoulou, and P. Müller. Generic universe types. In *21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 28–53. Springer, 2007.

[16] I. Jacobson, G. Booch, and J. E. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

[17] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06-rev29, Iowa State University, 2006.

[18] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *18th European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *Lecture Notes in Computer Science*, pages 491–516. Springer, 2004.

[19] K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *16th European Symposium on Programming (ESOP'07)*, Lecture Notes in Computer Science, pages 80–94. Springer, 2007.

[20] B. Meyer. *Object-Oriented Software Construction*. Prentice Hall Professional Technical Reference, 2nd edition, 1997.

[21] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[22] P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, 2006.

[23] S. Nelson, D. J. Pearce, and J. Noble. First class relationships for OO languages. In *6th International Workshop on Multiparadigm Programming with Object-Oriented Languages*, 2008.

[24] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *12th European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Science*, pages 158–185. Springer, 1998.

[25] K. Østerbye. Design of a class library for association relationships. In *ACM SIGPLAN Symposium on Library-Centric Software Design (LCSD'07)*, 2007.

[26] D. J. Pearce and J. Noble. Relationship aspects. In *5th International Conference on Aspect-Oriented Software Development (AOSD '06)*, pages 75–86. ACM, 2006.

[27] J. Rumbaugh. Relations as semantic constructs in an object-oriented language. In *2nd ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '87)*, pages 466–481. ACM, 1987.

[28] A. Wren. *Relationships for Object-oriented Programming Languages*. PhD thesis, University of Cambridge, November 2007.