Diss. ETH No. 19851

RUMER: A PROGRAMMING LANGUAGE AND MODULAR VERIFICATION TECHNIQUE BASED ON RELATIONSHIPS

A dissertation submitted to
ETH Zurich

for the degree of
Doctor of Sciences

presented by
Stephanie Balzer

accepted on the recommendation of
Prof. Dr. Thomas R. Gross, examiner
Prof. Dr. Sophia Drossopoulou, co-examiner
Prof. Dr. Peter Müller, co-examiner
Dr. Alexander J. Summers, co-examiner

2011

Abstract

The idea of asserting a program's correctness by a mathematical proof rather than exhaustive testing has persisted in software system development. The notion of an invariant to capture the constant properties of data plays a central role in such program verification attempts. Invariants have also infiltrated object-oriented programming languages and have become a foundation for verifying object-oriented programs.

Current invariant-based, object-oriented verification techniques provide modular verification of single-object invariants. However, as soon as an invariant relates several objects, modular reasoning about the invariant is not possible without imposing certain restrictions on the program. Current solutions to the verification of such multi-objects invariants either restrict aliasing in a program or expand the visibility of an invariant beyond the boundary of the declaring class.

This thesis represents a two-part approach to program verification based on invariants. We develop a programming language, Rumer, that provides higher-level abstractions to capture not only objects but also the relationships between them. In addition, we elaborate a visible-state verification technique for Rumer that facilitates the modular verification of Rumer programs.

The Rumer programming language employs the abstraction of a relationship as a key architectural element in building a software system. Rather than employing references in objects to represent the relationships between objects, those relationships are represented explicitly by a relationship instance. This indirection gives rise to a stratified programming model. Based on this model, we formulate the Matryoshka Principle, a modularization discipline that provides strong encapsulation for Rumer types.

The verification technique is based on the Matryoshka Principle and leverages the particular features of the Rumer programming and specification language. The Matryoshka Principle provides modular reasoning about state changes and permits the adoption of a visible-state semantics for invariants. The Rumer programming and specification language encompasses techniques to encapsulate multi-object invariants in relationships and includes a rich assertion language that is based on discrete mathematics.

Zusammenfassung

Die Idee, die Korrektheit eines Programms durch einen mathematischen Beweis anstatt durch ausgiebiges Testen sicherzustellen, hat seit jeher eine zenrale Bedeutung in der Entwicklung von Softwaresystemen eingenommen. Verifikationsansätze basieren traditionsgemäss auf Invarianten, die jene Eigenschaften von Daten spezifizieren, die zur Laufzeit des Programms konstant sind. Invarianten finden mittlerweile auch in objektorientierten Programmiersprachen Anwendung und haben sich zu einer wichtigen Grundlage für die Verifikation objektorientierter Programme entwickelt.

Gängige invariantenbasierte, objektorientierte Verifikationstechniken erlauben die modulare Verifikation von Invarianten, die sich auf ein einzelnes Objekt beziehen. Setzt eine Invariante jedoch mehrere Objekte zueinander in Beziehung, ist modulares Schliessen nur mittels Einschränkungen des Programms möglich. Gängige Lösungsansätze für die Verifikation solcher Multi-Objekt-Invarianten unterbinden entweder das Aliasing (d.h. die Existenz unterschiedlicher Referenzen auf denselben Speicherplatz) oder weiten die Sichtbarkeit einer Invariante über die Grenzen der deklarierenden Klasse hinaus aus.

Die vorliegende Dissertation präsentiert einen zweistufigen Ansatz für die Verifikation invariantenbasierter Programme. Wir beschreiben die Programmiersprache Rumer, welche Abstraktionen für die Repräsentation von Objekten und deren Beziehungen zueinander zur Verfügung stellt. Des Weiteren erarbeiten wir eine Verifikationstechnik für Rumer. Diese basiert auf sichtbaren Zuständen und ermöglicht die modulare Verifikation von Rumer Programmen.

Die Programmiersprache Rumer verwendet die Abstraktion einer Beziehung als grundlegendes Sprachkonstrukt zur Bildung eines Softwaresystems. Dabei werden die Beziehungen zwischen Objekten durch explizite Beziehungsinstanzen repräsentiert und nicht durch Referenzen in den beteiligten Objekten. Die Verwendung expliziter Beziehungsinstanzen zusätzlich zu den beteiligten Objekten führt zu einem mehrschichtigen Programmiermodell. Ausgehend von diesem Programmiermodell formulieren wir das Matrjoschka-Prinzip — ein Modularisierungsansatz, der starke Kapselungseigenschaften für Rumer Datentypen gewährleistet.

Die Verifikationstechnik basiert auf dem Matrjoschka-Prinzip und nutzt die spezifischen Eigenschaften der Programmiersprache Rumer und deren Spezifikationssprache aus. Das Matrjoschka-Prinzip ermöglicht das modulare Schliessen über Zustandswechsel und garantiert, dass Invarianten in sichtbaren Programmzuständen gelten. Die Programmierspra-

che Rumer und deren Spezifikationssprache bieten Techniken zur Kapselung von Multi-Objekt-Invarianten in Beziehungen und unterstützen eine reiche, auf diskreter Mathematik beruhende Aussagensprache.

Acknowledgments

I am grateful to the following persons who have accompanied my journey to a PhD:

My advisor Thomas Gross. His patience, allowing me to explore and substantiate my research ideas, and his scientific curiosity towards new, maybe even unconventional, research ideas are unparalleled. But above all, he has guided me to become, I believe, an independent researcher.

My co-examiners Sophia Drossopoulou, Peter Müller, and Alex Summers. I am grateful for their valuable feedback, allowing me to improve this thesis importantly. Sophia's thought-provoking questions on the occasion of my visit have decisively influenced the last revision of the Rumer language. The discussions with Alex on visible-state verification techniques have deepened my understanding thereof, and I am very appreciative of Alex's meticulous reading of this thesis, allowing me to greatly polish it.

The researchers I was fortunate to get to know at ETH. I am grateful to: Jean-Raymond Abrial for introducing me to formal methods and for his support of my PhD work; Laurent Voisin for his assistance in formal methods and for sharing his enormous knowledge; Patrick Eugster for his belief in first-class relationships and for pushing me to "go for my ideas". Also, I would like to thank Bertrand Meyer for providing me a start at ETH.

My colleagues at ETH. In particular, I want to thank Susanne Cech Previtali for sharing my passion in the search for the right abstraction, her advice, and sisterhood. Also, I want to thank my colleagues of the LST research group Christoph Angerer, Mihai Cuibus, Nicu Fruja, Zoltán Majó, Nicholas Matsakis, Valery Naumov, Albert Noll, Mathias Payer, Michael Pradel, Florian Schneider, Yang Su, Luca Della Toffola, Oliver Trachsel, Cristian Tuduce, and Faheem Ullah for stimulating discussions, both on and off research topics. Furthermore, I want to thank Karine Arnout, Arnaud Bailly, Ilinca Ciupa, Werner Dietl, Lisa Ling Liu, Piotr Nienaltowski, Joseph Ruskiewicz, Silvia Santini, Bernd Schoeller, Sebastien Vaucouleur, and Christoph Wintersteiger for their support and friendship.

The students whose projects I was privileged to supervise: Christoph Bäni, Alexandra Burns, Reto Conconi, Roland Schilter, Ágnes Sebestyén-Pál, Shinji Takasaka, Michelle Volery, and Andrea Zimmermann.

My parents and my sister Charlotte. I am grateful for their love and encouragement.

My partner Adnan. Without his love and unconditional support this work would not have been possible.

Contents

List of Figures

List of Tables

1 Introduction

This thesis explores the value of first-class relationships for program verification.

First-class relationships in object-oriented languages. Object-oriented programming languages have become mainstream in today's software systems development. The abstraction of a *class* has proved viable to represent the artifacts of a software system, and advanced concepts, such as inheritance and polymorphism, allow for adequate extensibility and flexibility of the resulting object-oriented code. While the abstraction of a class allows for an accurate expression of a software system's artifacts, if falls short of expressing the *relationships* that exist between the individual artifacts. During the design of an object-oriented software system, the identification of the relationships that exist between the classes of the system is equally important to the identification of the classes themselves. Conceptual modeling languages account for this fact and provide useful abstractions that allow the expression of both the classes and the relationships between the classes. However, once the system is implemented in an object-oriented programming language, relationships are represented in terms of classes and reference fields, and the abstraction of a relationship disappears from the resulting object-oriented code.

Implementing relationships in terms of the primitives of an object-oriented programming language is disadvantageous: besides disconnecting the implementation of a software system from its design, it results in code that compromises information hiding and is prone to error. As a result, it has been suggested to grant relationships *first-class citizenship* in object-oriented programming languages [1, 2, 3, 4, 5, 6]. Programming languages with first-class relationships complement object-oriented programming languages with the abstraction of a relationship to capture the relationships between classes. Like classes, relationships can declare fields and methods, they can be instantiated, and their instances can be stored in variables or bound to method arguments and returns. Relationship-based implementations turn out to be more "lightweight" than their object-oriented counterparts. Most importantly, however, they restore modular reasoning about the relationships between the classes of a software system since relationship-based programming languages provide a separate abstraction to encapsulate relationships.

Object-oriented verification. Since the seminal works on program correctness [7, 8, 9], the idea of asserting a program's correctness by a mathematical proof rather than by exhaustive testing has persisted in software development. Program verification has also been studied in the context of object-oriented programming languages. A wealth of object-oriented verification techniques [10, 11, 12, 13, 14, 15, 16, 17, 18, 19] has been suggested that adopt the idea of an *invariant* [20] as a foundation for verifying object-oriented programs [21]. An object invariant captures the properties of an object that the object exhibits in its consistent states.

A key issue for any practical object-oriented verification technique 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 only constrains the state of the current object and as long as an object's fields can only be written to 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 naturally modular to the current object. If aliases to the referenced objects exist, then those objects can be altered by the aliases and the multi-object invariant compromised without the current object noticing.

Multi-object invariants compromise also the adoption of a *visible-state semantics* for invariants. A visible-state verification technique [16] requires an object to meet its invariant in the initial and final states of method executions (i.e., the visible states) but allows an object to temporarily break its invariant during the execution of a method. A visible-state semantics cannot generally be applied soundly to multi-object invariants. To re-establish a multi-object invariant by its final state, a method may need to invoke methods on the objects related by the invariant, and those methods may in turn re-enter (i.e., "call-back" into) the original object in an inconsistent state.

Existing techniques [10, 11, 12, 13, 14, 15, 16, 17, 18, 19] 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. Whereas some techniques [10, 11, 12, 16, 18, 19] rely on a visible-state semantics for invariants and address the problems of multi-object invariants and re-entrant method invocations by leveraging ownership types or by enlarging or refining an invariant's scope of visibility, other techniques [13, 14, 15, 17] abandon a visible-state semantics for invariants and require programmers to indicate the program states in which invariants are expected to hold. Furthermore, a number of object-oriented verification techniques have been suggested that are not based on invariants, but leverage heap partitioning. To facilitate heap-local reasoning in the presence of aliasing, those works either introduce the ideas of separation logic [22] to Java [23, 24, 25, 26] or express heap disjointness assertions based on dynamic frames [27, 28, 29, 30].

1.1 Thesis

This thesis represents a combined effort to develop a *relationship-based programming language* as well as a *modular, invariant-based verification technique* for that language. Two observations initiated this work: *(i)* that relationship-based programming languages offer novel modularization possibilities and *(ii)* that in object-oriented verification an invariant's granularity is at odds with a type declaration's granularity since invariants may span across class boundaries. Based on these observations, this thesis hypothesizes that the challenges of invariant-based, object-oriented program verification can be addressed by reconciling the language used to implement object-oriented software with the language used to specify and verify object-oriented software.

This thesis contributes the relationship-based programming language Rumer as well as a modular, invariant-based verification technique for Rumer.

The design of the Rumer programming language was driven by the desire to grasp the genuine characteristics of first-class relationships and to leverage these characteristics both for program construction and program verification. These efforts resulted in a novel modularization discipline, the *Matryoshka Principle*, as well as in the support of new language mechanisms. The Matryoshka Principle requires programs to be composed from relationships, since relationships indicate how instances may refer to each other, and results in a stratification of a program's relationship and class declarations. Admissibility criteria are then specified based on this stratification, providing strong encapsulation guarantees. The new language mechanisms encompass *member interposition* and *predicate references*. Member interposition allows appropriate encapsulation of those object properties that are dependent on the object's participation in a relationship. Predicate references rely on Rumer's side-effect free heap-querying support and denote predicates over the program heap.

The verification technique developed for the Rumer programming language restores a visible-state semantics for multi-object invariants and allows the modular verification of Rumer programs. It supports a rich range of assertions to be expressed and allows the declaration of both object invariants and relationship instance invariants, including invariants that quantify over several objects or relationship instances. The verification technique leverages the particular characteristics of the Rumer programming language. It relies on the Matryoshka Principle to be guaranteed that there are no transitive call-backs and on member interposition to modularly reason about shared state. In addition, the Rumer verification technique allows the Matryoshka Principle to be superimposed with *selective ownership*, a flexible form of ownership that facilitates the modular verification of invariants that constrain several relationships.

1.2 Content and contribution

We next provide an overview of the thesis' content and detail its contributions.

Chapter 2. In this chapter, we provide a survey of how relationships are represented in terms of the primitives of an object-oriented programming language. We start with distilling a list of desired features for object-oriented relationship implementations and derive a set of *programming idioms* that categorize concrete object-oriented implementations of relationships. Then, we report on an *empirical study* on the occurrence of the programming idioms in a corpus of 24 real-world Java programs. We conclude the chapter with a discussion of the shortcomings of pure object-oriented relationship implementations as well as an introduction to first-class relationships and their related work.

Contributions: This chapter contributes an empirical study on the occurrence of object-oriented relationship implementations in Java programs and a discussion of the benefits of first-class support of relationships for software construction.

Chapter 3. In this chapter, we introduce the relationship-based programming language Rumer. We start with presenting the meta model underlying the Rumer language and then present the programming language itself, including the main language concepts, the specification language, the language's type system and semantics. The language's semantics give rise to a number of system invariants, upon which the verification technique relies. We delimit the differences between existing relationship-based languages and Rumer inline with the presentation of the Rumer language.

Contributions: This chapter contributes a relationship-based programming language that incorporates the novel language mechanisms member interposition and predicate references.

Chapter 4. In this chapter, we present the Matryoshka Principle. We begin with introducing the stratification layers underlying the principle and then detail the admissibility criteria imposed by the principle. The Matryoshka Principle gives rise to a number of system invariants, which are essential to the verification technique.

Contributions: This chapter contributes the Matryoshka Principle, a modularization principle providing strong encapsulation guarantees for a program's abstractions and prohibiting the occurrence of transitive call-backs.

Chapter 5. In this chapter, we develop a visible-state verification technique for the Rumer programming language. We first provide some background information about visible-state verification techniques and then present our verification technique and prove its soundness.

Contributions: This chapter contributes a visible-state verification technique for the Rumer language that leverages the Matryoshka Principle and member interposition.

Chapter 6. In this chapter, we extend the verification technique of the previous chapter with selective ownership. We first introduce selective ownership and then present the

extended verification technique and prove its soundness. We conclude the chapter with an overview of related work on object-oriented verification techniques.

Contributions: This chapter contributes a lightweight and flexible form of ownership that relies on the Matryoshka Principle for the prevention of transitive call-backs.

Chapter 7. In this chapter, we illustrate the Rumer visible-state verification technique based on a number of examples that constitute well-known specification and verification challenges for object-oriented program verification. We conclude with a discussion of future work.

Contributions: This chapter contributes an evaluation of the benefits of first-class support of relationships for program verification.

Chapter 8. This chapter concludes this thesis.

2 Relationships

Classes in object-oriented programs do not stand by themselves, but there exist numerous relationships between them. In this chapter, we discuss how programmers represent the relationships between classes in object-oriented programs (Section 2.1). We introduce a set of programming idioms that categorize possible implementations of relationships in object-oriented languages and present a survey on the occurrence of the programming idioms in real-world Java programs (Section 2.2). We then discuss known shortcomings of pure object-oriented implementations of relationships and demonstrate how programming languages with first-class support for relationships address those shortcomings (Section 2.3). We conclude the chapter with an overview of related work on first-class relationships (Section 2.4).

2.1 OO relationships

An important part in the design of an object-oriented system is the identification of the classes that make up the system. Equally important as the classes themselves are also the *relationships* that exist between the classes. Conceptual modeling languages, such as the Unified Modeling Language (UML) [31] or the Entity-Relationship (ER) model [32], account for this fact and provide useful abstractions that allow a system designer not only to express the classes but also the relationships between them. UML class diagrams support classes and associations to represent classes and their relationships, respectively. ER diagrams support entities and relationships to represent classes and their relationships, respectively.

To implement a system, programmers must map the abstractions of the conceptual model to the abstractions provided by the programming language. This mapping is straightforward for UML classes and ER entities as there is a direct correspondence between UML classes, or ER entities, and the classes of an object-oriented language. However, the mapping is less straightforward for UML associations and ER relationships since object-oriented languages lack a corresponding abstraction to capture the relationships between classes. As a result, programmers implement the relationships in terms of the primitives available in an object-oriented language. Relationships are most commonly implemented

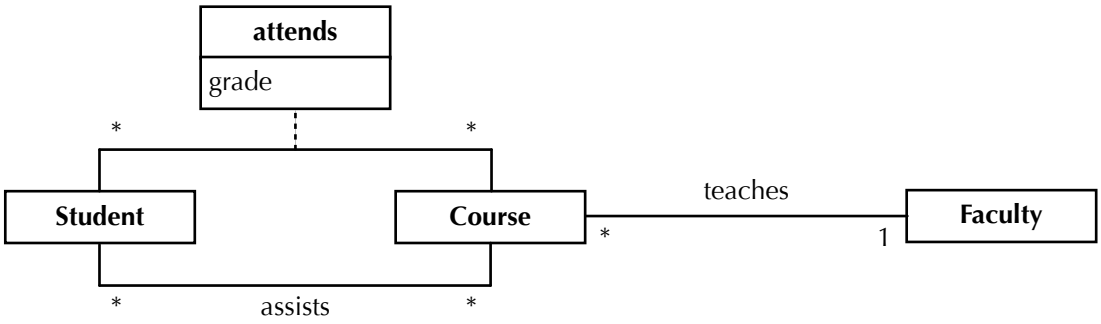

Figure 2.1: UML class diagram representing the information system of a university.

by declaring reference fields in one of the related classes. Depending on the desired features of the relationship implementation, the use of auxiliary classes may be necessary. For example, if enumeration of all related objects is required, an auxiliary class must be introduced that stores all related instances of the involved classes.

In this section, we first introduce a list of typically desired features for object-oriented implementations of relationships (Section 2.1.2) and then present a set of programming idioms that categorize concrete implementations of relationships in object-oriented languages (Section 2.1.3). For every programming idiom, we detail which features it supports. We illustrate the programming idioms based on a simple example (Section 2.1.1), which we introduce next.

2.1.1 Running example

We use the information system of a university as the running example of this chapter. Figure 2.1 shows the UML class diagram of the university system. The diagram consists of the classes Student, Course, and Faculty and the associations attends, assists, and teaches. Classes are represented by rectangles and associations as lines connecting the classes that the association relates. The diagram in Figure 2.1 indicates that students can attend as well as assist courses and that courses are taught by faculty members. Both classes and associations can declare attributes. The diagram in Figure 2.1 only displays an association attribute, but omits class attributes since they are not relevant for the current presentation. An association with an attribute is represented by an association class: a rectangle connected to the association by a dotted line. In Figure 2.1, attends is an association class and declares the association attribute grade. The attribute grade stores for each student and course pair the grade a particular student received for attending a particular course. The UML class diagram in Figure 2.1 declares also multiplicity constraints. Multiplicity constraints restrict the number of objects a particular object can be related to by an association. In the example, the multiplicities '*' and '1' are used, indicating that an object can be associated with any number of objects of the class to which the object is related by the association or to at most one object of the class to which the object is related by the association, respectively.

2.1.2 Relationship features

In this section, we present a list of *relationship features* that object-oriented implementations of relationships exhibit. The features are dependent on the particular kind of relationship considered. In this thesis, we only consider static relationships between classes that result from reference field declarations. These are the relationships that can be expressed as first-class relationships in programming languages with first-class relationship support and that are, consequently, of primary interest to this thesis. We ignore any form of temporary connections between instances of classes that result from method arguments or method-local references. We elaborated the relationship features in earlier work [33, 34]. They are based on Chen's entity relationship (ER) model [32, 35], but adapted to fit the object-oriented programming context. The relationship features are:

- **Degree:** Specifies the number of classes that a relationship relates.
- **Cardinality:** Prescribes, for all classes related by the relationship, the maximal number of instances with which an instance of the class can be related. As opposed to a UML multiplicity constraint, a cardinality constraint represents an upper bound rather than a precise number. For binary relationships, we distinguish 1:1, 1:m, and m:n cardinality constraints. For ternary relationships, we distinguish 1:1:1, 1:1:m, 1:m:n, and m:n:p cardinality constraints.
- **Relationship member:** Indicates whether a relationship implementation supports relationship members. A relationship member is a field or method that is associated with the relationship rather than with any of the individual classes that the relationship relates. A relationship field corresponds to a UML association attribute or an ER relationship attribute.
- **Extent:** Indicates whether a relationship implementation uses an auxiliary extent class which stores all instances of the related classes. An extent class is useful, for example, for enumerating all instances of the related classes.

The relationship features indicate that the degree of a relationship may be smaller than the number of actual classes occurring in a relationship implementation. We use different terminology to distinguish the classes that are actually related by the relationship, and that determine the degree of the relationship, from any auxiliary classes. We refer to the classes that a relationship relates as the relationship *participants*. On the other hand, we refer to the classes that occur in an object-oriented relationship implementation for supportive purpose only, but that are conceptually not part of the relationship, as *auxiliary* classes. Both participant classes and auxiliary classes build the *constituent* classes of a relationship implementation.

2.1.3 Programming idioms

In this section, we discuss the *programming idioms*. The programming idioms represent a categorization of concrete implementations of relationships in object-oriented languages. We derived the programming idioms in a multi-stage process [33, 34]: given the identified

Idiom	Degree/Cardinality							Support	
Configuration	binary			ternary				member	extent
	1:1	1:m	m:n	1:1:1	1:1:m	1:m:n	m:n:p		
DBU									
single-valued	-	●	-	-	-	-	-	○	-
multi-valued	-	-	●	-	-	-	-	-	-
DBB									
single-valued, single-valued	◐	-	-	-	-	-	-	○	-
single-valued, multi-valued	-	◐	-	-	-	-	-	○	-
multi-valued, multi-valued	-	-	◐	-	-	-	-	-	-
IIE									
single-valued	◐	●	-	-	-	-	-	-	●
multi-valued	◐	◐	●	-	-	-	-	-	●
IPE	◐	◐	●	-	-	-	-	●	●
ITE	-	-	-	◐	◐	◐	●	●	●

Table 2.2: Summary of the relationship features (see Section 2.1.2), realized by individual programming idioms. Symbols: '●' support of the feature, '◐' additional programmatic effort required for feature support, '○' conceptually suboptimal but technically doable, - no support.

relationship features (Section 2.1.2), we built a first set of possible programming idioms for implementing relationships. We then refined those idioms in an iterative process by inspecting their occurrence in larger object-oriented applications. We finally subjected the idioms to the external validation by Hans Wegener, a professional software developer, working at Swiss Re at the time.

Table 2.2 provides an overview of the programming idioms and indicates for each programming idiom the relationship features it supports. Full support of a feature is indicated by a '●' symbol. A '◐' symbol indicates that a feature is not inherently supported by a programming idiom, but a programmer can manually support the feature by supplying additional methods that include appropriate checking code. A '○' symbol indicates that a feature is unnatural to a programming idiom, but its support can technically be achieved. A '-' symbol indicates that a feature cannot be supported by a programming idiom. Table 2.2 distinguishes different *configurations* of a programming idiom. The configuration of a programming idiom is determined by the type declaration of the idiom's reference field(s). A reference field's type imposes an upper bound on the number of objects that the reference field can refer to. Table 2.2 classifies a configuration as *multi-valued* if several objects can be referred to, and as *single-valued* if at most one object can be referred to. To determine the configuration of a programming idiom, we treat reference field declarations of an array type, collection type[1], or map type as multi-valued reference fields and any remaining field declarations as single-valued reference fields.

The subsequent sections introduce the individual programming idioms and detail for each

[1] In this chapter, we use the term collection to denote one-dimensional data structures, such as sets and lists.

idiom which features it supports. Although the programming idioms were primarily devised for the purpose of evaluating how relationships are encoded in object-oriented programs, the programming idioms can also be useful for other purposes. For example, they can serve as an implementation reference for programmers, instructing programmers how to implement UML associations in object-oriented programming languages. In this respect, the programming idioms are related to Noble's relationships patterns [36] that consist of a set of design guidelines on how to represent the relationships between objects in object-oriented programming languages. On a broader scale, the programming idioms can also be regarded as kinds of design patterns [37], yet on a lower level of granularity.

Direct Binary Unidirectional (DBU)

Figure 2.3 depicts the UML class diagram of the *direct binary unidirectional (DBU)* programming idiom. This idiom represents a very straightforward implementation of a relationship: one of the two participant classes (A) declares a reference field (b) pointing to its partner participant class (B). The field can either be *single-valued* or *multi-valued*. In the case of a single-valued field, the type of the field is the type of the partner participant class (i.e., B)). In the case of a multi-valued field, the type of the field is either an array with the partner participant class as an element type (i.e., B[]) or a collection with the partner participant class as an element type (i.e., Collection[B]). The choice of the field type determines the cardinality of the relationship. The cardinality of the relationship is m:1 for single-valued reference field declarations and m:n for multi-valued reference field declarations. The cardinality is m:1 for a single-valued reference field b since, even though an A object can reference at most one B object, the referenced B object can be referenced by several A objects. The cardinality is m:n for a multi-valued reference field b since an A object can reference several B objects that, in turn, can be referenced by several A objects. As indicated by Table 2.2, the DBU programming idiom cannot support other cardinalities, not even by supplying appropriate checking code. To enforce more restrictive cardinalities (e.g., 1:1 for single-valued reference fields), an auxiliary extent class would need to be introduced that provides access to all instances of the participant classes, allowing a programmer to check those instances' compliance with the more restrictive cardinality constraint. Table 2.2 further indicates that the DBU idiom can support relationship members in the case of single-valued fields. Such a member must be declared at the many-side of the relationship (i.e., class A). This approach is most likely chosen in reality due to its practicability but unfortunately intermingles relationship members with participant members.

Example: Association teaches (Figure 2.1) can be implemented using the DBU idiom. To this end, a single-valued field of type Faculty is declared in class Course, making a course point to the faculty member teaching the course. This configuration of the DBU idiom guarantees that a course cannot be taught by several faculty members.

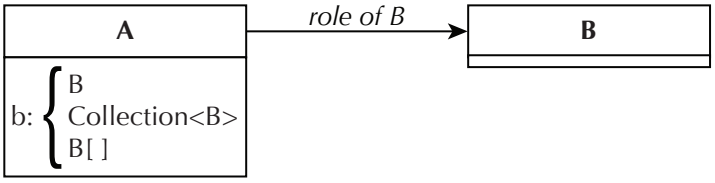

Figure 2.3: UML class diagram of the direct binary unidirectional (DBU) programming idiom.

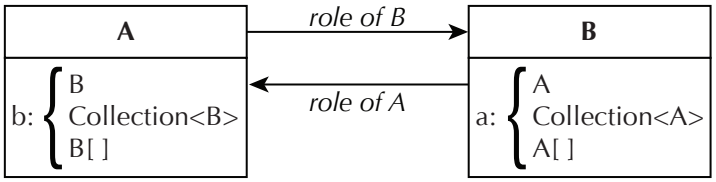

Figure 2.4: UML class diagram of the direct binary bidirectional (DBB) programming idiom.

Direct Binary Bidirectional (DBB)

Figure 2.4 depicts the UML class diagram of the *direct binary bidirectional (DBB)* programming idiom. This idiom attempts to maintain the bidirectionality inherent to relationships by declaring reference fields at both sides of the relationship. In Figure 2.4, the participant class A declares a reference to the participant class B, and vice versa. The fields can either be *single-valued* or *multi-valued*. As opposed to the DBU idiom, the existence of a "back-reference" in the DBB programming idiom, allows the enforcement of more restrictive cardinality constraints (see Table 2.2). For example, in the case of two single-valued reference fields, programmers can establish appropriate checks to guarantee that a B object that is referenced by an A object actually refers back to the A object. If no code is supplied to enforce cardinality compliance, all configurations of the DBB idiom degenerate to variants of m:n relationships where m and n can range from one to the number of instances of the respective participant. In the case of 1:1 and 1:m relationships and provided that the bidirectionality is appropriately enforced, the DBB idiom can also support relationship members. As indicated by Table 2.2, the declaration of relationship members in participant classes is not advisable since it results in intermingling participant members and relationship members.

Example: Association teaches (Figure 2.1) can alternatively be implemented using the DBB idiom. To this end, a single-valued field of type Faculty is declared in class Course and a multi-valued field with the element type Course (e.g., LinkedList<Course>) in class Faculty. The code assigning to those reference fields must include appropriate checks to guarantee that the Course objects referenced by a Faculty object refer back to the Faculty object.

Indirect Indexed Extent (IIE)

Figure 2.5 depicts the UML class diagram of the *indirect indexed extent (IIE)* programming idiom. As opposed to the direct programming idioms DBU and DBB, this idiom

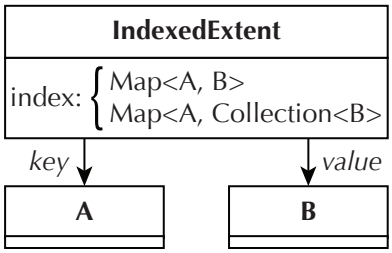

Figure 2.5: UML class diagram of the indirect indexed extent (IIE) programming idiom.

introduces a further indirection and uses an auxiliary class IndexedExtent in addition to the participant classes A and B. The extent class IndexedExtent stores all instances of the participant classes A and B. As opposed to the direct programming idioms DBU and DBB, the IIE idiom does not declare reference fields in the participant classes, but declares an index field in the extent class to relate the instances of the participant classes to each other. In a Java setting, the index is typically a descendant of Map and allows the retrieval of all values that are associated with a key. In Figure 2.5, map keys are single-valued and of type A and map values can either be single-valued or multi-valued and are of type B and Collection<B>, respectively. The cardinality of the relationship is 1:m for single-valued map values and m:n for multi-valued map values. Thanks to the existence of an extent class, more restrictive cardinality constraints than the inherent ones can be enforced.

Example: Association assists (Figure 2.1) can be implemented using the IIE idiom. To this end, an auxiliary class Assists is declared that represents the extent of the programming idiom. The extent class Assists declares an index field of type Map with a single-valued map key type Student and a multi-valued map value type with the element type Course (e.g., LinkedList<Course>).

Indirect Pair Extent (IPE)

Figure 2.6 depicts the UML class diagram of the *indirect pair extent (IPE)* programming idiom. The IPE programming idiom employs two auxiliary classes: an extent class Extent and a class Pair. Class Pair declares two single-valued reference fields a and b, referring to instances of the participant classes A and B, respectively. Conceptually, the auxiliary class Pair is a direct representative of the actual relationship. The extent class Extent stores all instances of class Pair in the multi-valued reference field pair. The inherent cardinality of the programming idiom is m:n, but thanks to the existence of an extent class more restrictive cardinality constraints can be enforced. Furthermore, the existence of class Pair, facilitates true support for relationship members. By declaring relationship members in class Pair, relationship members are associated with the referenced participant pair instances and distinguishable from participant members.

Example: Association attends (Figure 2.1) can be implemented using the IPE idiom. To this end, the auxiliary classes Attends and StudentCoursePair are declared that represent

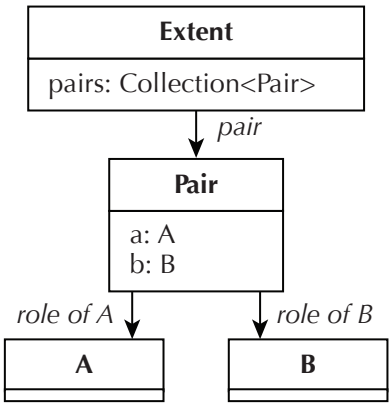

Figure 2.6: UML class diagram of the indirect pair extent (IPE) programming idiom.

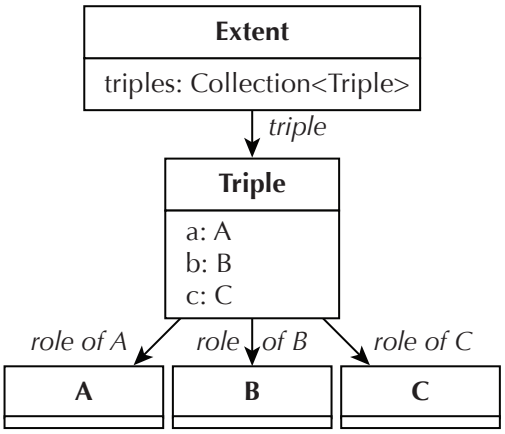

Figure 2.7: UML class diagram of the indirect triple extent (ITE) programming idiom.

the extent class and pair class of the programming idiom, respectively. The pair class StudentCoursePair declares single-valued reference fields of type Student and Course and the extent class Attends declares a multi-valued reference field with the element type StudentCoursePair (e.g., HashSet<StudentCoursePair>). Furthermore, the pair class StudentCoursePair declares the relationship attribute grade.

Indirect Triple Extent (ITE)

Figure 2.7 depicts the UML class diagram of the *indirect triple extent (ITE)* programming idiom, a generalization of the IPE idiom that supports ternary relationships. Like the IPE programming idiom, it introduces an auxiliary extent class Extent and represents the actual relationship explicitly by the auxiliary class Triple. Analogously, the inherent cardinality of the programming idiom is m:n:p, but more restrictive cardinality constraints can be enforced as well as relationship members declared.

2.2 Empirical study

To assess the prevalence of relationships in object-oriented programs, we conducted an empirical study on the occurrence of the programming idioms of Section 2.1.3 in today's object-oriented applications. The study is based on an analysis tool, the Relationship Detector for Java (RelDJ), that identifies occurrences of the programming idioms in Java code. The tool performs a static, fully automatic bytecode analysis and was implemented by Burns in her master's thesis [34]. Using the tool, we analyzed a corpus of 24 Java programs, ranging from the SPEC JVM98 [38] and the SPEC JBB2005 [39] benchmarks to open source Java projects, comprising up to thousands of classes and covering as diverse domains as compilers, games, and application servers.

In this section, we first introduce the corpus of Java applications that we analyzed in this study (Section 2.2.1). Then, we explain the underlying metrics of the study (Section 2.2.2) and present the empirical results (Section 2.2.3). We conclude by providing details on the analysis (Section 2.2.4).

2.2.1 Analysis corpus

Table 2.8 provides details on the analysis corpus of the empirical study. The analysis corpus consists of a total of 24 Java applications. It ranges from the SPEC JVM98 [38][2] and the SPEC JBB2005 [39] benchmarks to open source Java projects, comprising up to thousands of classes, and covers applications of various domains (e.g., application servers, graphical email clients, speech synthesizers, games, etc.). Additionally, the corpus comprises an in-house application that is a compiler for a subset of Java used as part of the compiler design lecture at ETH.

2.2.2 Metrics

We use several metrics to quantify the empirical results. An important concern in the choice of the metrics is to sustain validity of the empirical results. In particular, we took care not to introduce any bias. For example, to guarantee that the collected measurements are representative for a particular application, we excluded Java library classes from the analysis. This procedure is common practice [40, 41] since different Java applications typically share a considerable portion of library imports and, thus, a considerable portion of code. As a result, the empirical study only reports on relationships between application classes but not on any relationships between application classes and Java library classes.

We distinguish *basic metrics* from *aggregate metrics*. Aggregate metrics are computed from basic metrics. In addition, we use *corrective metrics* to account for possible imprecision of the basic metrics and to provide bounds on the range of realistic values. We

[2] We exclude the jvm_mtrt SPEC JVM98 benchmark from the analysis since the benchmark only consists of two classes, one of which is the main class.

Application	Version	Description	Classes	Bundle	Origin
azureus	3.0.5.0	BitTorrent client	1905	Binaries	SourceForge
columba	1.4	Graphical email client	1702	Sources	SourceForge
compiler	SS07	Compiler design sample solution	116	Sources	ETH internal
findbugs	1.3.2	Program to find bugs in Java code	1577	Sources	SourceForge
freetts	1.2.1	Speech synthesizer	189	Sources	SourceForge
gruntspud	0.4.6 beta	Graphical CVS client	746	Sources	SourceForge
jasperreports	2.0.4	Reporting tool	1209	Binaries	JasperForge
jbidwatcher	1.02.2	Auction site tracking tool	410	Sources	SourceForge
jboss	4.2.2 GA	JBoss applications server	6275	Binaries	SourceForge
jedit	4.3 pre13	Text editor	1008	Sources	SourceForge
jgraph	5.12.0.1	Graph visualization component	96	Sources	SourceForge
jhotdraw	6.0.1	Graphics framework for drawing editors	600	Binaries	SourceForge
jtopen	6.1	Toolbox for client/server programming	3600	Binaries	SourceForge
jvm_compress	JVM98	Modified Lempel-Ziv method	13	Binaries	SPEC JVM98
jvm_db	JVM98	Memory resident database	4	Binaries	SPEC JVM98
jvm_jack	JVM98	Parser generator	57	Binaries	SPEC JVM98
jvm_javac	JVM98	Java Compiler	177	Binaries	SPEC JVM98
jvm_jess	JVM98	Java expert shell system	152	Binaries	SPEC JVM98
jvm_mpegaudio	JVM98	Audio file decompression	56	Binaries	SPEC JVM98
jvm_raytrace	JVM98	Raytracer	26	Binaries	SPEC JVM98
lectcomm	1.0	ETH lecture communicator	268	Binaries	SourceForge
megamek	0.32.2	Battletech board game	876	Sources	SourceForge
pmd	4.2rc1	Java program analyzer	924	Sources	SourceForge
spec_jbb	JBB05	Three-tier client/server system	90	Binaries	SPEC JBB2005

Table 2.8: Analysis corpus consisting of 24 Java applications. Version: version number of analyzed application. Classes: total number of class files. Bundle: indicates whether sources or binaries were used. Origin: origin of application.

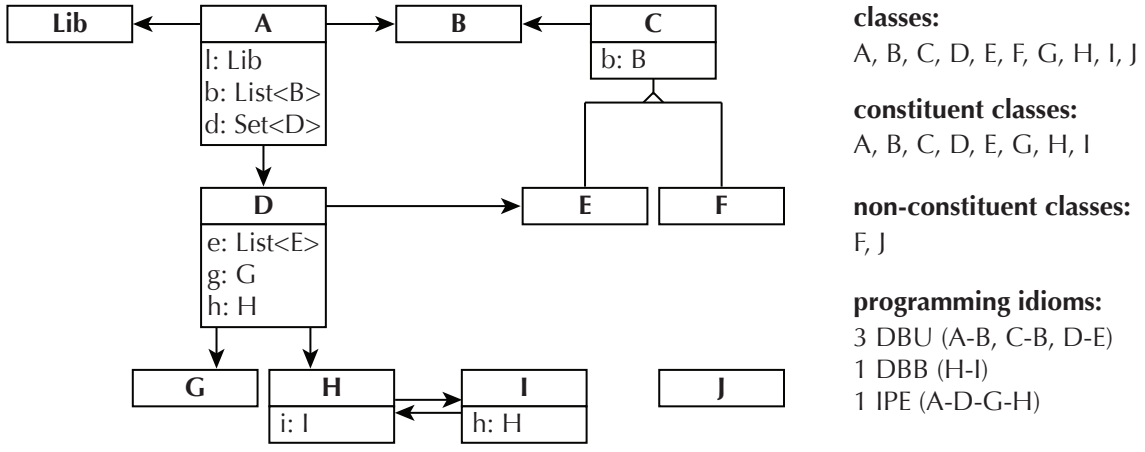

Figure 2.9: Synthetic example application to illustrate the metrics used for the empirical study. Left: UML class diagram of example application. Right: concrete measurements for example application.

introduce the metrics in the following subsections. We illustrate the metrics on the "synthetic" example application displayed in Figure 2.9. The figure shows the UML class diagram of the example application on the left, revealing the basic class structure of the application, and displays some measurements for the example application on the right.

Basic metrics

We distinguish the following basic metrics:

Total number of application classes (*#classes*). The total number of class files that constitute the analyzed application. As detailed previously, this metric excludes any Java library classes.

Example: The example application of Figure 2.9 consists of 10 classes, namely the classes A, B, C, D, E, F, G, H, I, and J. Since the class Lib is a Java library class, it is not considered for the analysis and not included in *#classes*.

Total number of programming idioms (*#idioms*). The total number of identified programming idioms. This metric reveals the number of relationships that exist between the application classes of the analyzed Java application.

Example: There are 5 occurrences of programming idioms in the example application of Figure 2.9. There are 3 instances of the DBU idiom (A-B, C-B, and D-E), 1 instance of the DBB idiom (H-I), and 1 instance of the IPE idiom (A-D, G-H). Again, library classes are ignored and, therefore, the relationship between the library class Lib and the application class A is not reported as an instance of the DBU programming idiom.

Total number of constituent classes (*#constituents*). The total number of application classes that are constituent classes of at least one programming idiom. A constituent class can either be a participant class or an auxiliary class of the programming idiom. Subclasses of constituent classes are not considered to be constituent classes.

Example: In the example application of Figure 2.9, 8 out of the overall 10 application classes are constituent classes of programming idioms. These are the classes A, B, C, D, E, G, H, and I. The remaining application classes F and J are not constituent classes. Class J may be a "driver" class, declaring the main method and maintaining temporary references to other classes only. Class F, on the other hand, is not considered to be a constituent class even though it is a subclass of the constituent class C.

Total number of lines of bytecode (*LOBC*). The total number of lines of bytecode of the analyzed application class files. The number includes: class signatures, method signatures, field declarations, and the bytecode instructions of method and constructor bodies.

```
 1 LinkedList<B> myBList;
 2
 3 void addB(B b) {
 4   doBookkeeping();
 5   myBList.add(b);
 6 }
 7
 8 B getB(B b) {
 9   int i;
10   i = myBList.indexOf(b);
11   doBookkeeping();
12   return myBList.remove(i);
13 }
14
15 void doBookkeeping() {...}
```

Figure 2.10: Illustration of computation of lower and upper bound of relationship-specific code.

Relationship-specific portion of LOBC (*R-LOBC*). The portion of the total number of lines of bytecode that is specific to the implementation of a programming idiom and, thus, relationship-specific. It is not possible to provide a precise measurement of that metric without the guidance of a human expert. A fully automated tool, such as RelDJ, can only identify those instructions that are directly related to the programming idiom implementation (i.e., declaration of reference fields and access to reference fields), but will miss any auxiliary instructions that are indirectly related. To provide a more realistic measure of the number of relationship-specific lines of bytecode, we use two metrics, the *R-LOBC lower bound* (*%lower*) and the *R-LOBC upper bound* (*%upper*). Those metrics indicate the bandwidth of the number of relationship-specific lines of bytecode. Whereas the lower bound only comprises reference field declarations and read and write instructions to those reference fields, the upper bound additionally comprises any residual instructions that occur in the method bodies where the read and write instructions happen. The actual number of relationship-specific lines of bytecode for an application lies in between its lower and upper bound. Both the lower and upper bound represent two extremes of the same measure as they are an underestimation and overestimation, respectively. A manual investigation of some of the small applications of the analysis corpus actually reveals that the lower bound is too conservative a metric and suggests the upper bound to be a more appropriate estimate of the actual number of relationship-specific lines of bytecode.

Example: We illustrate the two metrics based on the code shown in Figure 2.10 that comprises the member declarations of the class C of Figure 2.9. For simplicity, we use source code in the example, but the explanations apply analogously to bytecode. The R-LOBC lower bound metric comprises 4 instructions, namely the declaration of the reference field `myBList` on line 1 and the method invocations on the list object referenced by that field on line 5, line 10, and line 12. The R-LOBC upper bound metric, on the other hand, additionally considers the remaining instructions of the methods `addB()` and `getB()`, and comprises 7 instructions in total.

Aggregate metrics

This section introduces the aggregate metrics that are computed from the atomic metrics presented previously.

Percentage of non-constituent classes (%*islands*). The number of application classes that are not constituent classes of any programming idioms as a percentage of the total number of application classes. Equation 2.1 indicates how to compute this metric using the atomic metrics *#classes* and *#constituents* defined in Section 2.2.2.

$$\frac{\#classes - \#constituents}{\#classes} \tag{2.1}$$

Example: In the example application of Figure 2.9, 8 out of the overall 10 application classes are constituent classes of programming idioms. Thus, 2 out of the 10 application classes are not constituent classes, yielding the value $\frac{10-8}{10} = 20\ \%$, according to Equation 2.1.

Average involvement (*avgInvolvement*). Average number of programming idioms of which an application class is a constituent. Equation 2.2 indicates how to compute this metric. The formula computes the weighted sum of the number of times classes are constituent classes of programming idioms. The subscript *n* enumerates the number of programming idioms of which an application class is a constituent class. The numerator of the fraction, $\#classesInvolvedInNIdioms_n$, denotes the number of application classes that are constituent classes of the subscripted-number of programming idioms. The denominator of the fraction, *#classes*, denotes the total number of application classes, as defined in Section 2.2.2.

$$\sum_n \frac{\#classesInvolvedInNIdioms_n}{\#classes} * n \tag{2.2}$$

Example: In the example application of Figure 2.9, 2 application classes (i.e., F and J) are not constituent classes of any programming idioms, 4 applications classes (i.e., C, E, G, and I) are constituent classes of 1 programming idiom, and 4 applications classes (i.e., A, B, D, and H) are constituent classes of 2 programming idioms. Thus, for the example application, the metric has the following value: $\frac{2}{10} * 0 + \frac{4}{10} * 1 + \frac{4}{10} * 2 = 1.2$.

Average number of constituent classes (*avgConstituents*). Average number of constituent classes of all the identified programming idioms. Equation 2.3 indicates how to compute this metric. The formula computes the ratio between the cumulative number of constituent classes of the identified programming idioms and the total number of identified programming idioms. The subscript *idiom* enumerates the possible programming

idioms. Expression $\#occurrences_{idiom}$ denotes the number of occurrences of a particular programming idiom in the analyzed application and expression $\#constituents_{idiom}$ indicates the number of constituent classes of the particular programming idiom. The denominator of the fraction, *#idioms*, denotes the total number of identified programming idioms, as defined in Section 2.2.2.

$$\frac{\sum_{idiom} \#occurrences_{idiom} * \#constituents_{idiom}}{\#idioms} \tag{2.3}$$

Example: In the example application of Figure 2.9, there 3 occurrences of the DBU programming idiom, 1 occurrence of the DBB programming idiom, and 1 occurrence of the IPE programming idiom. Thus, for the example application, the metric has the following value: $\frac{3*2+1*2+1*4}{5} = 2.4$.

Corrective metrics

Unfortunately, it is not always possible to identify the classes that are constituent classes of programming idioms. The following factors complicate the identification of constituent classes: *(i)* the use of non-generic (i.e., raw) collections or maps and *(ii)* the occurrence of generic types that are not instantiated. In case *(i)*, RelDJ employs a data-flow analysis to infer the actual element type(s) of the collection or map (see also Section 2.2.4). If the type inference succeeds, then the constituent class(es) of the programming idiom can be identified and unified with the corresponding application class(es). Otherwise, the constituent class(es) of the programming idiom cannot be identified and, consequently, not be unified with any application class(es). In case *(ii)*, the constituent class(es) of the programming idiom can neither be identified nor unified with any application class(es).

Application classes that cannot be unified with constituent classes of programming idioms are not included in the *#constituents* metric. As a result, the reported values for *#constituents* are generally too low. We can use the number of unidentified constituent classes as an upper bound to give a more accurate value for the *#constituents* metric. We compute two metrics for that purpose:

Total number of unidentified constituent classes (*#unknowns*). The total number of constituent classes of programming idioms that cannot be identified and that, consequently, cannot be unified with existing application classes. This number can serve as an upper bound on the actual number of constituent classes: the actual number of constituent classes can be larger than the reported value *#constituents* by at most the value of *#unknowns*. The maximal value of *#constituents* + *#unknowns* is reached if each unidentified constituent class traces back to a different application class. On the other hand, if all unidentified constituent classes trace back to the same application class or, if there are no unidentified constituent classes, then the number of actual constituent classes is

#constituents + 1 or *#constituents*, respectively. The number of actual constituent classes thus lies between *#constituents* and *#constituents* + *#unknowns*. For instance, in Figure 2.9, if the classes B and E were not identified, then the value of *#constituents* would be 6 (i.e., the classes A, C, D, G, H, and I) and the value of *#unknowns* would be 3. The actual number of constituent classes in this example is 8, which is in-between 6 and 9 (i.e., $9 = 6 + 3$).

Percentage of unidentified constituent classes (%*unknowns*). The number of unidentified constituent classes as a percentage of the total number of application classes. Equation 2.4 indicates how to compute this metric.

$$\frac{\#unknowns}{\#classes} \tag{2.4}$$

2.2.3 Results

In this section, we discuss the results of our empirical study. We report on the overall frequency of relationships of the analysis corpus, on the occurrence of their different kinds of implementations, and on the portion of code consumed by relationship implementations.

Relationship frequency

Table 2.11 lists the number of identified programming idioms (column *#idioms*) per application of the analysis corpus. Since a programming idiom represents a concrete object-oriented implementation of a relationship, the number of identified programming idioms stands for the number of relationships existing in an application. Given the fact that each idiom consists of at least 2 classes, Table 2.11 reveals that some classes must be part of several relationships. Column *avgInvolvement* in Table 2.11 indicates the average number of programming idioms of which a class is a constituent class. The data are computed using Equation 2.2 introduced in Section 2.2.2. For some applications (e.g., jgraph, lectcomm, and spec_jbb), classes are involved in roughly 2.2 relationships on average. Although the average relationship involvement is less pronounced for other applications (e.g., jvm_jess and jvm_db with an average involvement of 0.6 and 0.8, resp.), the overall involvement for the analysis corpus amounts to 1.5 on average.

The average relationship involvement of a class varies from one class to another. Some classes are part of numerous relationships (up to 70) at a time and others are not part of a single relationship at all. Column *%islands* in Table 2.11 lists the number of application classes that are not constituent classes of any relationships as a percentage of the total number of application classes. The data are computed using Equation 2.1 introduced in Section 2.2.2. This number roughly correlates with the average involvement: the more island classes exist, the lower the average involvement turns out. As discussed in Section 2.2.2, the number of unidentified constituent classes establishes an upper bound on

Application	*#classes*	*#idioms*	*avgInvolvement*	*#constituents*	*%islands*	*%unknowns*
azureus	1905	1729	1.88	1648	13 %	25 %
columba	1702	1295	1.49	1182	31 %	13 %
compiler	116	60	1.15	61	47 %	7 %
findbugs	1577	1166	1.49	1151	27 %	14 %
freetts	189	122	1.42	116	39 %	14 %
gruntspud	746	764	2.00	725	3 %	15 %
jasperreports	1209	847	1.33	755	38 %	13 %
jbidwatcher	410	277	1.41	285	30 %	13 %
jboss	6275	4840	1.37	4228	33 %	17 %
jedit	1008	888	1.78	827	18 %	14 %
jgraph	96	99	2.20	95	1 %	28 %
jhotdraw	600	447	1.49	529	12 %	7 %
jtopen	3600	3222	1.58	2452	32 %	17 %
jvm_compress	13	5	1.15	9	31 %	15 %
jvm_db	4	2	0.75	2	50 %	25 %
jvm_jack	57	29	0.95	36	37 %	25 %
jvm_javac	177	116	1.31	99	44 %	27 %
jvm_jess	152	42	0.58	38	75 %	7 %
jvm_mpegaudio	56	44	1.55	44	21 %	9 %
jvm_raytrace	26	23	2.08	23	12 %	12 %
lectcomm	268	294	2.17	268	0 %	12 %
megamek	876	858	1.90	692	21 %	24 %
pmd	924	371	0.84	391	58 %	9 %
spec_jbb	90	91	2.26	72	20 %	19 %
Average	920	735	1.51	655	29 %	16 %

Table 2.11: Empirical results. *#classes*: total number of application classes (see Section 2.2.2). *#idioms*: total number of identified programming idioms (see Section 2.2.2). *avgInvolvement*: average number of programming idioms of which an application class is a constituent (see Equation 2.2). *#constituents*: total number of application classes that are constituent classes of at least one programming idiom (see Section 2.2.2). *%islands*: number of application classes that are not constituent classes as a percentage of total number of application classes (see Equation 2.1). *%unknowns*: number of unidentified constituent classes as a percentage of total number of application classes (see Equation 2.4).

the actual number of constituent classes and, thus, a lower bound on the number of island classes. Column *%unknowns* in Table 2.11 indicates the number of unidentified constituent classes as a percentage of the total number of application classes. For the applications jvm_javac, jvm_jack, jasper_reports, and compiler, for instance, the relatively high percentage of unidentified constituent classes suggests that a substantial portion of the classes that appear to be non-constituent classes actually are constituent classes of programming idioms. A manual investigation of the applications jboss and jvm_jess, which also exhibit a pronounced occurrence of island classes, reveals that the dominance of island classes results from a pronounced use of temporary relationships and from a deep inheritance structure, respectively. The application jvm_db, finally, only consists of 4 classes. One of the two island classes declares the main method and maintains temporary references to the instances of the other classes.

Relationship implementations

Table 2.12 indicates the frequency distribution of the individual programming idioms. Columns *%DBU – %ITE* indicate for each idiom the number of its occurrences as a percentage of the total number of identified programming idioms. The measurements show that the DBU programming idiom is by far the most frequently used idiom to implement relationships: 78 % of all discovered programming idioms of the analysis corpus are of this type. Considering the extra amount of declarations and boilerplate code necessary to set up any of the other programming idioms, this result is not surprising. The second most frequently used programming idiom of the analysis corpus is the IPE idiom: 11 % of all discovered programming idioms of the analysis corpus are of this type. Although the IPE idiom requires the programmer to set up and maintain 2 auxiliary classes apart from the participant classes, it is the binary programming idiom that supports most of the relationship features (see Table 2.2). The IPE idiom seems to be the programming idiom of choice whenever more functionality is required. The third most frequently used programming idioms are the IIE idiom and the ITE idiom. The occurrences of those idioms amount each to 5 % of all discovered programming idioms of the analysis corpus. The most uncommon programming idiom of the analysis corpus is the DBB idiom (1 %). Column *avgConstituents* in Table 2.12 summarizes these results by indicating the average number of constituent classes of all identified programming idioms. The data are computed using Equation 2.3 introduced in Section 2.2.2. The results show that a relationship is implemented on average with 2.41 constituent classes.

Relationship-specific code

The code necessary to implement relationships in class-based object-oriented programs represents a noticeable portion of a program. This code subsumes declarations of reference fields and corresponding setter and getter methods and, in the case of indirect programming idioms, the declaration of auxiliary classes. In addition to providing the code for the setup of a relationship, programmers must also provide the code necessary to preserve the consistency of a relationship. For instance, if support for bidirectionality is required and the DBB programming idiom is chosen, invocations of setter methods on either side of the relationship must be coordinated. Otherwise, the bidirectionality of the programming idiom, which is inherent to a relationship, is easily destroyed.

Column *LOBC* in Table 2.12 lists the total number of lines of bytecode for every application of the analysis corpus. Column *%lower* and *%upper* indicate the lower and upper bound, respectively, of the relationship-specific portion of lines of bytecode (see Section 2.2.2) as a percentage of the total number of lines of bytecode. The portion of code that is relationship-specific tends to increase with the occurrence of more complex programming idioms (i.e., DBB, IIE, IPE, and ITE) as well as with the average involvement of classes in relationships (column *avgInvolvement* in Table 2.11). For instance, the portion of relationship-specific code for azureus is considerable since the applica-

Application	%DBU	%DBB	%IIE	%IPE	%ITE	avgConstituents	LOBC	%lower	%upper
azureus	72 %	-	5 %	17 %	5 %	2.55	374627	22 %	59 %
columba	84 %	-	4 %	9 %	3 %	2.32	237132	25 %	56 %
compiler	77 %	5 %	2 %	17 %	-	2.35	32776	11 %	24 %
findbugs	75 %	-	8 %	13 %	5 %	2.47	312090	18 %	51 %
freetts	79 %	-	3 %	11 %	7 %	2.46	44991	14 %	41 %
gruntspud	83 %	1 %	-	12 %	5 %	2.38	144105	27 %	63 %
jasperreports	76 %	-	12 %	9 %	3 %	2.38	290135	21 %	47 %
jbidwatcher	80 %	1 %	8 %	7 %	4 %	2.34	140164	11 %	32 %
jboss	79 %	1 %	7 %	10 %	3 %	2.36	1423590	18 %	55 %
jedit	82 %	1 %	3 %	10 %	3 %	2.33	269756	21 %	54 %
jgraph	75 %	-	6 %	14 %	5 %	2.49	43371	21 %	52 %
jhotdraw	92 %	-	3 %	4 %	1 %	2.13	87788	14 %	31 %
jtopen	81 %	1 %	5 %	9 %	4 %	2.34	1413736	20 %	51 %
jvm_compress	40 %	-	-	40 %	20 %	3.40	1911	9 %	32 %
jvm_db	100 %	-	-	-	-	2.00	1686	13 %	48 %
jvm_jack	79 %	-	10 %	7 %	3 %	2.34	19895	14 %	31 %
jvm_javac	65 %	1 %	8 %	20 %	7 %	2.68	46126	18 %	45 %
jvm_jess	71 %	7 %	10 %	10 %	2 %	2.36	21255	18 %	45 %
jvm_mpegaudio	82 %	2 %	5 %	7 %	5 %	2.32	35026	4 %	10 %
jvm_raytrace	83 %	-	-	4 %	13 %	2.48	7054	19 %	55 %
lectcomm	85 %	2 %	3 %	8 %	1 %	2.23	35519	28 %	61 %
megamek	80 %	1 %	6 %	9 %	4 %	2.36	525996	21 %	60 %
pmd	77 %	1 %	10 %	11 %	1 %	2.36	195886	11 %	37 %
spec_jbb	78 %	-	1 %	15 %	5 %	2.48	46191	16 %	43 %
Average	78 %	1 %	5 %	11 %	5 %	2.41	239617	17 %	45 %

Table 2.12: Empirical results. *%DBU*, *%DBB*, *%IIE*, *%IPE*, *%ITE*: number of identified occurrences of respective programming idiom as a percentage of total number of identified programming idioms (see Section 2.2.2). *avgConstituents*: average number of constituent classes of all identified programming idioms (see Equation 2.3). *LOBC*: total number of lines of bytecode of analyzed application class files (see Section 2.2.2). *%lower*: lower bound of number of relationship-specific lines of bytecode as a percentage of total number of lines of bytecode (see Section 2.2.2). *%upper*: upper bound of number of relationship-specific lines of bytecode as a percentage of total number of lines of bytecode (see Section 2.2.2).

tion exhibits a relatively pronounced occurrence of more complex programming idioms (i.e., IPE and ITE) as well as a relationship involvement above average. The portion of relationship-specific code for jvm_compress, on the other hand, is below average. This result is surprising considering the relatively pronounced occurrence of complex programming idioms but is due to the poor relationship involvement of the jvm_compress classes. The occurrence of unidentified constituent classes (column *%unknowns* in Table 2.11) also has an adverse affect on the portion of relationship-specific code. For instance, the portion of relationship-specific code for jvm_compress is surprisingly low considering the relatively pronounced occurrence of the IPE and ITE programming idioms. This result is due to the occurrence of unidentified constituent classes. The average portion of relationship-specific code for the analysis corpus lies between 17% (lower bound) and 45% (upper bound) of the total number of lines of bytecode.

2.2.4 Analysis

In this section, we provide further details on the analysis tool used for executing the empirical study and give an overview of related code analysis approaches.

RelDJ

The code analysis tool for executing this empirical study is the *Relationship Detector for Java (RelDJ)*. RelDJ performs a static, fully automatic analysis of Java applications to detect occurrences of the programming idioms introduced in Section 2.1.3 and to collect measurements for the metrics defined in Section 2.2.2. RelDJ was implemented by Burns in her master's thesis [34]. It relies on the ASM Java bytecode manipulation framework [42] and performs a bytecode analysis.

Identification of programming idioms. The programming idioms introduced in Section 2.1.3 are recognizable by a tool since the individual idioms differ in implementation specificity. The differences are due to the disposition of reference field declarations in classes. To identify occurrences of the programming idioms in a Java application, RelDJ examines all reference field declarations in the application classes. Overall, RelDJ executes three passes over the input class files. In the first pass, RelDJ collects metadata about the program, such as the classes and their inheritance structure. In the second pass (see Algorithm 1), RelDJ analyzes each class separately and produces a preliminary identification of the programming idioms based on the local information available per class. In the third pass (see Algorithm 2), RelDJ completes the identification of the programming idioms and refines some of the identification decisions taken during the preliminary phase. For the identification of programming idioms, RelDJ treats nested classes (static nested classes and inner classes) and top-level classes alike but ignores the implicit "back-references" of inner classes to their enclosing classes.

Algorithm 1 details how RelDJ achieves a preliminary identification of programming idioms. The algorithm considers each class file in turn and maintains a set of variables to categorize the reference field declarations of a class. The variable singleValuedFields comprises the set of all single-valued reference field declarations of a class. The variables nonMapMultiValuedFields comprises the set of all multi-valued reference field declarations of a class that are not of a map type. The variables mapMultiValuedFields comprises the set of all multi-valued reference field declarations of a class that are of a map type. These variables are initialized for a particular class on lines 6 to 18. The algorithm excludes library classes from the analysis (see Section 2.2.2) and introduces a corresponding check on line 7. On lines 19 to 46 the algorithm achieves a preliminary identification of the ITE, IPE, IIE, and DBU programming idioms. The IIE programming idiom can be easily identified due to its declaration of a map field in constituent class IndexedExtent (see lines 20 to 23). The IPE and ITE programming idioms can be identified due to the declaration of single-valued reference fields in the Pair and Triple constituent classes, re-

```
Procedure PreIdentification(In classes, Out idioms):
 1: idioms ← { }   /* Identified idioms */
 2: for all class files c ∈ classes do
 3:    singleValuedFields ← { }   /* Relevant single-valued reference fields */
 4:    nonMapMultiValuedFields ← { }   /* Relevant multi-valued reference fields not of type map */
 5:    mapMultiValuedFields ← { }   /* Relevant multi-valued reference fields of type map */
 6:    for all fields f ∈ c do
 7:       if isOfApplicationType(f) then
 8:          if isSingleValued(f) then /* Check if f is a single-valued field */
 9:             singleValuedFields ← singleValuedFields ∪ f
10:          end if
11:          if isNonMapMultiValued(f) then /* Check if f is a non-map multi-valued field */
12:             nonMapMultiValuedFields ← nonMapMultiValuedFields ∪ f
13:          end if
14:          if isMapMultiValued(f) then /* Check if f is a map multi-valued field */
15:             mapMultiValuedFields ← mapMultiValuedFields ∪ f
16:          end if
17:       end if
18:    end for
19:    /* Indirect indexed extent (IIE) */
20:    for all fields f1 ∈ mapMultiValuedFields do
21:       iie ← initializeIIE(f1)
22:       idioms ← idioms ∪ iie
23:    end for
24:    for all methods m ∈ c do
25:       /* Indirect triple extent (ITE) */
26:       for all fields f1, f2, f3 ∈ singleValuedFields do
27:          if isAssignedIn(f1, f2, f3, m) then /* Check if f1, f2, f3 are assigned in m */
28:             ite ← initializeITE(f1, f2, f3)
29:             idioms ← idioms ∪ ite
30:             singleValuedFields ← singleValuedFields \ {f1, f2, f3}
31:          end if
32:       end for
33:       /* Indirect pair extent (IPE) */
34:       for all fields f1, f2 ∈ singleValuedFields do
35:          if isAssignedIn(f1, f2, m) then /* Check if f1 and f2 are assigned in m */
36:             ipe ← initializeIPE(f1, f2)
37:             idioms ← idioms ∪ ipe
38:             singleValuedFields ← singleValuedFields \ {f1, f2}
39:          end if
40:       end for
41:    end for
42:    /* Direct binary unidirectional (DBU) */
43:    for all fields f1 ∈ singleValuedFields ∪ nonMapMultiValuedFields do
44:       dbu ← initializeDBU(f1)
45:       idioms ← idioms ∪ dbu
46:    end for
47: end for
```

Algorithm 1: Algorithm for preliminary identification of programming idioms. This algorithm produces the input to Algorithm 2 and is invoked by Algorithm 2.

```
Procedure FinalIdentification(In classes):
 1: idioms ← { }
 2: PreIdentification(classes, idioms)
 3: for all idioms i1 ∈ idioms do
 4:    /* Check if i1 is a DBU idiom */
 5:    if instanceOf(i1, DBU) then
 6:       /* Direct binary bidirectional (DBB) */
 7:       for all idioms i2 ∈ idioms\{i1} do
 8:          /* Check if i2 is a DBU idiom and if B instance of i1 is A instance of i2 and vice versa */
 9:          if instanceOf(i2, DBU) and i1.B = i2.A and i2.B = i1.A then
10:             dbb ← initializeDBB()
11:             idioms ← idioms ∪ dbb
12:             idioms ← idioms \ {i1, i2}
13:          end if
14:       end for
15:       /* Indirect pair extent (IPE) */
16:       for all idioms i2 ∈ idioms\{i1} do
17:          /* Check if i2 is an IPE idiom and if B instance of i1 is Pair instance of i2 */
18:          if instanceOf(i2, IPE) and i1.B = i2.Pair then
19:             updateIPE(i2, i1)
20:             idioms ← idioms \ {i1}
21:          end if
22:       end for
23:       /* Indirect triple extent (ITE) */
24:       for all idioms i2 ∈ idioms\{i1} do
25:          /* Check if i2 is an ITE idiom and if B instance of i1 is Triple instance of i2 */
26:          if instanceOf(i2, ITE) and i1.B = i2.Triple then
27:             updateITE(i2, i1)
28:             idioms ← idioms \ {i1}
29:          end if
30:       end for
31:    end if
32: end for
```

Algorithm 2: Algorithm for final identification of programming idioms. This algorithm invokes Algorithm 1 to obtain a preliminary identification of programming idioms.

spectively. To distinguish an IPE programming idiom from an ITE programming idiom, assignments to the reference fields are tracked. Based on the assumption that the participant classes of a programming idiom are set up atomically, the algorithm inspects how many single-valued reference fields are assigned within a single method. If a method assigns to exactly three single-valued reference fields, the declaring class is categorized as a Triple constituent class of the ITE programming idiom (see lines 26 to 32). On the other hand, if a method assigns to exactly two single-valued reference fields, the declaring class is categorized as a Pair constituent class of the IPE programming idiom (see lines 34 to 40). The declaring classes of any residual single-valued reference fields as well as of any non-map multi-valued reference fields are finally categorized as constituent classes of DBU programming idioms (see lines 43 to 46).

The identification of programming idioms achieved by Algorithm 1 is preliminary since it does not cover the DBB programming idiom nor does it identify the Extent constituent

classes of the IPE and ITE programming idiom. In the set of programming idioms returned by Algorithm 1, an instance of the DBB programming idiom appears as two separate instances of the DBU programming idiom and instances of the IPE and ITE programming idiom appear as two separate instances of the IPE and ITE programming idiom, respectively, and one instance of the DBU programming idiom. For instance, for the example application displayed in Figure 2.9, Algorithm 1 would return the following programming idioms: 6 instances of the DBU programming idiom (A-B, C-B, A-D, D-E, H-I, and I-J) and 1 instance of the IPE programming idiom (D-G-H). Algorithm 2 refines the set of programming idioms returned by Algorithm 1 and identifies occurrences of the DBB programming idiom as well as the Extent constituent classes of the IPE and ITE programming idiom. It reconsiders each instance of a DBU programming idiom and tries to find a matching instance of a DBU programming idiom (see lines 7 to 14) or a matching instance of an IPE programming idiom (see lines 16 to 22), or a matching instance of an ITE programming idiom (see lines 24 to 30).

Element type inference. RelDJ employs an elementary *data-flow analysis* to infer the element type(s) of any raw collections or maps. As discussed in Section 2.2.2, this type inference is vital to identify the constituent classes of a programming idiom in case reference field declarations are of a collection or map type. The main idea of the analysis is similar to the one employed for the reverse engineering of UML associations [40, 41]: RelDJ monitors method invocations on raw collections or maps and deduces the element type(s) from any information provided as part of the invocation. For instance, the retrieval of an element type from a collection is typically accompanied by a cast instruction, revealing the actual element type. When an element is added to a collection, on the other hand, the type of the provided argument indicates the actual element type. The respective retrieval and addition methods of the Java library classes that RelDJ must consider for the type inference must be included in a configuration file. As opposed to previous approaches [40, 41], RelDJ is capable of inferring the least common supertype when several invocations with different actual argument types happen on the same collection or map. For that purpose, RelDJ uses the collected metadata about the class inheritance structure. The implemented data-flow analysis is of course only successful if methods are invoked on the raw collections or maps. If no methods are invoked, the corresponding constituent classes of a programming idiom cannot be identified, and RelDJ reports unidentified constituent classes.

Related approaches

The analysis underlying the presented empirical study is related to static model extraction and more loosely related to dynamic model extraction.

Static model extraction. Static model extraction targets the reverse engineering of UML class diagrams from an application's code. Whereas the reverse engineering of

UML class diagrams raises challenges similar to the ones of identifying programing idioms, there are fundamental differences between the two approaches. Most notable is the difference in the relevance of implementation details. Whereas a reverse engineering tool suppresses any implementation details on the recovered associations, our analysis exposes how relationships are represented in class-based object-oriented code. As a result, our analysis relies upon a more diverse set of implementation variations of relationships (e.g., bidirectional idioms or idioms with more than two constituent classes) than the ones covered by reverse engineering tools.

Jackson et al. [40] describe a tool Womble that allows the extraction of object models from Java programs. Womble performs a static bytecode analysis to identify UML associations. Like RelDJ, Womble applies a data-flow analysis to deduce the element types of collections. Unlike Womble, however, RelDJ takes advantage of generic declarations. Matzko et al. [43] contribute a similar reverse engineering tool for C++ and provide a comparison of a set of tools targeting the same goal.

Guéhéneuc and Albin-Amiot [44] provide clarification on the implementation of UML associations with the aim to bridge the gap between the design and the implementation of an object-oriented system. The authors define a set of formal properties of associations and use these properties to distinguish different kinds of binary class relationships. The authors also describe a tool suite to detect binary class relationships in class-based object-oriented code. Unlike Guéhéneuc and Albin-Amiot, we exclude temporary associations (established through local variables and method parameters) from the analysis, but cover bidirectional relationships and relationships that involve more than two classes.

Milanova [41] introduces an approach for identifying UML compositions in Java programs. The approach is based on a static inference of object ownership and aims to verify the implementation of UML compositions. The distinction between UML compositions and UML aggregations is not relevant for our empirical study.

Dynamic model extraction. More loosely related to our analysis is the research on dynamic object model extraction. Mitchell [45] focuses on the identification of run-time structures of object-oriented programs to infer any adverse affect the structures may have on memory footprint. Flanagan et al. [46] present a dynamic approach to extract object models from legacy code.

2.3 First-class relationships

In this section, we first summarize the results of the empirical study and discuss the deficiencies of object-oriented implementations of relationships. Then, we introduce the main ideas of first-class relationships.

2.3.1 Deficiencies of OO relationships

The empirical study on the occurrence of the programming idioms in today's Java applications reveals that relationships are an integral part of an application. On average, the number of relationships amounts to almost 80 % of the number of classes, and the average number of relationships in which a class is involved amounts to 1.51. The study also provides evidence that relationships do not remain encapsulated in a single class. On average, 21 % of all relationships distribute across at least 3 classes, with an average of 2.41 classes needed to implement a relationship. Lastly, the study demonstrates that a considerable portion of a program is concerned with the setup and maintenance of relationships. On average, between 17 % (lower bound) and 45 % (upper bound) of the total number of lines of bytecode is relationship-specific.

In the following, we discuss examples of object-oriented implementations of relationships and show that, especially for implementations supporting several relationship features (see Section 2.1.2), the resulting code *(i)* can be laborious, *(ii)* can compromise information hiding, and *(iii)* can be prone to error.

OO relationships can be laborious. Depending on the features that are desired for the relationship (see Section 2.1.2), programmers must declare auxiliary classes in addition to the actual participant classes of the relationship. For instance, Figure 2.13 sketches the implementation of relationship attends (Figure 2.1) based on the IPE idiom. The program declares the participant classes `Student` and `Course` as well as the auxiliary classes `StudentCoursePair` and `Attends`. The auxiliary class `StudentCoursePair` accommodates the relationship member `grade`, and the auxiliary class `Attends` facilitates the enumeration of all related `Student` and `Course` objects. The extent class `Attends` demonstrates that, in addition to the declaration of auxiliary classes, the declaration of auxiliary methods to set up and retrieve relationship participants is necessary. For instance, `Attends` declares methods to register and resign a student to (line 5) and from (line 6) a course and to set the grade that a student received for a course (line 7). Furthermore, `Attends` declares various methods (lines 10 to 12) to query the extent and to retrieve related `Student` and `Course` objects based on some search criteria. For instance, method `studentsAttendingCourse()` allows the retrieval of all students who attend the course passed as argument to the method. On the other hand, Method `studentsWithGradeForCourse()` allows the retrieval of all students who received the grade passed as argument for the attendance of the course passed as argument. For space concerns, Figure 2.13 does not show the implementations of the methods, but most of them involve iteration over the set `pairs` and the inspection of the current element to check whether it fits the selection criteria. Figure 2.13 exemplifies that object-oriented implementations of relationships can become quite involved and result in a considerable amount of boilerplate code. The writing (and maintenance) of the boilerplate code resulting from such implementations of more complex programming idioms is very laborious.

```
1 class Attends {
2   private HashSet<StudentCoursePair> pairs;
3
4   // set up
5   public void registerStudentForCourse(Student s, Course c) {...}
6   public void resignStudentFromCourse(Student s, Course c) {...}
7   public void setGrade(Student s, Course c, int g) {...}
8
9   // queries
10  public List<Student> studentsAttendingCourse(Course c) {...}
11  public List<Course> coursesAttendedByStudent(Student s) {...}
12  public List<Student> studentsWithGradeForCourse(int g, Course c) {...}
13 }
14
15 class StudentCoursePair {
16   private Student student;
17   private Course course;
18   private int grade;
19   // getters and setters
20 }
21 class Student {...}
22 class Course {...}
```

Figure 2.13: Illustration of relationship-specific methods for participant set up and retrieval. The code sketches the object-oriented implementation of association attends (Figure 2.1) based on the IPE programming idiom.

OO relationships can compromise information hiding. The object-oriented implementation of the Attends relationship in Figure 2.13 demonstrates that not all constituent classes of a relationship implementation should be exposed to client code. In the case of the program in Figure 2.13, class `Attends` defines the interface through which clients should set up and query the Attends relationship. Class `StudentCoursePair`, in contrast, is only declared for implementation purposes and should not be accessed by any clients. Unfortunately, the current implementation does not prevent clients form accessing class `StudentCoursePair`. By distributing the relationship code across several classes, the relationship is no longer encapsulated in a single class, and information hiding is thus compromised.

OO relationships can be prone to error. Loss of information hiding affects code maintenance adversely. For instance, let's assume for the running example that we impose the constraint that advanced courses can only be attended by graduate students and include appropriate checks in method `registerStudentForCourse()` (see Figure 2.13). As long as clients instantiate the Attends relationship using the methods declared by class `Attends`, the constraint is guaranteed to be preserved. However, as soon as clients instantiate the relationship directly by invoking the constructor of `StudentCoursePair`, the consistency of the relationship is endangered. The distribution of relationship code across several classes does not only compromise information hiding but also makes the relationship code prone to error. Figure 2.14 provides another example, illustrating the difficulties in preserving a relationship's consistency.

```
class Yin {
  private Yang yang;

  public void setYang(Yang y) {
    this.yang = y;
    yang.setYin(this);
  }

  public Yang getYang() {
    return yang;
  }
}
```

```
class Yang {
  private Yin yin;

  // access: package-private
  void setYin(Yin y) {
    this.yin = y;
  }

  public Yin getYin() {
    return yin;
  }
}
```

Figure 2.14: Code example illustrating the challenges of preserving a relationship's bidirectionality. The code is based on the DBB programming idiom.

It shows an instance of the DBB programming idiom that implements a bidirectional relationship between the classes `Yin` and `Yang`. As already noted by Rumbaugh [1], maintaining the bidirectionality of a relationship is intricate. In Figure 2.14, method `setYang()` attempts to preserve the bidirectionality of the relationship by updating the reference fields `yang` and `yin` simultaneously. Unfortunately, this precaution does still not guarantee that yin and yang objects mutually refer to each other. Direct invocations of method `setYin()` by any of the client classes residing in the same package as `Yang`, in particular, compromise the bidirectionality of the relationship. Limiting the access of method `setYin()` even further and changing its modifier from `package-private` to `private` does not alleviate the problem as it would prevent `setYang()`'s invocation of `setYin()`. Only a mechanism, such as friends in C++ to grant `Yin` write access to `Yang`'s fields, would prevent this kind of invalidation, but compromise encapsulation of the relationship participant `Yang`. But even if clients set up the relationship through invocations of `setYang()` only, the bidirectionality of the relationship is still at risk. Several invocations of `setYang()` on different receiver yin objects, but with the same argument yang object, in particular, would destroy the relationship's bidirectionality. Only appropriate checks or method specifications that prevent updates of the fields `yang` and `yin` if the argument yang object's `yin` field is `null` would prevent the invalidation.

2.3.2 Toward first-class relationships

Given the deficiencies of object-oriented implementations of relationships and the fact that relationships occur frequently in today's object-oriented applications, it is no surprise that various researchers have suggested to add first-class support for relationships to object-oriented programming languages. We review the related work on first-class relationships in Section 2.4. In this section, we briefly outline the main ideas of first-class relationships based on the running example.

The basic idea of first-class relationships is to represent the relationships between the classes of an object-oriented system explicitly in a programming language. A programming language with first-class relationship support thus allows programmers to declare

```
1  class Student {...}
2  class Course {...}
3  class Faculty {...}
4
5  relationship Attends (many Student, many Course) {
6    int grade;
7  }
8
9  relationship Assists (many Student, many Course) {...}
10
11 relationship Teaches (one Faculty, many Course) {...}
```

Figure 2.15: Implementation of running example (see Figure 2.1) with first-class relationships. The code uses features of RelJ [3] and/or the Data Structure Manager (DSM) [1]. See also Figure 2.13 for the pure object-oriented implementation of relationship Attends.

both the classes of an object-oriented system and the relationships between them. Like classes, relationships are first-class citizens: they can declare their own fields and methods, they can be instantiated, and their instances can be stored in variables, passed as arguments to method invocations, and returned as results of method executions.

Figure 2.15 sketches the implementation of the running example (see Figure 2.1) in a programming language that supports first-class relationships. The pseudo code shown in Figure 2.15 combines features of various first-class relationship approaches. The program declares the classes `Student`, `Course`, and `Faculty` as well as the relationships `Attends`, `Assists`, and `Teaches`. Relationship declarations indicate the types of the instances they relate. For instance, relationship `Attends` declares that it relates instances of the classes `Student` and `Course`. In the example, only classes appear as participants of relationships. In principle, relationships can relate both classes and/or relationships. Relationship declarations can also restrict the cardinality of a relationship. The cardinality constraints declared in Figure 2.15 match the UML multiplicities shown in Figure 2.1. For instance, the `Teaches` relationship restricts a course to be taught by at most one faculty member.

The program in Figure 2.15 demonstrates that relationship-based implementations are more "lightweight" than their object-oriented counterparts (see Figure 2.13 for the object-oriented implementation of relationship Attends). Most apparent is the decrease in declarations. Programming languages with first-class relationship support provide built-in constructs for relationship set up and instance retrieval. As a result, the declaration of auxiliary extent classes (e.g., `Attends` in Figure 2.13) to store related objects is unnecessary. Likewise, the declaration of auxiliary methods (e.g., set up and retrieval methods in Figure 2.13) to enumerate and query related objects is unnecessary. Besides the decrease in code size, relationship-based implementations restore modular reasoning for relationships as relationships are encapsulated in one program unit. Modular reasoning eases program maintenance since program changes become more local and are thus less likely to accidentally introduce errors in other parts of the program.

2.4 Related work

In this section, we review existing relationship-based programming languages and related attempts to capture the interdependencies between objects in a programming language. In Chapter 3, we introduce the relationship-based programming language Rumer that we elaborated in this thesis. We delimit the differences between the existing relationship-based languages discussed in this section and Rumer in Chapter 3, along with the introduction of the Rumer language concepts.

Relationship-based programming languages. Rumbaugh [1] first pointed out the importance of relations (relationship) as a semantic construct for object-oriented programming. He introduces an object-oriented programming language, the Data Structure Manager (DSM), that complements classes with relations. A relation in DSM, is an ordered list of an arbitrary number of classes. The index of a class in a relation can be denoted by a role name. DSM allows programmers to restrict the cardinality of a relation and also supports operations to add and remove instances of relations to and from the set of all instances of a relation or to iterate over such sets. While relations in DSM are explicit, they are not yet at the same level as classes. For example, relations can neither declare attributes nor methods. Besides contributing the programming language DSM, Rumbaugh provides an experience report on the representation of relations in object-oriented languages.

Albano et al. [2] propose an object-relationship data model that supports both classes and associations (relationships) to represent classes as well as the relationships between them. The data model is embodied in a statically and strongly-typed object-oriented database programming language. An important characteristic of the work is the availability of a rich range of constraints (e.g., disjointness constraints for classes, cardinality constraints for associations, and surjectivity and referential constraints for associations and their participating classes). The constraints are expressed declaratively and checked dynamically. Constraint checking is implemented based on the nested transactional support of the database programming language and guarantees atomic treatment of constraints (e.g., surjectivity) that can only be preserved by a sequence of operations. Similarly to Rumbaugh [1], Albano et al. also take a "set-oriented" approach: classes and associations are perceived as sets and relations, respectively, which are explicitly populated and depopulated by the programmer. The language supports relational bulk data operators to manipulate such sets of classes and associations. As opposed to Rumbaugh [1], Albano et al. allow associations to declare their own attributes. Furthermore, they introduce a minimal kernel language that expresses the complete object-relationship data model. The kernel language employs associations as its primitive abstraction and expresses classes as unary associations.

Bierman and Wren [3, 5] provide a formal treatment of first-class relationships in object-oriented programming languages and contribute the type system and operational seman-

tics of RelJ, a Java-like language with support for first-class relationships. As opposed to previous works, relationships in RelJ are at the same level as classes and feature not only attributes and methods but also advanced concepts, such as relationship inheritance. As opposed to class inheritance, relationship inheritance is based on a restricted form of delegation. Delegation determines the look-up of inherited relationship fields: a relationship instance passes (delegates) all look-up requests for inherited fields to its supertype relationship instance that exists between those same objects. In order for delegation to work, Bierman and Wren establish the invariant that corresponding supertype relationship instances exist for all subtype relationship instances. Bierman and Wren also discuss the addition of UML multiplicity constraints to RelJ. They outline a restricted, static approach to enforcing UML multiplicity constraints based on `one` and `many` annotations. The approach automatically results in the removal of any conflicting, existing relationship instances upon creation of a new relationship instance. In this context, Bierman and Wren point out a tension between the invariant that triggers implicit instantiation of supertype relationships and the implicit removal of relationship instances for multiplicity constraint enforcement. The implicit instantiation of a supertype relationship is problematic for supertype relationships that restrict the multiplicity of at least one participant to `one`. As soon as different subtype relationships are instantiated that involve different objects at the `one`-side, the implicit creation of the corresponding supertype relationship instances puts the multiplicity of the supertype relationship at risk. To prevent invalidation of a relationship's multiplicity, Bierman and Wren suggest not only the removal of the conflicting supertype relationship instance but also of all subtype relationship instances that exist between those same objects.

Wren [5] complements the class-based exploration of first-class relationships conducted in [3, 5] with an object-based calculus for heap queries. As opposed to RelJ, this calculus does not rely on an explicit abstraction for relationships but allows relationships to be expressed in terms of heap queries. To sustain this claim, Wren provides a translation of RelJ into the object-based calculus.

Nelson et al. [6] introduce an object-oriented programming language with support for first-class relationships that relies on a three-tiered model. In that model, objects are represented by one tier and relationships between objects are represented by two separate tiers, the link tier and the relationship tier. Whereas links are state-less object tuples, relationships denote groups of links and augment those links with state and behavior. Nelson et al. were the first to propose explicit instantiation of such relationships, allowing programmers to distinguish different collection instances of a relationship type. This proposal innovates over previous work which only permits singleton collections of a relationship type.

Liu and Smith [4] describe Classages, an object-oriented programming language designed to provide better support for expressing object interactions. Similarly to a class, a classage declares fields and methods. In addition, a classage specifies a separate interaction interface to capture the various collaborations in which an instance of a classage (called

objectage) may participate. There are various kinds of interaction interfaces available (e.g., connector, plugger, and mixer) to support the various kinds of collaborations that may arise between objects (e.g., whole-part-communication, peer-to-peer communication). As opposed to work on programming languages that provides explicit support for first-class relationships, Classages does not provide a separate language abstraction for relationships. However, the expression of relationships in Classages would definitively benefit from the availability of interaction-specific interfaces.

The Rumer programming and specification language differs in many aspects from existing relationship-based programming languages. We provide a more detailed comparison along with the introduction to Rumer in Chapter 3, but briefly highlight the main differences at this point. The resulting design of the Rumer programming language reflects our efforts to grasp the genuine characteristics of first-class relationships. Rather than extending an object-oriented programming language with first-class relationships, we elaborated a programming language that employs first-class relationships as its primary abstraction. This slight change in intention importantly shapes the resulting programming language. Most remarkable is the role references take in Rumer. Whereas references in traditional, object-oriented languages, are key to determining a program's heap structure, references lose this ability in Rumer. Instead, heap structures in Rumer are captured explicitly using first-class relationships. In this thesis, we leverage this structure-giving property of first-class relationships for program construction and program verification. Besides this difference in underlying intention, Rumer also differs from existing relationship-based programming languages in the range of supported language features. Unlike other relationship-based programming languages, Rumer supports Design-by-Contract-style [21] assertions (see Section 3.2.5 on page 55), side-effect free heap queries (see Section 3.2.3 on page 49), and member interposition, a mechanism to express relationship-dependent properties of objects (see Section 3.2.4 on page 51).

Relationship libraries. Pearce and Noble [47, 48] contribute the Relationship Aspect Library (RAL), a library facilitating the implementation of first-class relationships in a purely object-oriented language. RAL leverages aspect-oriented programming [49, 50] and represents relationships as separate abstractions using aspects. The relationships that can be implemented using RAL are similar in expressiveness to RelJ [3, 5] relationships. For example, both approaches require that relationships are explicitly populated and depopulated by the programmer and both approaches support relationship inheritance. In addition, Pearce and Noble provide a rich range of predefined relationship kinds and the possibility to declare fields for relationship participants using Aspect/J's inter-type declarations [51]. The predefined relationships, for example, support the declaration of UML multiplicity constraints, which are monitored at run-time.

Østerbye [52] contributes the Noiai association library that provides a reusable class library facilitating the implementation of first-class relationships in a purely object-oriented setting. As opposed to Pearce's and Noble's [47, 48] relationship library RAL, which is based on aspects, the Noiai association library is implemented in C♯. The features sup-

ported by the Noiai association library are similar in expressiveness to the ones supported by RAL; however, Noiai also supports advanced operators (i.e., Composition, Inverse, and Closure) to obtain new (read-only) associations from programmer-defined associations.

Bodden et al. [53] generalize the library-based approach of Pearce and Noble [47, 48] to use relational tracematches. Relational tracematches are based on tracematches [54], an extension of Aspect/J [51] to abstract over the execution history of an Aspect/J program. Relational tracematches represent aspect-based language features for executing an advice after having matched a sequence of events. The new implementation scheme supports n-ary relationships (a feature not supported by Pearce and Noble) and furthermore prevents name clashes that may arise under certain circumstances in Pearce and Noble's approach [53].

Relation types. Vaziri et al. [55] introduce relation types for the declarative definition of object identity. A relation type resembles a class in Java but entails a well-defined semantics for instance equality. Instance equality is no longer determined by the implementations of the methods `equals()` and `hashCode()` but by a number of immutable key fields declared by the relation type. Vaziri et al. provide a formalization of their equality model based on Featherweight Java [56] and a prototype compiler that extends Java with relation types. Although devised for a different purpose, relation types bear a resemblance to first-class relationships. Namely, if the key fields of a relation type are reference type fields, then a relation type defines a relationship.

Nelson et al. [57] build on Vaziri et al.'s work [55] and introduce Affinity, an untyped, object-oriented programming language with a new equality operator based on EGAL [58] and key fields. Similarly as in Vaziri et al., relationships can be implemented in Affinity by declaring a class with reference type key fields.

Design relationships. Noble and Grundy [59, 36] describe ways of persisting relationships from the modeling to the implementation stage of object-oriented development. The authors provide valuable guidelines that detail how to encode relationships in terms of the primitives of an object-oriented language so that they remain distinguishable as relationships in the resulting code. Their approach is purely class-based and does not consider language support for relationships.

Specification relationships. Jaspan and Aldrich [60] introduce relationships as an abstraction to specify the proper interactions between frameworks and their plugins. They contribute FUSION (Framework Usage SpecificatIONs), an annotation-based specification language to constrain the interactions between frameworks and plugins. In FUSION, relationships can be defined between objects, and method signatures are annotated to describe what effects method executions have on the relationships. In addition, FUSION allows the specification of propositional constraints on relationships. Those constraints

are expressed as class-level annotations and may be temporal, involve several objects, and depend on run-time values. The authors further contribute three variants (sound, complete, and pragmatic) of a static analysis to detect constraint violations in plugins, an implementation of the pragmatic analysis, and an empirical evaluation of this analysis in the context of the ASP.NET and Eclipse JDT frameworks.

Roles. Reenskaug [61] introduces role models to describe the structure of cooperating objects along with their static and dynamic properties. Role models are purely conceptual and focus on message-based interactions. However, they could assist in the identification of relationships during system design since relationships can be regarded as representations of particular role models. Riehle and Gross [62, 63] generalize role modeling for framework design and provide a unifying treatment of class-based modeling and role modeling.

Steimann [64] provides a comprehensive treatise of the concept of a role by surveying its occurrence in conceptual modeling languages and object-oriented programming languages. From this survey, Steimann derives a list of desired properties of a role, whereby some properties may conflict with each other. Steimann's list provides a valid foundation to assess any approach that supports roles (or a related abstraction) as an explicit concept.

Whereas Reenskaug [61] as well as Riehle and Gross [62, 63] employ roles as purely conceptual notions during software design, Herrmann et al. [65, 66] provide explicit support for roles at the level of the programming language. Herrmann et al. describe ObjectTeams, a programming language based on Java that centers on the notion of an Object Team. An Object Team is a module that combines a number of role-playing objects and that can be instantiated. The focus of Herrmann's et al.'s work is the a-posteriori integration of collaborations into existing systems. To this end, the language allows programmers to forward method calls from teams to base classes and offers advice-like constructs, known from aspect-oriented programming [49], to override methods of base classes.

Like Herrmann et al. [65, 66], Baldoni et al. [67] also provide explicit support for roles at the level of the programming language. They introduce powerJava, an extension of Java that allows a programmer to represent the individual roles an object may assume as Java inner classes. The role model supported by powerJava is particularly suited for the implementation of session-based applications (e.g., access control, web services) where the individual interactions between objects are represented by roles.

Pradel and Odersky [68] contribute a library for the Scala programming language [69] that supports roles as an explicit abstraction. The library-based approach is lightweight but provides all the necessary constructs to capture roles as well as the collaborations between roles. It leverages dynamic proxies to relate the various roles of an object to the role-playing object and to provide a matching identity both for an object and its roles.

Rumer: a relationship-based programming language

3

Rumer is a relationship-based programming language with Design-by-Contract-style assertions that provides programmers with appropriate abstractions to succinctly represent both the objects of real-world systems and the relationships between them. In this chapter, we first discuss the rationale underlying the design of the Rumer programming language (Section 3.1) and then introduce the programming language itself. We present Rumer's main language concepts (Section 3.2), its type system (Section 3.3) and semantics (Section 3.4). To gain practical experience with first-class relationships, we designed and implemented a prototype compiler for the Rumer programming language that supports run-time checking of Design-by-Contract-style assertions. We built and successively refined the compiler along with the development of the Rumer language.

3.1 Rationale

Once the relationships between classes are made explicit in a program using first-class relationships, new opportunities for program *modularization* arise. Thanks to first-class relationships, programmers are given the possibility to structure a software system not only from the view of an individual object but also from the view of the relationship in which the object participates.

Figure 3.1 and Figure 3.2 contrast the resulting run-time structure of an object-oriented implementation of the running example (see Section 2.1.1 on page 8) with the one of a relationship-based implementation, respectively. Figure 3.1 shows a snapshot of the program heap of the object-oriented implementation. It represents objects as dark gray circles. The program heap thus consists of the `Student` objects Susan, Paul, Alice, and John; the `Course` objects Math, Art, Compiler, and Programming; and the `Faculty` objects Franklin Wong, Jennifer Wallace, and Ramesh Narayan. The implementation is based on the DBU programming idiom (see Section 2.1.3 on page 9)[1]; the resulting unidirectional references are depicted by arrows in Figure 3.1. Figure 3.2, on the other hand, shows the corresponding snapshot of the program heap of a relationship-based

[1]Class `Student` actually declares a multi-valued reference field (e.g., `LinkedList<Course>`) to refer to the courses assisted by a student. Since irrelevant to the current discussion, Figure 3.1 omits this detail.

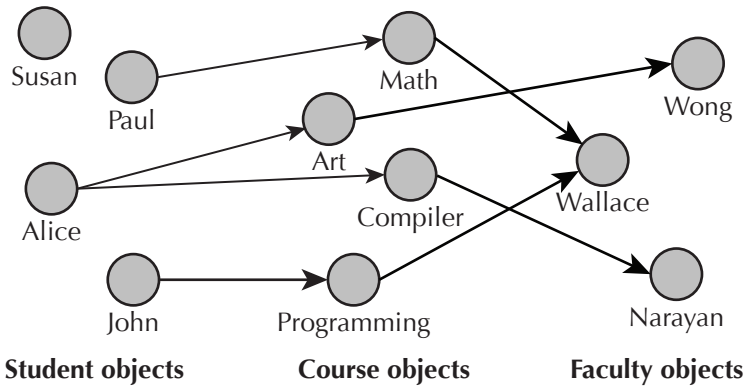

Figure 3.1: Abstract view of program heap of object-oriented implementation of Assists and Teaches relationships of running example (see Section 2.1.1 on page 8).

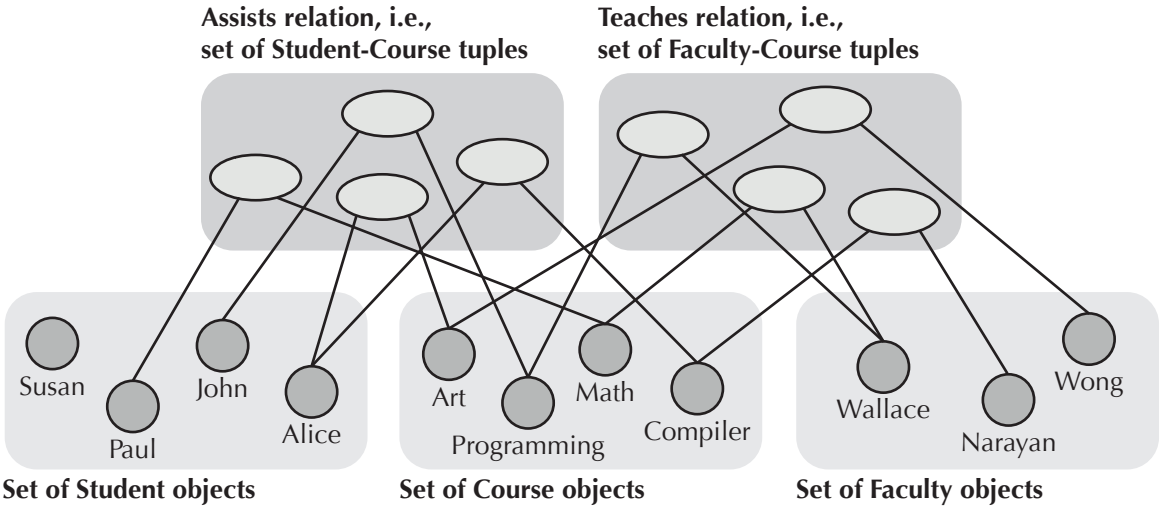

Figure 3.2: Abstract view of program heap of relationship-based implementation of Assists and Teaches relationships of running example (see Section 2.1.1 on page 8).

implementation of the running example. The relationship-based program heap consists of the same `Student`, `Course`, and `Faculty` objects as the object-oriented heap. In addition, the relationship-based heap comprises instances of the relationships `Assists` and `Teaches`. Figure 3.2 represents relationship instances as light gray ellipses that are connected by lines to the objects they relate. For example, there are two `Assists` relationship instances that connect the `Student` object Alice to the `Course` objects Art and Compiler, indicating that Alice assists both the courses Art and Compiler.

Given the separate run-time abstraction of a relationship instance, relationship-based programming languages introduce a layer of indirection between objects and restore the *bidirectional* nature of relationships. In the design of the Rumer programming language we drew from this observation. To facilitate access of the instances residing at either side of the relationship, Rumer supports appropriate retrieval operators. Since objects can participate in several relationships, those operators must be capable of dealing with sets of objects. To cover the complete spectrum of set-based operators, we designed Rumer so as to support a *set-oriented* view of a program's run-time structure. To this end, the Rumer

system maintains explicit collections of the various instances of a Rumer type. Figure 3.2 represents those explicit collections of instances as rounded rectangles, and Rumer allows them to be *queried* for instance retrieval.

To reason about Rumer query operators, we use the abstractions of a *set* and a *relation* to capture the explicit collections of instances in Figure 3.2. A relationship instance is abstracted as an *object tuple* since it relates objects. The heap of a relationship-based program can thus be abstractly described by the sets of objects of the declared classes and the relations (i.e., sets of object tuples) between those sets of objects. For example, the heap snapshot depicted in Figure 3.2 can be abstracted by the sets:

$$\begin{aligned} \mathit{Student} &= \{\mathit{Susan}, \mathit{Paul}, \mathit{John}, \mathit{Alice}\} \\ \mathit{Course} &= \{\mathit{Art}, \mathit{Program}, \mathit{Math}, \mathit{Compiler}\} \\ \mathit{Faculty} &= \{\mathit{Wallace}, \mathit{Narayan}, \mathit{Wong}\} \end{aligned}$$

and the relations:

$$\begin{aligned} \mathit{Assists} &= \{\mathit{Paul} \mapsto \mathit{Math}, \mathit{John} \mapsto \mathit{Program}, \mathit{Alice} \mapsto \mathit{Art}, \mathit{Alice} \mapsto \mathit{Compiler}\} \\ \mathit{Teaches} &= \{\mathit{Wallace} \mapsto \mathit{Program}, \mathit{Wallace} \mapsto \mathit{Math}, \mathit{Narayan} \mapsto \mathit{Compiler}, \mathit{Wong} \mapsto \mathit{Art}\} \end{aligned}$$

Given the abstractions of a set and relation, Rumer query operators can leverage the theoretical underpinning of relational database [70] and support the full range of relational query operators. For example, to retrieve all the `Student` objects that assist courses, the projection operator can be applied to the `Assists` collection. The abstractions of a set and relation also build the basis of the *relational model* that we introduced in [71]. We used this model as a meta model during the design of the Rumer programming language. The relational model allows a Rumer program to be captured in terms of the constructs of discrete mathematics. For example, the running example can be abstracted by the following model:

$$\begin{aligned} \mathit{Assists} &\subseteq \mathit{Student} \times \mathit{Course} \\ \mathit{Teaches} &\subseteq \mathit{Faculty} \times \mathit{Course} \end{aligned}$$

The existence of a mathematical model of a program is essential for reasoning about the program and, ultimately, verifying the program. Facilitating static *program verification* is also the ultimate goal that guided us in the design of the Rumer programming language. To facilitate program verification, the Rumer language enforces certain principles that follow naturally from first-class relationships. For example, the use of explicit collections to capture sets of objects and sets of object tuples proves beneficial to speak about *disjoint regions* of the program heap, and the Rumer language semantics thus guarantees that the resulting heap partitions remain disjoint. Also, the availability of *side-effect free* query operators proves valuable in the expression of assertions on a program's state, and

the Rumer type system and language semantics thus enforces purity of Rumer query expressions.

In Rumer, references lose their significance in building the object graph. Instead, the heap structure is built declaratively using first-class relationships. References also lose their importance in navigating the object graph since instances in Rumer can be accessed by means of queries. These observations raise the legitimate question whether references are needed after all in Rumer. In an early version of the Rumer programming language, we deliberately chose not to support references, to probe the practicability of such an approach. The nonexistence of references did actually not rule out any software designs. In some cases, the set-oriented take even rendered the resulting implementation more lightweight and practicable. In other cases, it turned out to be more convenient to refer to a particular instance rather than having to deal with several instances. As a result, the current version of the Rumer programming language supports both references and queries, leaving the programmer the choice to choose whichever construct is more suited for a particular situation.

3.2 Language concepts

In this section, we provide an overview of Rumer's main language concepts.

3.2.1 Entities and relationships

Figure 3.3 shows the skeleton of the Rumer program implementing the running example (see Section 2.1.1 on page 8). The program consists of a sequence of *entity* and *relationship* declarations as well as one *application* declaration. The Rumer program skeleton shown in Figure 3.3 is similar to the first-class relationship-based implementation of the running example shown in Figure 2.15 on page 33. Apart from the application declaration, all declarations of the Rumer program skeleton have a matching counterpart in Figure 2.15: the entity declarations `Student`, `Course`, and `Faculty` correspond to the class declarations `Student`, `Course`, and `Faculty` in Figure 2.15 and the relationship declarations `Attends`, `Assists`, and `Teaches` correspond to the relationship declarations `Attends`, `Assists`, and `Teaches` in Figure 2.15. We deliberately chose the name "entity" for class declarations in Rumer to set Rumer entities apart from classes in existing relationship-based programming languages [1, 2, 3, 4, 5, 6]. In those languages, classes are essentially equivalent to the classes of pure object-oriented programming languages but they can be augmented with relationships if explicit representation of the relationships between classes is desired. Rumer, in contrast, takes advantage of the relationship declarations and offers unique encapsulation guarantees for entities. As we detail in Chapter 4, those encapsulation guarantees result from a stratification of program declarations, which is induced by the relationship declarations.

As opposed to entities and relationships, an application is a mere organizational unit that

```
1  entity Student {...}
2  entity Course {...}
3  entity Faculty {...}
4
5  relationship Attends participants (Student learner, Course course) {
6    int grade;
7  }
8  relationship Assists participants (Student ta, Course course) {...}
9  relationship Teaches participants (Faculty prof, Course course) {...}
10
11 application University {
12   main() {
13     printWelcomeMessage();
14   }
15   void printWelcomeMessage() {
16     writeString("Hello university application!\n");
17   }
18
19 }
```

Figure 3.3: Rumer program skeleton implementing running example (see Section 2.1.1 on page 8).

cannot be instantiated. The differentiation between entities and relationships, on the one hand, and an application, on the other hand, is analogous to Liskov and Zilles' [72] differentiation between operation clusters and procedures to facilitate data abstraction and functional abstraction, respectively. Every Rumer program must declare one application. This requirement is syntactically enforced and manifest in the Rumer grammar, which is shown in Appendix A.1 on page 211. The grammar is specified in EBNF (Extended Backus-Naur Form) form: curly braces { *term* } denote zero, one, or more repetitions of *term* and squared brackets [*term*] denote zero or one repetition of *term*. It conforms to the ANTLR [73] style guidelines and uses lower case initial letters for non-terminals (i.e., parser rules) and upper case initial letters for terminals (i.e., lexer rules). Production program in Appendix A.1 specifies that a Rumer program consists of a sequence of entity and relationship declarations and one application declaration. Production applicationDecl specifies that an application declaration must declare the so called `main()` action. The `main()` action denotes the program entry point. Figure 3.3 declares the application `University`, which declares the `main()` action on line 12. In addition to the `main` action, an application can declare global variables (appVarDecl in Figure 3.3) as well as further value-returning or void routines, so called actions (actionDecl in Figure 3.3). Actions can invoke each other and can be invoked by the `main()` action. In the example, the `main()` action invokes the void action `printWelcomeMessage()`, which is declared on line 15 and writes the text "Hello university application!" to the console. Action `printWelcomeMessage()` invokes the built-in operator `writeString()` (see production statement in Appendix A.1) for that purpose.

Moreover, there are some further syntactic differences between the implementation of the running example shown in Figure 2.15 on page 33 and the Rumer implementation of the running example. The most obvious syntactic difference is the declaration of the participants of a relationship. In Rumer, the participants of a relationship are declared using a

`participants` clause, which denotes the types of the instances that can be related by a relationship instance. The current Rumer language only supports binary relationships; however, all the features of the Rumer language introduced in this thesis generalize to n-ary relationships. According to production participantsDecl in Figure 3.3, programmers can assign identifiers to relationship participants to denote the particular role a participant plays in a relationship. For example, the `participants` clause of relationship `Assists` assigns the identifiers `ta` and `course` to the participant types `Student` and `Course` (line 8 in Figure 3.3), respectively, to denote that a student plays the role of a teaching assistant and a course the role of a course when participating in the `Assists` relationship. Role identifiers can be used in the program text to access the participants of a relationship. As opposed to the implementation of the running example shown in Figure 2.15, the Rumer implementation does not declare the cardinality of a relationship as part of its participants declaration. In Section 3.2.5, we discuss how cardinality constraints are expressed in Rumer. In that section, we introduce Rumer's specification language, which allows the expression of a rich range of consistency constraints.

3.2.2 Elements and extents

Programming languages with support for first-class relationships provide built-in constructs for relationship set up and relationship instance retrieval. Typically, such languages rely on the notion of an *extent*. The term extent originates from the relational database community and the ODMG (Object Data Management Group) object model [74] in particular, where an extent of a type denotes the set of all instances of the type. To facilitate enumeration of objects and relationship instances, the run-time systems of relationship-based programming languages maintain an implicit extent for each class and relationship declaration. In most existing relationship-based programming languages [1, 2, 3, 4, 5, 6] extents are singleton sets. In this case, an extent contains all the existing instances of a type. Nelson et al. were the first to propose explicit extent instantiation to support multiple extent instances for first-class relationships [6]. If multiple extent instances of a type exist, then an extent instance may only contain a subset of all the existing instances of a type and only the union of all extent instances amounts to the set of all the existing instances of a type. Our experience has shown that extent instantiation is vital for implementing recursive data structures with relationships. As a result, Rumer supports explicit extent instantiation for entities and relationships.

We illustrate explicit extent instantiation using the example of the Composite pattern. The Composite pattern *"composes objects into tree structures to represent part-whole hierarchies and lets clients treat leaf objects and composite objects uniformly."* [37]. Figure 3.4 provides the UML class diagram of the Composite pattern. The UML aggregation between class Component and class Composite indicates that the pattern is an instance of a recursive data structure. Figure 3.5 sketches the corresponding implementation of the Composite pattern in Rumer. The Rumer program consists of the entity declaration `Component` and the relationship declarations `Parent` and `Composite`. Entity

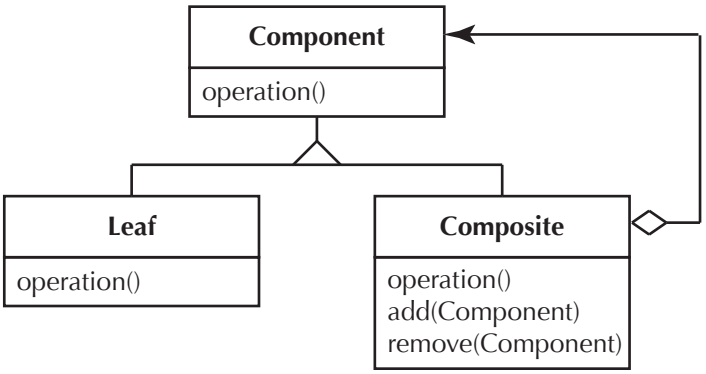

Figure 3.4: UML class diagram modeling the Composite pattern [37].

```
1  entity Component {...}
2
3  relationship Parent participants (Component child, Component parent) {
4    extent void append(Component c, Component p) {
5      these.add(new Parent(c, p));
6    }
7  }
8
9  relationship Composite participants (Component root, Extent<Parent> tree) {
10   extent void createComposite(Component c) {
11     these.add(new Composite(c, new Extent<Parent>()));
12   }
13
14   void appendComponent(Component c, Component p) {
15     this.tree.append(c, p);
16   }
17   // methods to append a (sub)composite to a composite
18 }
19
20 application CompositePattern {
21   main() {
22     Component root1, root2;
23     Extent<Component> components = new Extent<Component>();
24     Extent<Composite> composites = new Extent<Composite>();
25     ...
26     composites.createComposite(root1);
27     composites.createComposite(root2);
28     ...
29   }
30 }
```

Figure 3.5: Rumer program skeleton implementing the Composite pattern.

`Component` represents the basic constituents of a composite. Relationship `Parent` represents the hierarchical structure between the components of a composite, and relationship `Composite` represents the actual composite. Relationship `Composite` represents a "nested" relationship as it declares another relationship (i.e., `Parent`) as its participant.

The `participants` clause of the relationship `Parent` indicates that the `Parent` relationship relates one `Component` entity to another `Component` entity, and assigns the role identifiers `child` and `parent` to those entities, respectively. Figure 3.6 provides a

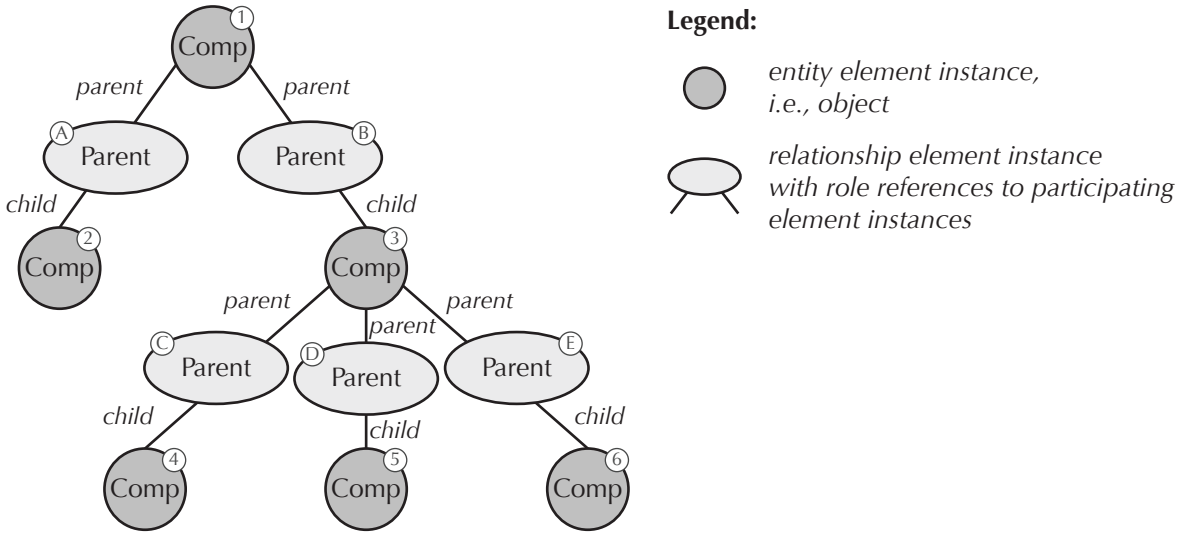

Figure 3.6: Graphical illustration of element instances based on the Composite program sketched in Figure 3.5. The graphic shows the `Component` entity element instances 1, 2, 3, 4, 5, and 6 and the `Parent` relationship element instances A, B, C, D, and E.

graphical illustration of a number of run-time instantiations of the `Parent` relationship and its participating `Component` entity. The figure is based on the graphical notation introduced in Figure 3.2: entity instances are represented as dark gray circles and relationship instances as light gray ellipses, and lines are used to connect relationship instances to their participant instances. Figure 3.6 labels entity instances and relationship instances with numbers and letters, respectively, allowing us to refer to those numbers and letters in the text. To prevent possible confusion between instances of entities and relationships, on the one hand, and instances of entity extents and relationships extents, on the other hand, we use the term *element instance* and *extent instance*, respectively. We chose the term "element instance" for the former to emphasize the fact that those instances are elements of extent instances. Figure 3.6 shows the five `Parent` relationship element instances A, B, C, D, and E, and the six `Component` entity element instances 1, 2, 3, 4, 5, and 6. The graphic depicts the hierarchical structuring achieved by the `Parent` relationship by displaying a `Component` element instance that plays the role of a parent relatively above the `Component` element instances that assume the corresponding child role. The lines connecting a `Parent` element instance to its participating `Component` element instances are labelled with the role identifiers declared by `Parent`'s `participants` clause. In the program text, role identifiers can be used to access the participant instances of a relationship element instance.

The Rumer Composite program in Figure 3.5 makes use of extent instantiation in its `main()` action. It instantiates a `Component` extent on line 23 and binds it to the variable `components`, and it instantiates a `Composite` extent on line 24 and binds it to the variable `composites`. This use of extent instances as "global" element containers is in line with the proposal by Nelson et al. [6]. The `participants` clause of the relationship `Composite`, however, demonstrates that extent instances can also be participants of relationship element instances. A `Composite` element instance uses an extent in-

stance of the `Parent` relationship to represent the hierarchical structure of a composite. It has a `Component` element instance as a root participant and a `Parent` extent instance as a tree participant. We introduced extent instances as participants of relationship element instances in [75]. Such participating extent instances are vital for implementing recursive data structures with relationships. In the Rumer Composite program, `Parent` extents are instantiated and related to their root `Component` element instance in method `createComposite()` on line 11.

Figure 3.7 shows a graphical illustration of the complete heap of the Rumer Composite program. The figure uses the same graphical notation as Figure 3.6 but complements it with extent instances. Extent instances are represented as rectangular boxes. The program heap comprises four extent instances: one `Component` extent instance, two `Parent` extent instances, and one `Composite` extent instance. Both the `Component` and the `Composite` extent instance are created on line 23 and 24, respectively, in Figure 3.5, and the two `Parent` extent instances are created by the two invocations of method `createComposite()` on line 26 and 27 in Figure 3.5. Each extent instance comprises a number of element instances. The `Component` extent instance comprises seven `Component` element instances, the two `Parent` extent instances comprise five `Parent` element instances and one `Parent` element instance, respectively, and the `Composite` extent instance comprises two `Composite` element instances. As Figure 3.6, Figure 3.7 connects relationship element instances to their participant instances by lines. To keep the graphical layout well-arranged, Figure 3.7 introduces an intermediate "shadow" participant instance and connects the relationship element instance to this "shadow" participant instance whenever a direct connecting line to the actual participant instance would result in any crossing lines. A shadow participant instance is a graphical copy of the participant instance that is the actual participant of the relationship element instance. Shadow participant instances are represented by the same symbol as their actual participant instances, but with dotted lines, and they are connected to their actual participant instances by dotted lines. Figure 3.7 uses shadow participant instances for `Component` element instances. For example, the `Parent` element instances residing in the left `Parent` extent instance make use of three shadow `Component` element instances: one for the `Component` element instance assuming the topmost parent component and one for each of its direct children components. Some `Component` element instances and `Parent` element instances in Figure 3.7 are labelled with numbers and letters, respectively. These are exactly the instances that appear in Figure 3.7. The dotted lines between the shadow of the topmost component of a composite's tree and the shadow of a composite's root component indicate that the root component of a composite is at the top of its tree. The dotted lines converging in the `Component` element instance 3 illustrate that the same `Component` element instance can be a constituent of different `Composite` element instances.

The differentiation between element instances and extent instances reveals that entity or relationship declarations in Rumer give rise to two type definitions: an element type of the declared entity or relationship as well as an extent type of the declared entity or rela-

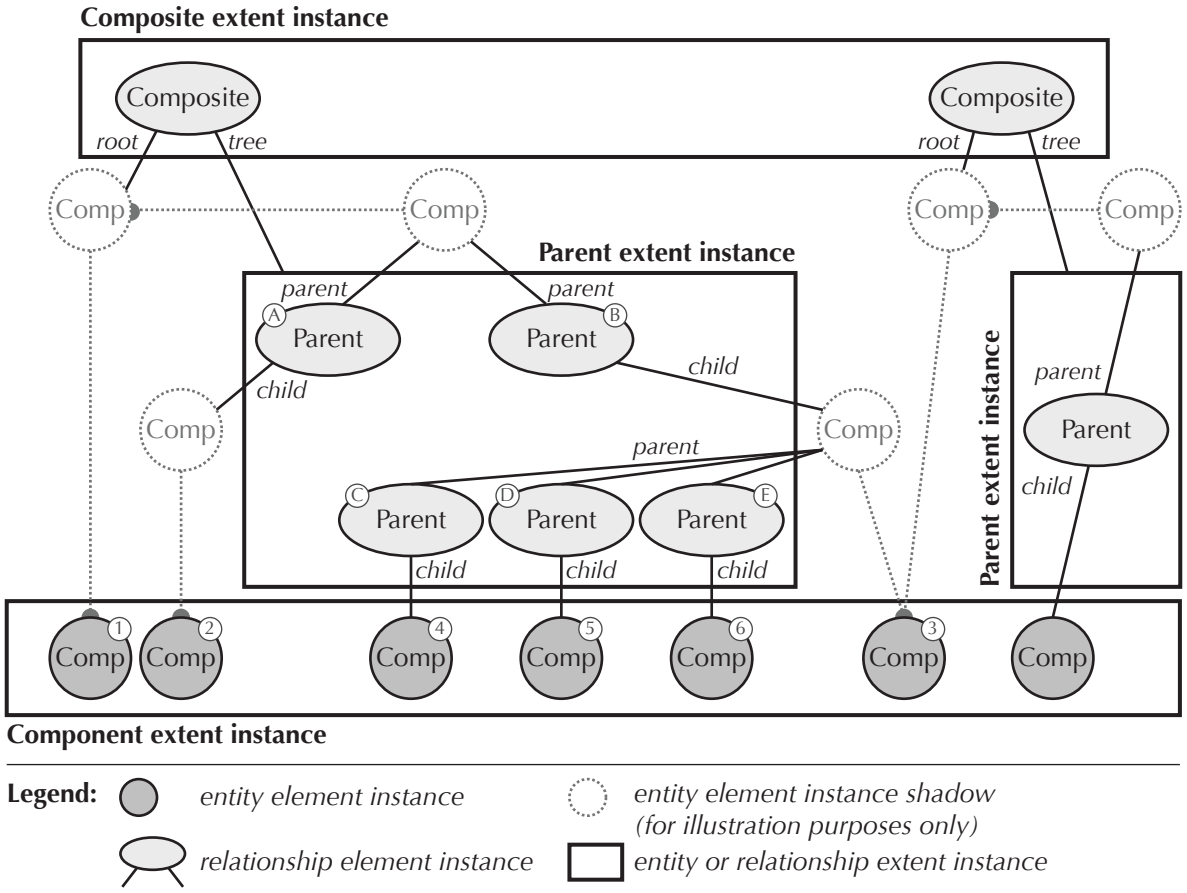

Figure 3.7: Graphical illustration of extent instantiation based on the heap snapshot of the Composite program sketched in Figure 3.5. The graphic shows 1 `Component` extent instance with 7 `Component` element instances, 2 `Parent` extent instances with 5 and 1 `Parent` element instance(s), resp., and 1 `Composite` extent instance with 2 `Composite` element instances. The graphic subsumes Figure 3.6 in its left `Parent` extent instance and uses the same numbers and letters as in Figure 3.6 to label the common `Component` element instances and `Parent` element instances. The `Component` extent instance is referred to by the local variable `components` (line 23 in Figure 3.5) and created on line 23 in Figure 3.5. The `Composite` extent instance is referred to by the local variable `composites` (line 24 in Figure 3.5) and created on line 24 in Figure 3.5. The 2 `Parent` extent instances are created by the two invocations of method `createComposite()` on line 26 and line 27 in Figure 3.5; method `createComposite()` instantiates a `Parent` extent (line 11 in Figure 3.5).

tionship. We detail the Rumer type system in Section 3.3. In this section, we focus on the more practical aspects of element and extent type declaration. According to productions entityDecl and relationshipDecl of the Rumer grammar in Appendix A.1 on page 211, element and extent types of entities or relationships are declared in-line with the entity or relationship declaration. The differentiation between element and extent types occurs at the level of member declarations. Entity and relationship members encompass fields, methods, queries, and invariants (production memberDecl), and according to the productions fieldDecl, queryDecl, methodDecl, and invariantDecl each of those members can either be attributed to an element type or to an extent type. We discuss queries and invariants in Section 3.2.3 and Section 3.2.5, respectively.

Extent members are denoted by the `extent` keyword, which precedes the member declaration. In the Rumer Composite program in Figure 3.5, relationship `Parent` declares the extent method `append()`, and relationship `Composite` declares the extent method `createComposite()` and the element method `appendComponent()`. Both element and extent members have an implicit target associated. In the case of an element member, the implicit target is an element instance of the type declaring the element member. In the case of an extent member, the implicit target is an extent instance of the type declaring the extent member. The current receiver instance of an element member or an extent member can be referred to by the `this` or `these` keywords, respectively. For example, the element method `appendComponent()` refers to the current `Composite` element receiver instance on line 15 in Figure 3.5 to invoke method `append()` on its tree `Parent` extent instance. Similarly, the extent method `append()` of relationship `Parent` invokes the `add()` operator on its current `Parent` extent receiver instance on line 5. The operator `add()` is a built-in operator of the Rumer language (see production primaryExpr in Appendix A.1 on page 211) and allows populating an extent instance. Depopulation of an extent instance is achieved by `add()`'s inverse operator `remove()` (see production statement in Appendix A.1).

To indicate the type of an instance declaration, such as the type of a local variable, programmers use the Rumer source types listed in Appendix A.1 on page 211. According to production elementType, the type of an element instance of an entity or relationship is denoted by the name of the declaring entity or relationship. According to production extentType, the type of an extent instance of an entity or relationship is denoted by the name of the declaring entity or relationship embraced by `Extent<>`. For example, the `main()` action of the Rumer Composite program declares two local variables of the element type `Component` on line 22 and two local variables of the extent types `Extent<Component>` and `Extent<Composite>` on line 23 and 24, respectively. Furthermore, the `participants` clauses of relationship `Parent` and relationship `Composite` indicate that a `Parent` element instance relates an instance of the element type `Component` to an instance of the element type `Component` and that a `Composite` element instance relates an instance of the element type `Component` to an instance of the extent type `Extent<Parent>`, respectively. Valid relationship participant types are element types and extent types of entities and relationships.

3.2.3 Predicate references

The Rumer programming language supports a rich range of *query operators* to retrieve element instances from extent instances. All supported query operators are listed in Appendix A.1 on page 211. Using such a query operator, we can find out for our university example (see Figure 3.3), which students received a grade higher than 5. This example assumes a Swiss grading scale, in which 6 represents the highest grade and 1 the lowest one. Given a variable `Extent<Attends> attends`, the corresponding query is formulated as:

```
attends.select(s_c: s_c.grade > 5).learner;
```

The query uses the `select()` operator and invokes it on the extent instance referred to by `attends`. The `select()` operator takes a *lambda expression* as argument that specifies the selection criterion. In the example, the lambda expression consists of the lambda variable `s_c` and the lambda term `s_c.grade > 5`. The lambda term must evaluate to a boolean expression. The lambda variable is chosen by the programmer and its type is an element type of the type of the expression on which the `select` operator is invoked. In the example, the lambda variable is of the element type `Attends` since the `select()` operator is invoked on an instance of the extent type `Attends`. A `select()` operator evaluates to a subset of the set on which the operator is invoked. In the example, the expression `attends.select(s_c: s_c.grade > 5)` thus evaluates to a subset of all student-course pairs that are contained in the extent instance referred to by `attends`. By applying the role projection operator `learner` on the set returned by the `select()` operator, the result set is reduced to the set of all students who participate in the `Attends` relationship and who attend at least one course for which they received a grade higher than 5.

Query expressions can be bound to variables in Rumer. For the above query expression, this can be achieved using the `let...be` binding (see Appendix A.1 on page 211) in one of the two following ways: by either supplying a name for the query expression

```
let cleverStudents be attends.select(s_c: s_c.grade > 5).learner;
```

or by supplying a name for the query expression as well as the type of its result expression.

```
let query Set<Student> cleverStudents be
  attends.select(s_c: s_c.grade > 5).learner;
```

If no type is supplied, the compiler infers the type from the type of the target expression. As opposed to an assignment, the `let...be` binding has a *non-strict evaluation* semantics. Rather than evaluating the query expression, it binds the query expression (as opposed to its result) to the provided variable. The query expression is only evaluated once the variable to which the query expression is bound is read. Since the program state may change in-between subsequent reads of a query variable, the query may actually evaluate to a different result upon subsequent reads. For example, in the code fragment shown below, the variable `cleverStudents` may evaluate to a different result set upon its first read on line 3 than upon its second read on line 8.

```
1 Student s;
2 ...
3 if (s isElementOf cleverStudents) {
4   writeString("Student " + s.name + " is a clever student.\n");
5 }
6 ... // some modifications of attends
7 let query int avgGradeCleverStudent be
8   Attends.select(s_c: s_c.learner isElementOf cleverStudents).grade.average();
```

The code fragment shown above exemplifies that the `let...be` binding achieves a *call-by-name* evaluation strategy and thus guarantees that a variable bound to a query expression faithfully reflects the state of the program heap upon each read. Due to this property,

we also call query variables *predicate references*. The code fragment shown above further exemplifies that query expressions can be nested and that query variables can be of a primitive type. The variable `avgGradeCleverStudent`, for example, evaluates to an integer value and uses the query variable `cleverStudents` as part of its query expression. The query variable evaluates to the average grade of all the grades of students who attend at least one course for which they received a grade higher than 5. In the code examples shown so far, all the query expressions were bound to local variables. As indicated by production queryDecl in Appendix A.1 on page 211, query expressions can also be declared as element or extent members of entities and relationships. Element or extent queries are bound to an element or extent instance, respectively, and thus come with an implicit target. For completeness, Rumer supports also storing the results of query expressions in variables. To achieve a call-by-value binding, as opposed to a call-by-name binding, programmers must use the assignment operator.

The idea to incorporate database-like queries into a programming languages to query the program heap is not new. Recently, database heap queries found their ways into object-oriented programming languages where queries are granted first-class citizenship [76, 5, 77, 78, 79]. In such languages, queries are not only used to interface with external data stores but also to access the objects that exist in the program heap. Whereas most existing work focuses on heap query support in object-oriented languages, Wren [5] pointed out the importance of heap queries in languages with first-class relationships. He provides an object-based calculus for heap queries and demonstrates that relationships can be expressed in terms of such heap queries. Our work is based on Wren's observation, but adds heap queries on top of a language that provides an explicit abstraction for relationships. Rumer supports a similar range of query operators as those in LINQ (.NET Language-Integrated Query) [78, 79]. Like LINQ, Rumer uses lambda expressions as part of query operators. Rumer heap queries, however, differ in an important aspect from existing heap querying approaches: they are *side-effect free*. This property makes Rumer queries become true predicates over the extents they are applied to. A more subtle distinction also arises from Rumer's meta model. Given its foundation in sets and relations, Rumer's query operators cover the full range of discrete mathematics operators, including set union and relational closure.

3.2.4 Member interposition

Some properties of objects only apply when the object is fulfilling a particular role [61]. In the university example, for instance, we may need to record the language in which a student takes a written exam. The language in which a student takes an exam is an instance of a property of a student that is relevant only to those students who attend courses, but not to all students in general. Rumer provides the mechanism of *member interposition* to express properties of instances that are dependent on the instance's participation in a relationship. We introduced member interposition in [71]. In Chapter 5 and Chapter 7 we demonstrate that member interposition eases the modular verification of invariants over

```
1  entity Student { string name; }
2  entity Course { string title; }
3
4  relationship Attends participants (Student learner, Course course) {
5    int grade; // non-interposed element field
6    string >learner language; // interposed element field
7
8    void >learner print() { // interposed element method
9      writeString(this.name + ": " + this.language);
10   }
11
12   void print() { // non-interposed element method
13     writeString("Student - Course:\n");
14     this.learner.print();
15     writeString(" - " + this.course.title + "\n");
16   }
17 }
```

Figure 3.8: `Attends` relationship of Rumer university program (see Figure 3.3) augmented with an interposed element field declaration as well as with an interposed element method declaration.

shared state. In this section, we focus on the characteristics of member interposition from a programmer's point of view.

Figure 3.8 shows an excerpt of the Rumer university program shown in Figure 3.3. The excerpt shows an extended version of the `Attends` relationship. The new version of the `Attends` relationship uses member interposition to "decorate" a `Student` element instance that participates in the `Attends` relationship as a learner with further properties. According to production interpositionDecl in Appendix A.1 on page 211, an interposed member is declared using the `'>'` sign, which precedes the role identifier of the relationship participant into which the member is interposed. The `Attends` relationship thus declares the interposed relationship element field `language` (line 6) and the interposed relationship element method `print()` (line 8) and interposes both members into the role `learner`. In addition to the interposed element members, relationship `Attends` declares the non-interposed element field `grade` (line 5) and the non-interposed element method `print()` (line 12).

Both non-interposed element members and interposed element members of a relationship have an implicit target. For a non-interposed relationship element member, the implicit target is an element instance of the declaring relationship. For example, the `this` reference in the non-interposed element method `print()` (line 14 and line 15) refers to the current `Attends` element receiver instance. For an interposed relationship element member, on the other hand, the implicit target is an instance of the declaring relationship's participant type that plays the role into which the interposed member is interposed. As we discuss in more detail in Section 3.3, member interposition gives rise to a new type definition: the *role type* of the participant into which the member is interposed. A role type of a relationship participant is a subtype of the participant type. For example, the `this` reference in the interposed element method `print()` (line 9) refers to the current `learner` role element instance of relationship `Attends`, which is a subtype of the

element type `Student`. A role type of a relationship inherits all members from its participant supertype. For example, role `learner` inherits the element field `name` from its supertype `Student`. Inherited members are accessible to a role element instance by the `this` reference (e.g., `this.name` on line 9 of interposed method `print()`). However, interposed members do not override any equally named members of their participant supertype and are thus not dynamically dispatched. For example, if entity `Student` declared an element method `print()`, then the interposed element method `print()` in `Attends` would shadow that method.

Figure 3.9 shows a program heap snapshot of the Rumer university program of Figure 3.8. Figure 3.9 uses the same graphical items as Figure 3.7 to represent element instances and extent instances. As Figure 3.7, Figure 3.9 introduces shadow element instances of entity `Student` to keep the graphical layout well-arranged. As opposed to Figure 3.7, Figure 3.9 displays the values of selected element fields as part of the graphical items rather than the element instance's type. For example, the circles representing `Student` element instances display the values of the `Student` element field `name` and the ones representing `Course` element instances display the values of the `Course` element field `title`. The ellipses representing `Attends` element instances display the values of the non-interposed `Attends` element field `grade`. In addition to the graphical items of Figure 3.7, Figure 3.9 depicts role element instances to display the values of interposed element fields. Role element instances are represented as light gray arcs. They are connected by a line to element instances of the relationship that declares the interposed element members and "huddle" against the element instance of their participant supertype. For example, the leftmost `learner` role element instance with the value "English" for the interposed `Attends` element field `language` is connected to the `Attends` relationship element instances with the values 4 and 6 for the non-interposed `Attends` element field `grade` and huddles against the `Student` entity element instance with the value Paul for the `Student` element field `name`. Figure 3.9 thus indicates that the student Paul attends the courses Program and Compiler, for which he receives the grades 6 and 4, respectively, and that he takes any written exams in English.

The program heap shown in Figure 3.9 comprises two `Attends` extent instances, one `Student` extent instance, and two `Course` extent instances. This setup reflects the fact that the courses are offered by two separate schools, a technical university and an art school, and that students can attend courses at both schools. The right `Course` extent instance denotes the art school's extent instance and the left `Course` extent instance denotes the technical university's extent instance. In Figure 3.9, Susan is the only student who attends courses at both schools. She attends the courses Program and Art, for which she receives the grades 5 and 6, respectively. The heap snapshot shown in Figure 3.9 further indicates that there exists a role element instance for each relationship extent instance in which an element instance participates. As a result, interposed members can be customized for relationship extent instances. In the heap snapshot shown in Figure 3.9, for example, Susan takes written exams in English at the technical university and in French at the art school.

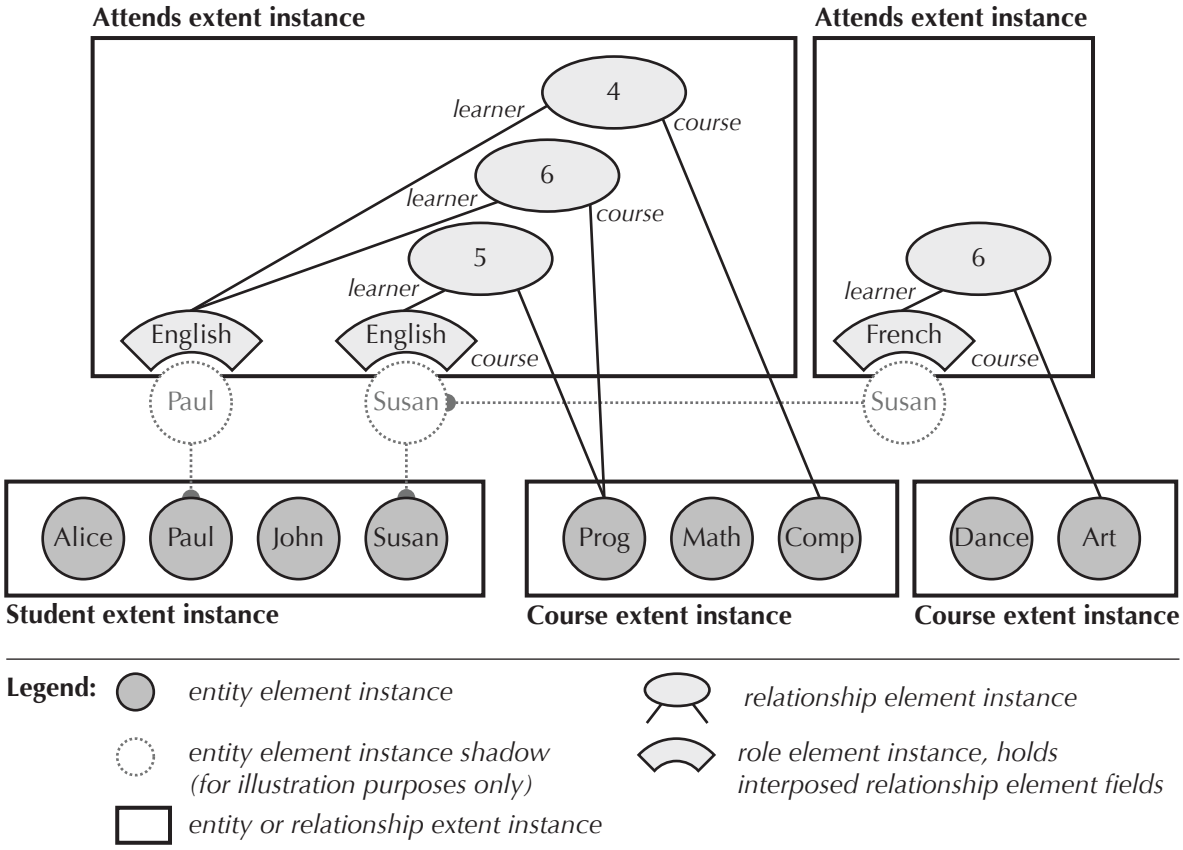

Figure 3.9: Graphical illustration of member interposition based on the heap snapshot of the university program shown in Figure 3.8. The graphic demonstrates that there exists a role element instance for each extent instance of a particular relationship type in which an element instance participates.

Interposed members are encapsulated in their declaring relationships. As a result, different relationships can interpose members into the same participant type. Figure 3.10 shows a partial view of the university heap snapshot of Figure 3.9 from the perspective of the student Paul. In addition to instances of the `Attends` relationship, Figure 3.10 depicts also instances of the `Assists` relationship. For illustration purposes, we assume that relationship `Assists` also declares an interposed relationship element field `language` for the role `ta`. This field denotes the language a teaching assistant uses to hold exercise sessions. Even though relationships `Attends` and `Assists` interpose the field `language` into the same participant type, these declarations are not conflicting. At runtime, there exists a role element instance for each relationship type, and interposed fields are part of the role element instance and not part of the participant supertype instance. As a result, Paul can use a different language for holding exercise sessions (i.e., German) than for taking written exams (i.e., English).

The notion of a role has also received attention in other works. Whereas some works employ roles as purely conceptual notions for system [61] or framework [63] design, other works take the notion of a role further and provide explicit support for roles by a library [68] or by a programming language [67, 65, 66]. In all of these works, roles are granted first-class citizenship. In Rumer, in contrast, roles are implicit and tied to the re-

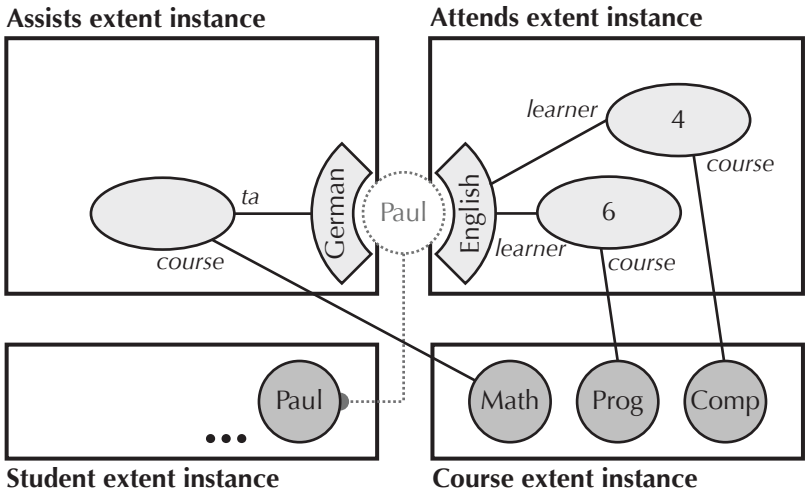

Figure 3.10: Partial view of the university program heap shown in Figure 3.9 extended with relationship `Assists`. The graphic focuses on student Paul and indicates that different relationship types (i.e., `Assists` and `Attends`) can interpose equally named element fields (i.e., `language`) into the same participant type. See legend in Figure 3.9.

lationship that declares the interposed members. The current treatment of roles in Rumer already provides considerable flexibility. However, to provide even more flexibility and to increase the scalability of the language, the support of first-class roles in Rumer may be necessary. Appropriate extensions have been worked out by Sebestyén-Pál [80] and Conconi [81]. Relationship libraries [47, 48, 53] that are based on aspect-oriented programming [49, 50] can further achieve an effect similar to member interposition. By leveraging inter-type declarations of aspect-oriented programming, relationship-dependent properties of objects can be expressed.

3.2.5 Specification language

In this section, we introduce the Rumer specification language. We focus on the main features of the Rumer specification language in this section and discuss aspects related to the verification of specifications in Chapter 4, Chapter 5, and Chapter 6.

The Rumer specification language supports Design-by-Contract-style [21] assertions as present in object-oriented contract languages, such as Eiffel [82], JML (Java Modeling Language) [83, 84], and Spec♯ [85]. A distinguishing feature of the Rumer specification language is its seamless integration with the Rumer programming language. Leveraging the query operators available in the Rumer programming language, set comprehensions and quantified expressions can be easily expressed as part of specifications. Furthermore, Rumer query expressions can be used as part of assertions since query expressions are side-effect free. As indicated by the Rumer grammar shown in Appendix A.1 on page 211, the Rumer specification language includes routine *preconditions* and routine *postconditions* (production routineSignatureTail), *assert* statements (production statement), and *invariants* (production invariantDecl). Invariants can be declared both for element types and extent types as well as for a Rumer program's application, giving rise

to the following five invariant categories:

- **Entity element invariant:** Property that must hold for each element 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 element invariant:** Property that must hold for each element 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.
- **Application invariant:** Property that must hold for the singleton instance of the application that declares the invariant.

These invariant categories refine the categories introduced in earlier work [71]. Element and extent invariants correspond to value-based and structural invariants, respectively, in [71]. In addition, we support extent invariants for entities, for which only the declaration of value-based invariants is possible in [71].

Figure 3.11 shows an extended version of the Rumer Composite program in Figure 3.5. The new version introduces invariants, method preconditions and postconditions, and the interposed element field `total` (line 2). The interposed element field `total` of the `Parent` relationship indicates per `parent` role element instance the number of `child` role element instances it is directly or indirectly related to. This number must be equal to the number of children components contained within the sub-tree rooted at a parent component. The new version of the Composite program declares a relationship extent invariant for relationship `Parent` (line 21) and a relationship element invariant for relationship `Composite` (line 30). As indicated by production invariantDecl in Appendix A.1 on page 211, extent invariants are distinguished from element invariants by the keyword `extent`.

The extent invariant of relationship `Parent` leverages Rumer query expressions (see Section 3.2.3) and guarantees the following properties: *(i)* that every child component is related to at most one parent component (line 22), *(ii)* that the relation described by the current `Parent` extent instance is acyclic (line 23), 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 by the `Parent` relationship (line 24). Conjuncts *(i)* and *(ii)* nicely demonstrate how specifications can benefit from the abstractions available in the relational model underlying the Rumer language. Conjunct *(i)* refers to the invariant's `Parent` extent instance and guarantees that the relation described by the extent instance forms a partial function. As indicated by production postfixExprTail in Appendix A.1 on page 211, the operator `isPartialFunction()` is a built-in query operator that evaluates to a boolean. Conjunct *(ii)* guarantees that the transitive closure of the relation described by the invariant's `Parent` extent instance is irreflexive. The operators `transitiveClosure()` and `isIrreflexive()` are both built-in query operators (production postfixExprTail in Appendix A.1) and they return a set of instance

```
1  relationship Parent participants (Component child, Component parent) {
2    int >parent total; // interposed element field
3
4    extent void append(Component c, Component p)
5      requires // precondition
6        c != null & p != null & c != p &
7        !(c isElementOf these.child union these.parent) &
8        (!these.isEmpty() => p isElementOf these.child union these.parent);
9      ensures // postcondition
10       these == old(these) union Set(Pair(c, p)) &
11       forAll(x isElementOf
12         these.transitiveClosure().select(cp: cp.child == c).parent:
13           x.total == old(x.total) + 1);
14   {
15     these.add(new Parent(c, p));
16     foreach (x isElementOf
17       these.transitiveClosure().select(cp: cp.child == c).parent)
18     { x.total = x.total + 1; }
19   }
20
21   extent invariant // extent invariant
22     these.isPartialFunction() &
23     these.transitiveClosure().isIrreflexive() &
24     forAll(p isElementOf these.parent: p.total ==
25        these.transitiveClosure().select(cp: cp.parent == p).count());
26 }
27
28 relationship Composite participants (Component root, Extent<Parent> tree) {
29   ...
30   invariant // element invariant
31     !(this.root isElementOf this.tree.child) &
32     (!this.tree.isEmpty() => this.root isElementOf this.tree.parent) &
33     this.tree.transitiveClosure().select(cp: cp.parent == this.root).child ==
34        this.tree.child;
35 }
```

Figure 3.11: Rumer Composite program extending the version shown in Figure 3.5 with the interposed element field `total` as well as with invariants and method pre- and postconditions. See complete Composite pattern specification in Section 7.1.2 on page 193.

tuples and a boolean, respectively. Conjuncts *(i)* and *(ii)* guarantee that the relation described by a `Parent` extent instance forms a forest of trees. Conjunct *(iii)* uses the built-in universal quantifier `forAll` (production quantifiedPredicate in Appendix A.1). Rumer only supports bounded quantifiers. The `forAll` operator uses a lambda expression to restrict the range of the quantified lambda variable (left of the colon) as well as a lambda expression to specify the predicate of the quantifier (right of the colon). In the example, the lambda variable ranges over all `parent` role element instances contained in the invariant's `Parent` extent instance. The predicate of the universal quantifier guarantees, for each `parent` role element instance, that its `total` field is equal to the number of `Parent` element instances that transitively relate a `child` role element instance to the `parent` role element instance. The built-in operator `count()` yields the number of element instances contained in the set on which the operator is invoked.

The extent invariant of relationship `Parent` does not yet guarantee that a composite's

tree `Parent` extent instance indeed forms a tree, but also permits a forest. To restrict a `Composite` element instance's tree `Parent` extent instance to a real tree, relationship `Composite` declares the following element invariant: *(i)* that a composite's root component never appears as a child component in the relation described by the composite's tree `Parent` extent instance (line 31), *(ii)* that a composite's root component appears as a parent component in the relation described by the composite's tree `Parent` extent instance, unless this relation is empty (line 32), and *(iii)* that a composite's root component is the transitive parent component of all children components of the relation described by the composite's tree `Parent` extent instance (line 33). Conjunct *(iii)* uses again the `transitiveClosure()` operator to compute the transitive closure of the relation described by the composite's tree `Parent` extent instance. From the resulting relation, conjunct *(iii)* chooses those `Parent` relationship instances that have the composite's root component as a parent component and projects onto their children components. Those children components have to be exactly the children components of a composite's tree `Parent` extent instance. The element invariant of relationship `Composite` thus guarantees 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. As indicated by production postfixExprTail in Appendix A.1 on page 211, the operator `isEmpty()` is a built-in query operator that evaluates to a boolean, indicating whether the set on which the operator is invoked is empty or not.

The heap snapshot shown in Figure 3.7 represents a valid instantiation of the Composite pattern specification in Figure 3.11[2]. The displayed `Composite` element instances all have tree `Parent` extent instances that form proper trees. As illustrated by Figure 3.7, it is possible for a `Component` element instance to be part of different `Parent` extent instances. For example, the two shadow element instances connected to the sixth `Component` element instance in Figure 3.7 indicate that the sixth `Component` element instance is part of both `Parent` extent instances shown in Figure 3.7. The sharing of `Component` element instances among different `Parent` extent instances does not compromise `Parent`'s extent invariant. Extent invariants must only hold for each extent instance but not for the union of all extent instances.

To guarantee the Composite pattern invariants listed in Figure 3.11, the methods of the Rumer Composite pattern program define preconditions and postconditions. We briefly illustrate the most important features of method preconditions and postconditions based on the preconditions and postconditions of method `append()` (see line 5 and line 9, resp. in Figure 3.11). By appending the argument component `c` as child of the argument component `p`, method `append()` may compromise the extent invariant of `Parent`. To prevent introducing cycles and relating a child component to several parent components, the method requires the following precondition:

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

As indicated by production expression in Appendix A.1 on page 211, the operator `union`

[2]Note that Figure 3.7 does not display `Parent`'s interposed element field `total`.

is a built-in query operator that yields the union of its left and right operand sets. Method `append()` further increments 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(cp: cp.child == c).parent:
  x.total == old(x.total) + 1)
```

The `old()` operator is a language built-in operator (production primaryExpr in Appendix A.1 on page 211), which can only occur in routine postconditions. It captures the value of the argument expression in the pre-state of the routine invocation in whose postcondition the operator occurs.

Rumer is the only relationship-based programming language that supports Design-by-Contract-style [21] assertions. However, there exist a number of object-oriented contract languages that support Design-by-Contract-style [21] assertions, such as Eiffel [82], JML (Java Modeling Language) [83, 84], and Spec♯ [85]. The individual contract languages vary in range of supported assertions. JML is the object-oriented contract language that supports the richest range of assertions. For example, it supports not only instance invariants (a.k.a. object invariants) but also static invariants. Also, JML features list comprehensions and quantified expressions. An extension of the Spec♯ methodology with static (including quantified) invariants is discussed in [86], but not yet integrated into the Spec♯ programming system. Extent invariants are related to static invariants in that they are decoupled from an individual instance and support quantified expressions. In contrast to JML, the Rumer specification language relies on a stratified programming model (see Chapter 4), which allows certain invariants to be broken in initial and final states of method executions, and also imposes different admissibility criteria on invariant declarations. Furthermore, Rumer supports a richer range of assertions than JML. Leveraging the abstractions of the relational model underlying the Rumer language, the Rumer specification language provides support not only for set comprehensions and quantified expressions but also for the full range of discrete mathematics operators (e.g., set union, Cartesian product, composition, transitive closure, etc.). The notion of an extent has also proved viable for quantified expressions. In Rumer, quantified variables can only range over instances contained in an extent instance. This treatment avoids the problem of quantifying over possibly non-allocated objects [87].

3.3 Type system

In this section, we introduce the types of the Rumer language and the subtyping relations between them.

3.3.1 Types

Figure 3.12 lists the types of the Rumer language. These types correspond to the source types listed in Appendix A.1 on page 211 that can occur as part of type declarations in the

SingletonType ≙ ApplicationName
NominalType ≙ EntityName | RelationshipName | RelationshipName.RoleName

ReferenceType ≙ ElementType | ExtentType
ElementType ≙ Element⟨NominalType⟩ | Any
ExtentType ≙ Extent⟨Element⟨EntityName⟩⟩ | Extent⟨Element⟨RelationshipName⟩⟩

ValueType ≙ SetType | PairType | BagType | ItemType
SetType ≙ Set⟨CoordinateType⟩
PairType ≙ Pair⟨CoordinateType, CoordinateType⟩
CoordinateType ≙ ReferenceType | SetType | PairType
BagType ≙ Bag⟨ItemType⟩ | Bag⟨BagType⟩
ItemType ≙ MoldType | PrimitiveType
MoldType ≙ Mold⟨NominalType⟩
PrimitiveType ≙ boolean | string | int | float

Figure 3.12: Rumer programming language types. These types constitute the complete set of Rumer types. Some of the types listed above are only used internally for type checking and cannot appear in the source text. See Appendix A.1 on page 211 for the types that can appear in the source text.

Rumer program text. A comparison of the source types listed in Appendix A.1 and the types listed in Figure 3.12 reveals that, in some cases, the naming of a type is different for source types and internal types. In the case of element type declarations, the identifier Element is dropped for source types. For example, the type of the local variables `root1` and `root2` on line 22 in Figure 3.5 is Element⟨Component⟩, which is represented in the Rumer source text as `Component`. Furthermore, such a comparison reveals that the source types are a subset of the internal types. Figure 3.12 also includes types that are only used for type checking purposes internally (e.g., Mold⟨RelationshipName.RoleName⟩) but that cannot appear in the source text.

The Rumer types are divided into *reference types* (ReferenceType in Figure 3.12), *value types* (ValueType in Figure 3.12), and *singleton types* (SingletonType in Figure 3.12). Whereas multiple instances of reference types and value types may exist, there only exists a single instance of a singleton type. The application declaration of a Rumer program gives rise to a singleton type, of which an instance is generated by the run-time system. Both reference types and value type are based on nominal types (NominalType in Figure 3.12) that subsume all programmer-definable entity and relationship declarations. As opposed to value types, reference types have a unique, system-generated identity. As a result, reference types are compared by identity ("by reference") and value types are compared structurally. Since the particular kind of comparison is type-dependent, Rumer only supports a single comparison operator, which is the `'=='` operator.

Reference types are either *element types* (ElementType in Figure 3.12) or *extent types* (ExtentType in Figure 3.12). As pointed out in Section 3.2.2, element and extent types arise from entity or relationship declarations. Whereas an entity or relationship element type subsumes the element members of an entity or relationship declaration, an entity or relationship extent type subsumes the extent members of an entity or relation-

ship declaration. For example, entity `Student` in Figure 3.8 gives rise to the following two types: Element⟨Student⟩ and Extent⟨Student⟩. Type Element⟨Student⟩ has the element field `name`. Due to member interposition (see Section 3.2.4), relationship declarations result in two kinds of element types: one denoting the relationship element type Element⟨RelationshipName⟩ and one denoting the role element type of a relationship participant Element⟨RelationshipName.RoleName⟩. For example, relationship `Attends` in Figure 3.8 brings about the element types Element⟨Attends⟩, Element⟨Attends.learner⟩, and Element⟨Attends.course⟩. Furthermore, it gives rise to the extent type Extent⟨Attends⟩. Type Element⟨Attends⟩ has the element field `grade` and the element method `print()`. Type Element⟨Attends.learner⟩ has the element field `language` and the element method `print()`. In addition to the programmer-defined element types, the Rumer language includes the element type constant Any, which serves as the supertype of every element type (see also Section 3.3.2).

Value types are either set types (SetType in Figure 3.12), pair types (PairType in Figure 3.12), bag types (BagType in Figure 3.12), or item types (ItemType in Figure 3.12).

The *set type* is used in the typing of query expressions that evaluate to a subset of the extent type or set type instance on which the query operator is invoked. For example, assuming the variable `Extent<Student> ethStudents` and the Rumer university program shown in Figure 3.8, the query expression

```
ethStudents.select(s: s.name == ``Heidi'')
```

yields all students with the name "Heidi" comprised in the target extent instance and is of type Set⟨Element⟨Student⟩⟩. As opposed to extents, which are mutable reference types and can be explicitly instantiated and populated by programmers, sets are immutable value types and can only be generated by querying extent or set instances.

The *pair type* closes the type system under set and relational operators. For example, assuming the variables `Extent<Attends> ethAttends` and `Extent<Assists> ethAssists` and the Rumer university program shown in Figure 3.3, the query expression

```
ethAttends union ethAssists
```

yields the union of the students-course pairs comprised in the two operand extent instances and is of type Set⟨Pair⟨Element⟨Student⟩, Element⟨Course⟩⟩⟩.

Both set types and pair types have *coordinate types* (CoordinateType in Figure 3.12) as their elements and coordinates, respectively. The name CoordinateType shall convey the idea that such a type can only be a "coordinate" of a set or pair. Coordinate types subsume element types, extent types, set types, and pair types. Extent types may appear as coordinate types of a set or pair since extent instances can be participants of relationship element instances. Furthermore, the inclusion of the set type and pair type allows for recursive set and pair definitions, respectively.

The *bag type* is analogous to the bag types present in relational database management systems [35] that result from projections [70] onto relations. For example, assuming

the variable `Extent<Student> ethStudents` and the Rumer university program shown in Figure 3.8, the query expression

```
ethStudents.name
```

yields a bag of strings indicating the name of every student contained in the target extent instance, and is of type Bag⟨string⟩. As opposed to extents and sets, whose elements are distinct, the elements contained in a bag may occur a (finite) number of times. Rumer bags are thus equivalent to the bags (or multisets) of set theory. To emphasize the distinction between extents and sets, on the one hand, and bags, on the other hand, we call the elements of sets "elements" and the elements of bags "items". The distinctness of extent and set elements is due to the identity of their elements (or of the elements they are based on). Bag items, on the other hand, are confined to value types and thus they do not have identity. Both extent and set instances as well as bag instances can be empty.

The *item type* subsumes mold types (MoldType in Figure 3.12) and primitive types (PrimitiveType in Figure 3.12), whose instances can both appear as bag items. The Rumer *primitive types* are analogous to the Java primitive types, with the exception that Rumer treats strings as immutable value types and that it does not support the declaration of memory size requirements for numerical types. A *mold type*, on the other hand, is conceptually similar to a delayed type [88] because it represents an incompletely initialized "element instance". A mold type exists for each programmer-defined element type and has a field for each of the element type's element fields. Unlike an element type, a mold type is a value type and does not have identity. For example, the argument expression `new Parent(c, p)` on line 15 in Figure 3.11 is of type Mold⟨Parent⟩ and represents a newly instantiated `Parent` mold instance that has the `Component` element instances referred to by `c` and `p` as participants. We detail our motivation to introduce mold types and further benefits in Section 3.4.1.

We find it further convenient to introduce the notion of a *base type*, allowing us to abstract from the fact whether an instance is an element or an extent instance:

Definition 3.1 (Base type): For any application type App, any relationship type R with participant nominal types A a and B b, and any entity type E, a base type of an instance o is defined as:

$$BaseType(o) = \begin{cases} \mathsf{App} & \text{if } o \text{ is of type App.} \\ \mathsf{R} & \text{if } o \text{ is of type Extent⟨Element⟨R⟩⟩, Element⟨R⟩, Element⟨R.a⟩,} \\ & \text{Element⟨R.b⟩, or Mold⟨R⟩.} \\ \mathsf{E} & \text{if } o \text{ is of type Extent⟨Element⟨E⟩⟩, Element⟨E⟩, or Mold⟨E⟩.} \end{cases}$$

The notion of a base type is particularly helpful if we want to refer to a relationship element instance's participating instances without knowing whether those instances are element instances or extent instances. As we discuss in Chapter 4, the base type of an instance also indicates the stratification layer of an instance.

3.3.2 Subtyping

Figure 3.13 details the subtyping relations between the Rumer language types discussed in the previous section. The first four subtyping rules SUB-ROLE1, SUB-ROLE2, SUB-ROLE3, and SUB-ROLE4 capture the subtyping relations induced by member interposition. The rules indicate that both role types of a relationship are subtypes of their corresponding participant types. As already pointed out in Section 3.2.4, interposed members do not override any equally named members of their participant supertype and are not dynamically dispatched. The subtyping rule SUB-ENTITY-ANY guarantees that every programmer-defined entity element type conforms to Any. Any is a type constant that is the maximal element of the subtyping relation defined on element types. The subtyping rules SUB-RELATIONSHIP-PAIR and SUB-PAIR-ANY guarantee that also programmer-defined relationship element types conform to Any. The subtyping rule SUB-RELATIONSHIP-PAIR guarantees that every relationship element instance conforms to a pair type instance with the corresponding role element instances of the relationship element instance as participants. The subtyping rule SUB-PAIR-DEPTH establishes depth subtyping for pairs. This rule makes the pair type constructor covariant with its coordinate types. Since the pair type is an immutable value type, covariance for pair types can be soundly supported. The subtyping rule SUB-EXTENT-SET guarantees that an instance of an extent type conforms to a corresponding set type instance that has the same element type as the extent type instance. Furthermore, this subtyping rule ensures that all query operators applicable to set instances are also applicable to extent instances. The subtyping rule SUB-COVSET makes the set type constructor covariant with its element type. Again, this subtyping rule does not compromise type soundness since the set type is an immutable value type. The differentiation between extent types and set types is similar to the type constructors Sink and Source described in [89, 90], which separate the writing to and reading of a type instance, respectively. Unlike the Sink type constructor, however, the extent type constructor is invariant. The subtyping rule SUB-COVBAG makes also the bag type constructor covariant with its item type. Again, this subtyping rule does not compromise type soundness since the bag type is an immutable value type. The subtyping rule SUB-INT-FLOAT allows for widening conversions between numeric primitive type instances. The last two subtyping rules SUB-REFLX and SUB-TRANS, finally, close the subtyping rules under transitivity and reflexivity, establishing a partial order on the subtyping relation.

We illustrate the subtyping rules in Figure 3.13 on the typing of the query expression

```
ethAttends union ethAssists
```

that was introduced in the previous section. To determine the type of the query expression, we must determine the least common supertype of the operand expressions `ethAttends` and `ethAssists`. The types of the operand expressions `ethAttends` and `ethAssists` are Extent⟨Element⟨Attends⟩⟩ and Extent⟨Element⟨Assists⟩⟩, respectively. To determine the least common supertype of these operand types, we simultaneously "climb up" the subtyping relation for both operand types. Applying the rule

$$\frac{\texttt{relationship R participants (A a, \_\_)}}{\text{Element}\langle \text{R.a}\rangle \ <:\ \text{Element}\langle \text{A}\rangle} \quad \text{(SUB-ROLE1)}$$

$$\frac{\texttt{relationship R participants (Extent<A> a, \_\_)}}{\text{Element}\langle \text{R.a}\rangle \ <:\ \text{Extent}\langle\text{Element}\langle \text{A}\rangle\rangle} \quad \text{(SUB-ROLE2)}$$

$$\frac{\texttt{relationship R participants (\_\_, B b)}}{\text{Element}\langle \text{R.b}\rangle \ <:\ \text{Element}\langle \text{B}\rangle} \quad \text{(SUB-ROLE3)}$$

$$\frac{\texttt{relationship R participants (\_\_, Extent<B> b)}}{\text{Element}\langle \text{R.b}\rangle \ <:\ \text{Extent}\langle\text{Element}\langle \text{B}\rangle\rangle} \quad \text{(SUB-ROLE4)}$$

$$\text{Element}\langle \text{EntityName}\rangle \ <:\ \text{Any} \quad \text{(SUB-ENTITY-ANY)}$$

$$\frac{\texttt{relationship R participants (\_a, \_b)}}{\text{Element}\langle \text{R}\rangle \ <:\ \text{Pair}\langle\text{Element}\langle \text{R.a}\rangle, \text{Element}\langle \text{R.b}\rangle\rangle} \quad \text{(SUB-RELATIONSHIP-PAIR)}$$

$$\frac{\text{S} \ <:\ \text{T} \quad \text{U} \ <:\ \text{V}}{\text{Pair}\langle \text{S}, \text{U}\rangle \ <:\ \text{Pair}\langle \text{T}, \text{V}\rangle} \quad \text{(SUB-PAIR-DEPTH)}$$

$$\text{Pair}\langle \text{S}, \text{U}\rangle \ <:\ \text{Any} \quad \text{(SUB-PAIR-ANY)}$$

$$\text{Extent}\langle\text{Element}\langle \text{A}\rangle\rangle \ <:\ \text{Set}\langle\text{Element}\langle \text{A}\rangle\rangle \quad \text{(SUB-EXTENT-SET)}$$

$$\frac{\text{S} \ <:\ \text{T}}{\text{Set}\langle \text{S}\rangle \ <:\ \text{Set}\langle \text{T}\rangle} \quad \text{(SUB-COVSET)}$$

$$\frac{\text{S} \ <:\ \text{T}}{\text{Bag}\langle \text{S}\rangle \ <:\ \text{Bag}\langle \text{T}\rangle} \quad \text{(SUB-COVBAG)}$$

$$\text{int} \ <:\ \text{float} \quad \text{(SUB-INT-FLOAT)}$$

$$\text{T} \ <:\ \text{T} \quad \text{(SUB-REFLX)}$$

$$\frac{\text{S} \ <:\ \text{T} \quad \text{T} \ <:\ \text{U}}{\text{S} \ <:\ \text{U}} \quad \text{(SUB-TRANS)}$$

Figure 3.13: Subtyping relation between Rumer language types. The meta-variables A and B range over entity or relationship names and the meta-variables S, T, U, and V range over type names. See Figure 3.12 for the Rumer language types.

SUB-EXTENT-SET to both operand types, we get the types Set⟨Element⟨Attends⟩⟩ and Set⟨Element⟨Assists⟩⟩ for the left and right operands types, respectively. Then, by applying the rules SUB-RELATIONSHIP-PAIR and SUB-COVSET to both operand types, we get the type Set⟨Pair⟨Element⟨Attends.learner⟩, Element⟨Attends.course⟩⟩⟩ and the type Set⟨Pair⟨Element⟨Assists.ta⟩, Element⟨Attends.course⟩⟩⟩ for the left and right operand types, respectively. Finally, by applying the rules SUB-ROLE1 and SUB-ROLE3 to both operand types, we get the type Set⟨Pair⟨Element⟨Student⟩, Element⟨Course⟩⟩⟩ and the type Set⟨Pair⟨Element⟨Student⟩, Element⟨Course⟩⟩⟩ for the left and right operand types, respectively. By rule SUB-COVSET the least common supertype of the two operand types thus becomes Set⟨Pair⟨Element⟨Student⟩, Element⟨Course⟩⟩⟩.

3.4 Semantics

In this section, we detail the semantics of the Rumer built-in operators. The implementation of those built-in operators guarantees, in particular, that entity and relationship element instances reside in an extent instance and that those extent instances are disjoint. The disjointness of extent instances leads to a partitioning of the Rumer program heap, which is leveraged by the verification technique introduced in Chapter 5. Furthermore, the implementation of the Rumer query operators guarantees that query expressions in Rumer are side-effect free, thus making them predicates over the program heap.

3.4.1 Instance instantiation and initialization

Rumer provides a number of built-in operators for the instantiation and initialization of mold types, element types, and extent types.

Elements and molds

Element type instances in Rumer can only be created from corresponding mold type instances. This restriction enforces a strict *instantiation and initialization pattern*: to create an element instance of type Element⟨A⟩, a programmer first creates an instance of the corresponding mold type Mold⟨A⟩, (successively) initializes the mold instance, and then creates the element instance by providing the mold instance as a template. As briefly mentioned in Section 3.3.1, mold types are inspired by delayed types [88] and represent partially initialized "element instances". A mold type exists for each programmer-defined element type and has a field for each of the element type's element fields. Representing the two distinct phases of the life cycle of an element instance — its creation and initialization phase as opposed to its operational phase — at the level of the type system has a number of advantages. Most importantly, it facilitates a separate treatment of partially initialized "element instances" and (supposedly) fully initialized element instances at compile-time. For example, as we detail in Chapter 5, only element instances are expected to meet the element invariants of their nominal type, but not mold instances.

Mold instances are created by using the Rumer built-in *mold constructors* (see Table 3.14 and Table 3.16). Mold constructors exist for the creation of entity mold instances and relationship mold instances. Although the Rumer type system includes also a role mold type (Mold⟨RelationshipName.RoleName⟩ in Figure 3.12), role mold types are not explicitly instantiated by the programmer.

As detailed by Table 3.14, a mold instance of an entity type E is created by calling the built-in *entity mold constructor* `new E()`. The entity mold constructor creates an instance of type Mold⟨E⟩, initializes all fields of the newly created mold instance with default values, and returns the mold instance. If default initialization is not appropriate, a programmer can customize mold initialization by declaring a *mold initializer*.

Operator for an entity E		
`new E()`	**Return:**	Mold⟨E⟩ *e*
	Description:	Creates an instance *e* of type Mold⟨E⟩, initializes *e*'s fields with default values, and returns *e*.
	Postcondition:	*e* has all fields initialized with default values.

Table 3.14: Built-in entity mold constructor.

```
1  entity Student {
2    string name;
3    int id;
4    string language;
5
6    init initialize(string name, int id) {  // mold initializer
7      this.name = name;
8      this.id = id;
9    }
10
11   init setLanguage(string language) {  // mold initializer
12     this.language = language;
13   }
14
15   extent void createStudent(string name, int id, string language) {
16     Student.add(new Student().initialize(name, id).setLanguage(language));
17   }
18 }
```

Figure 3.15: `Student` entity of Rumer university program (see Figure 3.3) augmented with further element fields and mold initializer declarations.

A mold initializer is a programmer-defined routine for the initialization of mold fields. Figure 3.15 displays a revised declaration of the `Student` entity of the Rumer university program shown in Figure 3.3. Entity `Student` declares the two mold initializers `initialize()` (line 6) and `setLanguage()` (line 11). As indicated by production moldInitializerDecl in Appendix A.1 on page 211, mold initializers are denoted by the `init` keyword, which precedes the name of the mold initializer. Mold initializers can only be invoked on mold instances and they return a mold instance. Since molds are value types, a mold initializer creates a copy of the current mold receiver instance, on which it executes the body of the mold initializer and which it finally returns. The mold initializers `initialize()` and `setLanguage()` are invoked on line 16 in Figure 3.15. The invocations demonstrate that several initializers can be invoked in succession. The separation between mold constructors and mold initializers is more a matter of convenience than necessity. By separating mold instantiation from mold initialization — as opposed to combining the two by constructor overloading — incremental initialization becomes possible. For example, the fields of several mold instances can be equally initialized by using the same mold instance as the initial target of the successive initializer invocations.

The instantiation and initialization of relationship mold instances is similar to the one of entity mold instances. Table 3.16 lists the built-in Rumer *relationship mold constructor*. Whereas entity mold constructors do not take any arguments, relationship mold construc-

Operator for a relationship R with participant base types A a and B b		
`new R()`	**1. argument:**	Element⟨A⟩ *a* **or** Extent⟨Element⟨A⟩⟩ *a*
	2. argument:	Element⟨B⟩ *b* **or** Extent⟨Element⟨B⟩⟩ *b*
	Return:	Mold⟨R⟩ *r*
	Description:	Creates the instances *r*, *ra*, and *rb* of type Mold⟨R⟩, Mold⟨R.a⟩, and Mold⟨R.b⟩, respectively, such that *r* relates *ra* to *rb* and such that *ra* refers to *a* and *rb* refers to *b*. Initializes the fields of *r*, *ra*, and *rb* with default values and returns *r*.
	Precondition:	*a* and *b* must refer to existing instances.
	Postcondition:	*r* has all fields initialized with default values and relates *ra* to *rb*. *ra* has all fields initialized with default values and refers to *a*. *rb* has all fields initialized with default values and refers to *b*.
`new R()`	**1. argument:**	Set⟨Element⟨A⟩⟩ *x* **or** Set⟨Extent⟨Element⟨A⟩⟩⟩ *x*
	2. argument:	Element⟨B⟩ *b* **or** Extent⟨Element⟨B⟩⟩ *b*
	Return:	Bag⟨Mold⟨R⟩⟩ *z*
	Description:	For every instance *a* in *x*, invoke `new R(a, b)`.
	Precondition:	*a* must not be empty, *b* must refer to an existing instance.
	Postcondition:	Postcondition of first variant holds analogously for each instance in *z*.
`new R()`	**1. argument:**	Set⟨Element⟨A⟩⟩ *x* **or** Set⟨Extent⟨Element⟨A⟩⟩⟩ *x*
	2. argument:	Set⟨Element⟨B⟩⟩ *y* **or** Set⟨Extent⟨Element⟨B⟩⟩⟩ *y*
	Return:	Bag⟨Mold⟨R⟩⟩ *z*
	Description:	For every instance *a* in *x* and for every instance *b* in *y*, invoke `new R(a, b)`.
	Precondition:	*a* and *b* must not be empty.
	Postcondition:	Postcondition of first variant holds analogously for each instance in *z*.

Table 3.16: Built-in relationship mold constructor. Overloaded variants with permuted argument types are not shown, but supported analogously.

tors require that the instances that are to be related by the relationship mold instance are provided as arguments. Depending on the types of the relationship participants, these arguments are either element instances or extent instances. For convenience, sets of element instances and sets of extent instances can be provided as arguments too, supporting query expressions as arguments. Table 3.16 lists the various overloaded variants of relationship mold constructors. For simplicity, the table omits those overloaded variants that are only permutations of other variants with regard to the constructor arguments. As detailed by Table 3.16, an invocation of a relationship mold constructor `new R()` does not only result in the creation of a relationship mold instance of type Mold⟨R⟩ but also in the creation of the corresponding role mold instances of type Mold⟨R.a⟩ and Mold⟨R.b⟩. The fields of all created mold instances are initialized with default values. If default initialization is not appropriate, customized initializers can be defined.

Element instances are created from mold instances by using the Rumer built-in *addition operators* (see Table 3.17 and Table 3.18). Addition operators exist for the creation of entity element instances and relationship element instances.

As detailed by Table 3.17, an entity element instance of type Element⟨E⟩ is created by invoking the built-in *entity addition operator* `add()` on an entity extent instance *t* of type Extent⟨Element⟨E⟩⟩. The `add()` operator takes an entity mold instance *m* of type

Operator for an entity E		
`add()`	**Target:**	Extent⟨Element⟨E⟩⟩ *t*
	Argument:	Mold⟨E⟩ *m*
	Return:	Element⟨E⟩ *e*
	Description:	Creates a unique instance *e* of type Element⟨E⟩, initializes *e*'s fields with *m*'s field values, adds *e* to *t*, and returns *e*.
	Precondition:	*m*'s fields must satisfy the invariant of Element⟨E⟩.
	Postcondition:	*e* is unique, has the same field values as *m*, satisfies its invariant, and resides in *t*.

Table 3.17: Built-in entity addition operator.

Mold⟨E⟩ as argument. It creates a unique entity element instance *e* of type Element⟨E⟩, initializes *e*'s fields with the field values of the argument mold instance *m*, adds *e* to the target extent instance *t*, and returns *e*. As a precondition, the `add()` operator requires that the field values of the argument entity mold instance *m* satisfy the invariant of Element⟨E⟩. As a result, the `add()` operator guarantees that the newly created entity element instance *e* satisfies its invariant.

The instantiation of relationship element instances is similar to the instantiation of entity element instances. Table 3.18 lists the built-in Rumer *relationship addition operator*. There exist two overloaded variants of the operator: one taking a relationship mold instance as an argument and one taking a bag of relationship mold instances as an argument. We review the semantics of the relationship addition operator based on the first variant; the explanations apply analogously to the second variant. A relationship element instance of type Element⟨R⟩ is created by invoking the built-in `add()` operator on a relationship extent instance *t* of type Extent⟨Element⟨R⟩⟩. The `add()` operator takes a relationship mold instance *mr* of type Mold⟨R⟩ as argument. It creates a unique relationship element instance *r* of type Element⟨R⟩, initializes *r*'s fields with the field values of the argument mold instance *mr*, adds *r* to the target extent instance *t*, and returns *r*. Furthermore, the operator checks, for each role mold instance mr.a and mr.b referred to by *mr*, whether a corresponding role element instance needs to be instantiated. This is the case whenever there does not yet exist a corresponding role element instance that is related by a relationship element instance that resides in the target extent instance *t*. In this case, a unique instance *ra* or *rb* of the corresponding role element type Element⟨R.a⟩ or Element⟨R.b⟩, respectively, is created, initialized with the field values of the corresponding mold instance, and related to the same instance as the one referred to by the corresponding mold instance. Finally, the operator relates the newly created relationship element instance *r* to its corresponding, possibly newly created, role element instances *ra* and *rb*. Besides expecting the field values of the mold instances *mr*, *mr.a*, and *mr.b* to satisfy the invariant of Element⟨R⟩, a relationship addition operator requires that there must not exist an instance Element⟨R⟩ r' in the target extent instance *t* that relates the instances referred to by *mr.a* and *mr.b*. The latter requirement guarantees that a relationship element instance is identified by its participating instances (see System Invariant 3.4 below).

The instantiation and initialization pattern enforced by the Rumer built-in language op-

Operator for a relationship R with participant base types A a and B b		
`add()`	**Target:**	Extent⟨Element⟨R⟩⟩ *t*
	Argument:	Mold⟨R⟩ *mr*
	Return:	Element⟨R⟩ *r*
	Description:	Creates a unique instance *r* of type Element⟨R⟩, initializes *r*'s fields with *mr*'s field values, and adds *r* to *t*. If there exists an instance Element⟨R.a⟩ *ra* that is referred to by an instance in *t* and that refers to the same instance of *BaseType*(A) *a* as *mr.a*, then relates *r* to *ra* and updates *ra*'s fields with *mr.a*'s field values. Otherwise, creates a unique instance *ra* of type Element⟨R.a⟩, initializes *ra*'s fields with *mr.a*'s field values, and relates *ra* to the instance referred to by *mr.a* as well as *r* to *ra*. If there exists an instance Element⟨R.b⟩ *rb* that is referred to by an instance in *t* and that refers to the same instance of *BaseType*(B) *b* as *mr.b*, then relates *r* to *rb* and updates *rb*'s fields with *mr.b*'s field values. Otherwise, creates a unique instance *rb* of type Element⟨R.b⟩, initializes *rb*'s fields with *mr.b*'s field values, and relates *rb* to the instance referred to by *mr.b* as well as *r* to *rb*. Returns *r*.
	Precondition:	The fields of *mr*, *mr.a*, and *mr.b* must satisfy the invariant of Element⟨R⟩ and *mr.a* and *mr.b* must refer to existing instances. There must not yet exist an instance Element⟨R⟩ *r'* in *t* that relates the instances referred to by *mr.a* and *mr.b*.
	Postcondition:	*r* is unique, relates *ra* to *rb*, has the same field values as *mr*, satisfies its invariant, and resides in *t*. *ra* and *rb* are both unique and have the same fields values as *mr.a* and *mr.b*, resp.
`add()`	**Target:**	Extent⟨Element⟨R⟩⟩ *t*
	Argument:	Bag⟨Mold⟨R⟩⟩ *x*
	Return:	Set⟨Element⟨R⟩⟩ *z*
	Description:	For every instance *mr* in *x*, invoke `t.add(mr)`.
	Precondition:	Precondition of first variant must hold analogously for each instance in *x*.
	Postcondition:	Postcondition of first variant holds analogously for each instance in *z*.

Table 3.18: Built-in relationship addition operator.

erators for element instances gives rise to a number of system invariants. The fact that entity and relationship elements can only be instantiated by adding a mold instance to a corresponding extent instance (see Table 3.14 and Table 3.16) establishes the following system invariant:

System Invariant 3.2 (Element containment): Consider an entity or relationship type T. For every element instance *t* of type Element⟨T⟩, there exists an extent instance *tx* of type Extent⟨Element⟨T⟩⟩ such that *t* resides in *tx*.

System Invariant 3.2 guarantees that an existing entity or relationship element instance always resides in an extent instance. As a result, query expressions in Rumer only quantify over allocated instances (see also discussion in Section 3.2.3). This restriction combined with the availability of mold types for instance initialization guarantees that programmers are given a chance to properly initialize instances before querying them.

The postconditions of the entity and relationship addition operators (see Table 3.17 and Table 3.18) guarantee to create new element instances that are unique. These postconditions establish the following system invariant:

System Invariant 3.3 (Element distinctness): Consider an entity or relationship type T and an extent instance *tx* of type Extent⟨Element⟨T⟩⟩. The element instances that reside in *tx* are distinct. Furthermore, if T is a relationship type, then, for any of its role types T.ab and for any two element instances t_1 and t_2 residing in *tx*, the role element instances $t_1.ab$ and $t_2.ab$ are distinct, unless they refer to the same participating instance.

The precondition of the relationship addition operator (see Table 3.18) further establishes the following system invariant:

System Invariant 3.4 (Relationship element weak identity): Consider a relationship type R with the participant base types A and B and consider an extent instance *rx* of type Extent⟨Element⟨R⟩⟩. For any element instance *r* that reside in *rx* and its participating instances *a* and *b* of $BaseType(A)$ and $BaseType(B)$, respectively, if there exists an element instance r' that reside in *rx* and that also has the participating instances *a* and *b*, then r' must be *r*.

System Invariant 3.4 guarantees that a relationship element instance is uniquely identified by the extent instance in which it resides and by its participating instances. This requirement is in line with Invariant 2 in RelJ [3][3].

Extents

Extent instances are created by using the Rumer built-in *extent constructors* (see Table 3.19 and Table 3.20). Unlike the creation of element instances, the creation of extent instances is not based on mold types. Rumer does not support "meta" extents for extent instances, to coalesce the existing extent instances of a type. As a result, extent instances are not retrievable by queries, and the indirection of a mold is not necessary to guarantee that extent instances are properly initialized upon instantiation and before retrieval. As detailed by Table 3.19, an entity extent instance of type Extent⟨Element⟨E⟩⟩ is created by invoking the built-in *entity extent constructor* `new Extent<E>()`. The constructor creates a unique, empty entity extent instance *ex* of type Extent⟨Element⟨E⟩⟩, initializes *ex*'s fields with default values, and returns *ex*. The creation of a relationship extent instance is analogous and detailed in Table 3.20. An extent constructor must guarantee that the newly created and returned extent instance satisfies its invariant. Depending on the concrete extent invariant declaration, customized initialization of extent fields and customized extent population may be necessary. To this end, programmers are allowed to overload default extent constructors.

Entity extent constructors and relationship extent constructors (see Table 3.19 and Table 3.20) guarantee that extent instances are distinct:

System Invariant 3.5 (Extent distinctness): Consider an entity or relationship type T. The extent instances of type Extent⟨Element⟨T⟩⟩ are distinct.

[3]Note, however, that RelJ only supports singleton extents

Operator for an entity E		
`new Extent<E>()`	**Return:**	Extent⟨Element⟨E⟩⟩ *ex*
	Description:	Creates a unique instance *ex* of type Extent⟨Element⟨E⟩⟩, initializes *ex*'s fields with default values, and returns *ex*.
	Postcondition:	*ex* is unique, empty, has all fields initialized with default values, and satisfies its invariant.

Table 3.19: Built-in default entity extent constructor.

Operator for a relationship R with participant base types A a and B b		
`new Extent<R>()`	**Return:**	Extent⟨Element⟨R⟩⟩ *rx*
	Description:	Creates a unique instance *rx* of type Extent⟨Element⟨R⟩⟩, initializes *rx*'s fields with default values, and returns *rx*.
	Postcondition:	*rx* is unique, empty, has all fields initialized with default values, and satisfies its invariant.

Table 3.20: Built-in default relationship extent constructor.

The addition operators (see Table 3.17 and Table 3.18) are the only operators to populate extent instances and to instantiate role element instances. Since those operators only accept mold instances as arguments (as opposed to element instances), it is impossible to add or associate the same element instance or role element instance, respectively, multiple times to/with the same or a different extent instance. As a result, the following system invariant holds:

System Invariant 3.6 (Extent disjointness): Consider an entity or relationship type T. The extent instances of type Extent⟨Element⟨T⟩⟩ are pairwise disjoint. Furthermore, if T is a relationship type, then, for any of its role types T.ab and for any two element instances t_1 and t_2 of type Element⟨T⟩, the role element instances $t_1.ab$ and $t_2.ab$ can only be the same if t_1 and t_2 reside in the same extent instance.

System Invariant 3.6 guarantees, together with System Invariant 3.2 and System Invariant 3.3, that *(i)* an entity or relationship element instance resides in exactly one extent instance, *(ii)* that a role element instance is associated with exactly one extent instance, and *(iii)* that an element instance (entity, relationship, or role) is unique across extent instances.

3.4.2 Instance destruction

As opposed to mold instances and extent instances, element instances can be "retrieved" based on the values of their fields by issuing an appropriate query. The existence of references to an element instance is thus not the right criterion to decide whether an element instance will be accessed in the future. As a result, element instances have to be deleted manually by the programmer. Mold instances and extent instances, on the other hand, are automatically reclaimed by a garbage collector.

Operator for an entity E		
`remove()`	**Target:**	Extent⟨Element⟨E⟩⟩ *t*
	Argument:	Element⟨E⟩ *e*
	Description:	If *e* is in *t*, then schedules *e* for deletion.
	Postcondition:	If *e* is in *t*, then *e* is scheduled for deletion.
`remove()`	**Target:**	Extent⟨Element⟨E⟩⟩ *t*
	Argument:	Set⟨Element⟨E⟩⟩ *x*
	Description:	For every instance *e* in *x*, invoke `t.remove(e)`.
	Postcondition:	Postcondition of first variant holds analogously for each instance in *x*.

Table 3.21: Built-in entity removal operator.

The Rumer programming language provides built-in *removal operators* (see Table 3.21 and Table 3.22) for the destruction of element instances. As detailed by Table 3.21 and Table 3.22, an element instance is destructed by removing the instance from the target extent instance in which the element instance resides. The element instance to-be destructed is passed as an argument to the removal operator. Both the entity removal operator and the relationship removal operator accept as arguments element instances and sets of element instances of the entity or relationship, respectively. In addition, the relationship removal operator allows the destruction of a relationship element instance by indicating its participating instances.

The Rumer built-in removal operators do have a *non-strict execution* semantics. To prevent a relationship element instance from relating nonexistent instances, an instance is only destructed once it participates no longer in any relationship. An invocation of `remove()` on an extent instance *tx* of type Extent⟨Element⟨T⟩⟩ requiring the destruction of the element instance *t* of type Element⟨T⟩ will mark the element instance *t* as "to be scheduled for deletion". For an entity or relationship element instance, "to be scheduled for deletion" means that the instance will be destructed as soon as there no longer exists any relationship element instance in which the instance participates. For a role element instance, "to be scheduled for deletion" means that the role element instance will be destructed as soon as all its associated relationship element instances are destructed. This treatment guarantees that a role element instance exists as long as there exists at least one relationship element instance of which the role element instance is a participant. The non-strict execution semantics of the removal operators requires programmers to unfold removal operations in a top-down manner, starting at the to be deleted element instance of which all remaining to be deleted element instances are transitive participants.

Mold instances and extent instances, on the other hand, are automatically reclaimed by a garbage collector. Mold instances are reclaimed if they are no longer reachable. Extent instances are reclaimed if they are no longer referred to by a variable or no longer participate in a relationship element instance and if none of the extent instance's element instances participate in a relationship element instance. The collection of an extent instance results not only in the destruction of the extent instance itself but also in the destruction of any element instances comprised in the extent instance. As a result, a form of

Operator for a relationship R with participant base types A a and B b		
`remove()`	**Target:**	Extent⟨Element⟨R⟩⟩ *t*
	Argument:	Element⟨R⟩ *r*
	Description:	If *r* is in *t*, then schedules *r*, *r.a*, and *r.b* for deletion.
	Postcondition:	If *r* is in *t*, then *r*, *r.a*, and *r.b* are scheduled for deletion.
`remove()`	**Target:**	Extent⟨Element⟨R⟩⟩ *t*
	Argument:	Set⟨Element⟨R⟩⟩ *x*
	Description:	For every instance *r* in *x*, invoke `t.remove(r)`.
	Postcondition:	Postcondition of first variant holds analogously for each instance in *x*.
`remove()`	**Target:**	Extent⟨Element⟨R⟩⟩ *t*
	1. argument:	Element⟨A⟩ *a* **or** Extent⟨Element⟨A⟩⟩ *a*
	2. argument:	Element⟨B⟩ *b* **or** Extent⟨Element⟨B⟩⟩ *b*
	Description:	For the instance *r* in *t* with *r.a* = *a* and *r.b* = *b*, invoke `t.remove(r)`.
	Postcondition:	Postcondition of first variant holds analogously for the instance *r* in *t* with *r.a* = *a* and *r.b* = *b*.
`remove()`	**Target:**	Extent⟨Element⟨R⟩⟩ *t*
	1. argument:	Set⟨Element⟨A⟩⟩ *x* **or** Set⟨Extent⟨Element⟨A⟩⟩⟩ *x*
	2. argument:	Element⟨B⟩ *b* **or** Extent⟨Element⟨B⟩⟩ *b*
	Description:	For every instance *a* in *x* and for every instance *r* in *t* with *r.a* = *a* and *r.b* = *b*, invoke `t.remove(r)`.
	Postcondition:	Postcondition of first variant holds analogously for every instance *a* in *x* and for every instance *r* in *t* with *r.a* = *a* and *r.b* = *b*.
`remove()`	**Target:**	Extent⟨Element⟨R⟩⟩ *t*
	1. argument:	Set⟨Element⟨A⟩⟩ *x* **or** Set⟨Extent⟨Element⟨A⟩⟩⟩ *x*
	2. argument:	Set⟨Element⟨B⟩⟩ *y* **or** Set⟨Extent⟨Element⟨B⟩⟩⟩ *y*
	Description:	For every instance *a* in *x* and for every instance *b* in *y* and for every instance *r* in *t* with *r.a* = *a* and *r.b* = *b*, invoke `t.remove(r)`.
	Postcondition:	Postcondition of first variant holds analogously for every instance *a* in *x* and for every instance *b* in *y* and for every instance *r* in *t* with *r.a* = *a* and *r.b* = *b*.

Table 3.22: Built-in relationship removal operator. Overloaded variants with permuted argument types are not shown, but supported analogously.

implicit automatic collection also happens for element instances. However, this form of automatic collection of element instances is admissible (and also desirable) since those element instances are non-participating and since the unreachability of the extent instance prevents any retrieval of its comprised element instances.

The availability of removal operators for the destruction of element instances is vital for the prevention of memory leaks. Similar observations have been made by Østerbye [52] in the context of the Noiai association library and by Vaziri et al. [55] and Nelson et al. [57] in the context of defining object identity based on relation types. These works essentially rely on automatic garbage collection for instance destruction. To prevent memory leaks association, linkages and relation type tuples use weak references to refer to their participating objects and key objects, respectively, and the underlying implementations intercept the garbage collector to reclaim linkages and relation type tuples, respectively, that refer to "garbage" objects. Furthermore, both Vaziri et al. [55] and Nelson et al. [57]

support the notion of a scope and space, respectively, to prevent the accidental collection of relation type tuples whose identity relies on primitive types only. A scope (or space) is conceptually similar to an extent instance since tuples residing in a scope cannot be collected as long as the containing scope is referenced.

The postconditions of the removal operators (see Table 3.21 and Table 3.22) guarantee that an element instance will not be removed as long as it participates in a relationship element instance. As a result, the following system invariant holds:

System Invariant 3.7 (Referential integrity): Consider a relationship type R with the participant base types A and B. For any element instance r of type Element⟨R⟩, there exist two instances a and b of $BaseType(A)$ and $BaseType(B)$, respectively, which r relates.

3.4.3 Heap querying

The fact that Rumer query expressions have a non-strict evaluation semantics and are side-effect free (see Section 3.2.3) gives rise to the following system invariant:

System Invariant 3.8 (Predicate semantics): Consider a query expression x or a variable x bound to a query expression by call-by-name. Upon each read of x, the returned value is equal to the evaluation of the query expression in the state of the program heap at the time of reading x.

System Invariant 3.8 thus guarantees that query expressions or query variables become true predicates over the extents they are applied to.

4 The Matryoshka principle

The Rumer programming language embodies a modularization discipline that we summarize as the *Matryoshka Principle*. We name the principle after the Russian nesting dolls since it leads to a *stratification* of Rumer programming language abstractions. Based on this stratification, the principle defines admissibility criteria that stipulate restrictions on writes to locations, the expression of invariants, and routine invocations. In this chapter, we first discuss the rationale underlying the Matryoshka Principle and its embodiment in the Rumer programming language (Section 4.1). Then, we introduce the stratification caused by the Matryoshka Principle (Section 4.2), discuss the imposed admissibility criteria (Section 4.3), and conclude with a discussion of the principle's underlying design decisions and its connection to object ownership (Section 4.4).

4.1 Rationale

An important aim of the design of the Rumer programming language is to facilitate the *modular verification* of Rumer programs. A modular verification technique allows Rumer entity and relationship declarations to be verified independently from each other and is crucial for scaling the verification technique to real-world seized programs.

To facilitate modular verification, a programming language has to provide strong *encapsulation* guarantees. In particular, it should be possible to modularly reason about state changes of an instance and to modularly determine the invariants that are vulnerable to the execution of a routine. Invariants can be violated by writing to the instance on which the invariant is imposed. Traditionally [82, 21, 10, 16], a routine is allowed to invalidate its receiver instance's invariant for the duration of the routine execution. Such a regime facilitates *data type induction* [20, 91, 92] as a proof technique: each routine may assume that the invariant holds in the routine's initial state, provided that each routine ensures that the invariant holds in the routine's final state. However, as pointed out by Huizing and Kuiper [11], data type induction cannot be soundly adopted for routines that potentially lead to a *call-back*. A call-back occurs if a routine *r()*, executing on an instance *o*, invokes either directly or transitively (by further routine invocations) a routine *t()* on the original instance *o*. Since *t()* re-enters *o* in a state in which the invariant of *o* may be temporarily

broken, *t()* cannot soundly assume *o*'s invariant in *t()*'s initial state.

To restore data type induction soundness, a programming language must either prevent call-backs or its verification technique must impose additional proof obligations on the caller of a routine. These obligations require the calling routine *r()* to re-establish, before the invocation, the invariants of its possibly transitive receiver instances whose invariants are vulnerable to *r()*'s execution. In either case, the transitive receivers of a routine invocation must be identified at compile-time. Without further restrictions enforced by the programming language, however, those receivers cannot be modularly determined at compile-time. Moreover, prohibiting call-backs altogether would be too limiting a restriction as it would rule out direct call-backs (including recursive invocations) too.

First-class relationships not only build the object graph declaratively but also describe the possible structures of the program heap at compile-time. In the design of the Rumer programming language, we leverage this property for modular program verification. We enforce a modularization discipline, the *Matryoshka Principle*, that requires any modifications of a program's heap to comply with the structure prescribed by relationship declarations. To facilitate sound, modular reasoning about invariants, in particular, we require relationship declarations to be *acyclic*. This requirement results in a *stratification* of invariants and of the program heap and facilitates the modular determination of the vulnerable invariants and the prevention of transitive call-backs, respectively. In this chapter, we introduce the Matryoshka Principle and its entailed stratification and enforced encapsulation guarantees. In Chapter 5, we introduce our modular verification technique that builds on the guarantees made the the Matryoshka Principle.

4.2 Stratification

The `participants` clauses of relationship declarations restrict how instances may relate to each other and reveal how entities and relationships are composed to build a software system. The `participants` clauses of a Rumer program define a *relation* from the set of relationship declarations of the program to the set of entity and relationship declarations of the program that are participants of relationships. The resulting relation thus is a relation between base types (see Definition 3.1 on page 62). For example, the `participants` clauses of the Rumer university program shown in Figure 3.3 on page 43 define the following relation:

$$\{\mathit{Attends} \mapsto \mathit{Student}, \mathit{Attends} \mapsto \mathit{Course}, \mathit{Assists} \mapsto \mathit{Student}, \mathit{Assists} \mapsto \mathit{Course},\\ \mathit{Teaches} \mapsto \mathit{Faculty}, \mathit{Teaches} \mapsto \mathit{Course}\}$$

The `participants` clauses of the Rumer Composite program in Figure 3.5 on page 45 define the following relation:

$$\{\mathit{Composite} \mapsto \mathit{Parent}, \mathit{Composite} \mapsto \mathit{Component}, \mathit{Parent} \mapsto \mathit{Component}\}$$

The Matryoshka Principle requires that the transitive closure of the relation defined by the `participants` clauses of a Rumer program forms a *strict partial order*. As illustrated by Figure 4.1, this requirement results in a *stratification* of programming language abstractions. Figure 4.1 depicts a schematic illustration of the Rumer stratification layers based on the Composite heap shown in Figure 3.7 on page 48 (see Composite program in Figure 3.5 on page 45). As opposed to Figure 3.7, Figure 4.1 depicts each base type of the Composite program in addition to the extent instances and element instances of those types. Figure 4.1 uses the same graphical notation as Figure 3.7 and displays base types as gray, rounded rectangles. It displays four such rectangles, one for the entity type `Component`, two for the relationship types `Parent` and `Composite`, and one for the application singleton type `CompositePattern`. Apart from the rectangle representing the application singleton type `CompositePattern`, each of these rectangles contains the extent instances and element instances shown in Figure 3.7. To keep the graphic well-arranged, Figure 4.1 only hints at the connecting lines between participant instances and relationship element instances and depicts role element instances as part of relationship extent instances.

Figure 4.1 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* abstraction of its participants, and conversely, the participants become a *lower* abstraction of their relationships. Figure 4.1 indicates further that an application declaration becomes the topmost abstraction since it denotes the program entry point. As a result, the participants relation of a Rumer program also includes a tuple for the program's application. For example, the participants relation for the Rumer Composite program shown in Figure 3.5 on page 45 becomes:

$$\{\mathit{CompositePattern} \mapsto \mathit{Composite}, \mathit{Composite} \mapsto \mathit{Parent}, \mathit{Composite} \mapsto \mathit{Component}, \\ \mathit{Parent} \mapsto \mathit{Component}\}$$

The requirement imposed by the Matryoshka Principle, enforcing that the transitive closure of the participants relation of a Rumer program forms a strict partial order, gives rise to the following system invariant:

System Invariant 4.1 (Acyclicity): The transitive closure of the relation defined by the `participants` clauses of a Rumer program forms a strict partial order, guaranteeing that relationship declarations are acyclic.

The stratification induced by the Matryoshka Principle is enforced at compile-time, providing some liberalness with regard to the actual heap structure. As we detail in Section 4.4.2, the Matryoshka Principle bears resemblance to ownership type systems [93, 94, 95, 12, 96, 97, 98, 99, 100] since ownership type systems also impose a certain structure on a program's heap. Besides the difference in the imposed heap structure (see Section 4.4.2), ownership type systems also differ from the Matryoshka Principle in that they enforce the heap structure at run-time, but do generally not allow an instance to change its

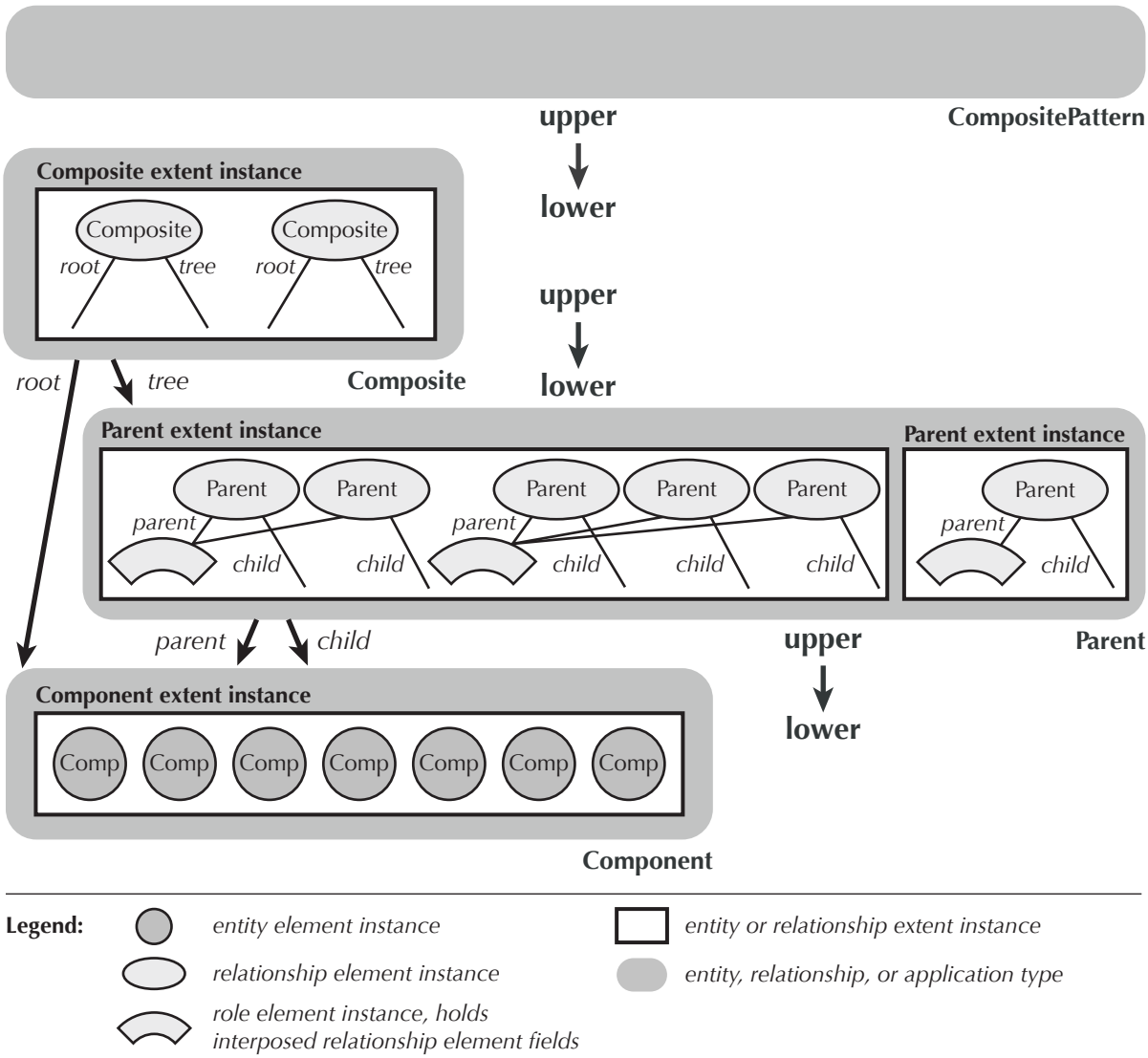

Figure 4.1: Schematic illustration of the stratification of Rumer language abstractions based on the heap snapshot of the Rumer Composite program shown in Figure 3.7 on page 48.

position in the heap structure at run-time. The Matryoshka Principle, on the other hand, allows an instance to change its participation in a relationship at run-time.

4.3 Admissibility criteria

In this section, we introduce the admissibility criteria imposed on a Rumer program by the Matryoshka Principle. The admissibility criteria provide strong encapsulation guarantees for Rumer types. They can be checked statically and are enforced by the Rumer compiler.

The admissibility criteria rely on the stratification of language abstractions resulting from the `participants` clauses of a Rumer program and stipulate restrictions that define *(i)* which routines 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 routine can invoke a routine. Locations in Rumer are application global variables, content

Type		Location	Statement
Application		global variable	assignment
Relationship	**extent**	content field	addition, removal assignment
	mold	non-interposed field interposed field	assignment assignment
	element	non-interposed field interposed field	assignment assignment
Entity	**extent**	content field	addition, removal assignment
	mold	field	assignment
	element	field	assignment

Table 4.2: Programmer-definable locations and program statements that can write to these locations. Table 4.3 details as part of which routines those statements may occur.

and fields of extent instances, fields of element instances, and fields of mold instances. Table 4.2 lists all possible Rumer locations and indicates for each location by which program statement it can be written to. As detailed by Table 4.2, fields of instances can be written to by assignments and the content of extent instances can be written to by invoking the built-in addition or removal operators (see Section 3.4.1 on page 65) on the extent instance. From a pure implementation perspective, invocations of the built-in addition or removal operators are routine invocations, rather than primitive statements. However, since those operators are system-provided and have a well-defined semantics, we treat them as primitive statements from the point of view of the Matryoshka Principle and its associated verification technique (see Chapter 5).

4.3.1 Admissible writes

To provide strong encapsulation of Rumer types, the Matryoshka Principle allows a routine to write at most to the locations of the routine's current receiver instance. This requirement facilitates modular reasoning about the state changes of an instance.

Table 4.3 details, for each routine kind in Rumer, to which locations the routine is allowed to write. Table 4.3 states the admissible locations relative to the current receiver instance of a routine invocation. For example, a relationship extent method for a relationship type R with the participant base types A and B and the role identifiers a and b, respectively, has the newly created extent instance *rx* of type Extent⟨Element⟨R⟩⟩ as current receiver instance. This instance can be referred to by the `these` keyword. As indicated by Table 4.3, a relationship extent method can only write to the content of the current relationship extent instance *rx*, to the fields of *rx*, and to the non-interposed and interposed fields of any relationship element instance residing in *rx*. For relationship extent constructors and default relationship extent constructors, the writable locations are identical, with the

Executing routine	Current receiver	Writable location
Application App		
action	App *app*	Global variables of *app*.
Relationship R with participant base types A a and B b		
default extent constructor	Extent⟨Element⟨R⟩⟩ *rx*	Fields of *rx*.
extent constructor	Extent⟨Element⟨R⟩⟩ *rx*	Content of *rx*.
		Fields of *rx*.
		For any instance Element⟨R⟩ *r* in *rx*: fields of *r*, fields of *r.a*, and fields of *r.b*.
extent method	Extent⟨Element⟨R⟩⟩ *rx*	Content of *rx*.
		Fields of *rx*.
		For any instance Element⟨R⟩ *r* in *rx*: fields of *r*, fields of *r.a*, and fields of *r.b*.
non-interposed mold constructor	Mold⟨R⟩ *mr*	Fields of *mr*, fields of *mr.a*, and fields of *mr.b*.
non-interposed mold initializer	Mold⟨R⟩ *mr*	Fields of *mr*, fields of *mr.a*, and fields of *mr.b*.
non-interposed element method	Element⟨R⟩ *r*	Fields of *r*, fields of *r.a*, and fields of *r.b*.
interposed element method	Element⟨R._⟩ *rab*	Fields of *rab*.
Entity E		
default extent constructor	Extent⟨Element⟨E⟩⟩ *ex*	Fields of *ex*.
extent constructor	Extent⟨Element⟨E⟩⟩ *ex*	Content of *ex*.
		Fields of *ex*.
		Fields of any instance Element⟨E⟩ *e* in *ex*.
extent method	Extent⟨Element⟨E⟩⟩ *ex*	Content of *ex*.
		Fields of *ex*.
		Fields of any instance Element⟨E⟩ *e* in *ex*.
mold constructor	Mold⟨E⟩ *me*	Fields of *me*.
mold initializer	Mold⟨E⟩ *me*	Fields of *me*.
element method	Element⟨E⟩ *e*	Fields of *e*.

Table 4.3: Locations to which executing routine is allowed to write. This table instantiates parameter $\mathbb{U}$ of the unified framework for visible-state verification techniques (see Section 5.1 on page 89). See Table 4.2 for summary of locations and corresponding write statements.

exception that default relationship extent constructors operate on empty extent instances and thus do not alter any element fields. The admissible writes for entity extent methods and (possibly default) entity extent constructors are analogous. As a result, Table 4.3 guarantees for extent instances that state changes of the extent instance as well as state changes of any element instances comprised in the extent instance are fully encapsulated by the methods of the type declaring the extent type.

The guarantees are similar for element instances and mold instances. Given the restric-

tions of Table 4.3, state changes of an element instance or mold instance are fully encapsulated by the declaring type's element methods and mold constructors or initializers, respectively. Table 4.3 also confirms that interposed relationship element fields are encapsulated by their declaring relationships. As indicated by Table 4.3, only routines of the declaring relationship can write to interposed fields, but not the routines of the participant supertype of the role type into which the fields are interposed. As discussed in previous work [101], this property makes member interposition attractive for the modular verification of invariants over shared state. We elaborate on this in Section 7.1.3 on page 200. Similarly, Table 4.3 indicates that a role type cannot write to the fields inherited from its participant supertype, guaranteeing that those fields are encapsulated by the declaring participant supertype. Table 4.3 further requires that an application's global variables can only be written to by the application's actions and thus guarantees that an application's global variables are encapsulated by the application.

According to Table 4.3, a relationship element instance's role references, by which the relationship element instance refers to its participating instances, are immutable. The creation and destruction of element instances can only by achieved by the designated addition and removal operators introduced in Section 3.4.1 on page 65 and Section 3.4.2 on page 71, respectively. According to Table 4.2, those operators write to the content of an entity or relationship extent instance, and according to Table 4.3, invocations of such operators can only occur as part of extent methods and extent constructors of the extent instance's base type. As a result, the instantiation and destruction of element instances is encapsulated by the extent methods and extent constructors of the element instance's base type.

A program's compliance with Table 4.3 can be checked at compile-time since the admissible writes are stated relative to the current receiver instance of an executing routine. For example, for the Composite pattern program displayed in Figure 3.11 on page 57, the invocation of the addition operator on line 15 is admissible since the addition operator writes to the `Parent` relationship extent instance that is the current receiver of the extent method `append()`. Similarly, the assignment to the interposed element field `total` on line 18 is admissible since `x` refers to an instance of type Element⟨Parent.parent⟩ that resides in the current extent receiver instance of the extent method `append()`. Likewise, the field assignments as part of the initializer declarations of the university program shown in Figure 3.15 on page 66 are admissible since they write to the `Student` mold instance that is the current receiver instance of the initializer.

4.3.2 Admissible invariants

To facilitate the modular verification of invariants, the Matryoshka Principle allows an invariant to depend at most on the locations encapsulated by the invariant-declaring type. As a result, it can be modularly determined which invariants are vulnerable to the execution of a routine.

Invariant	Instance of invariant	Referable location
Application App		
invariant	App *app*	Global variables of *app*.
Relationship R with participant base types A a and B b		
extent invariant	Extent⟨Element⟨R⟩⟩ *rx*	Content of *rx*.
		Fields of *rx*.
		For any instance Element⟨R⟩ *r* in *rx*: fields of *r*, fields of *r.a*, and fields of *r.b*.
element invariant	Element⟨R⟩ *r*	Fields of *r*, fields of *r.a*, and fields of *r.b*.
Entity E		
extent invariant	Extent⟨Element⟨E⟩⟩ *ex*	Content of *ex*.
		Fields of *ex*.
		Fields of any instance Element⟨E⟩ *e* in *ex*.
element invariant	Element⟨E⟩ *e*	Fields of *e*.

Table 4.4: Locations on which an invariant is allowed to depend. This table indirectly instantiates parameter $\mathbb{D}$ of the unified framework for visible-state verification techniques (see Section 5.1 on page 89). See Table 4.2 for a summary of locations and corresponding write statements.

Table 4.4 details for each programmer-definable invariant on which locations the invariant is allowed to depend. Table 4.4 displays a row for each of the five invariant categories introduced in Section 3.2.5 on page 55. These categories are: application invariant, relationship extent invariant, relationship element invariant, entity extent invariant, and entity element invariant. Mold types are not included in this list since they are value types and no invariants can be declared for value types. Table 4.4 states the admissible invariants relative to the instance on which the invariant is imposed. For example, a relationship extent invariant for a relationship type R with the participant base types A and B and the corresponding role types R.a and R.b, respectively, is imposed on the extent instance *rx* of type Extent⟨Element⟨R⟩⟩. This instance can be referred to by the `these` keyword. As indicated by Table 4.4, such a relationship extent invariant can only depend on the locations that are encapsulated by the extent methods and extent constructors of the declaring relationship R. These locations include the content of the relationship extent instance *rx*, the fields of *rx*, and the non-interposed and interposed fields of any relationship element instance residing in *rx*. The admissibility criteria for entity extent invariants are analogous.

To guarantee that an invariant only depends on the locations to which the invariant's instance is also allowed to write, the restrictions enforced by Table 4.4 have to match those enforced by Table 4.3. For example, like an extent method, which can write to the content and fields of its current extent receiver instance as well as to the fields of the element instances residing in that extent instance, an extent invariant is allowed to depend on the content and fields of its current extent receiver instance as well as on the fields of the element instances residing in that extent instance. Similarly, only an invariant of the

relationship that declares an interposed field is allowed to depend on that field, but not any invariant of the participant supertype of the role type into which the field is interposed. And conversely, a relationship invariant is not allowed to depend on any fields of the instances of its role types' supertype.

A program's compliance with Table 4.4 can be checked at compile-time since the admissible invariants are stated relative to the instance on which the invariant is imposed. For example, in the case of the Composite pattern program shown in Figure 3.11 on page 57, the relationship extent invariant of `Parent` (line 21) is admissible since it only depends on the content of the invariant's `Parent` extent instance as well as on the interposed fields `total` of the element instances residing in that extent instance. These locations are encapsulated by the relationship `Parent`. The relationship element invariant of `Composite` (line 30), however, does not comply with the admissibility criterion specified in Table 4.4 as the invariant depends on locations of the invariant's element instance's tree role extent instance. On line 31, for example, the invariant refers to the child projection of the relation described by the `Parent` extent instance referred to by the `tree` role reference. This projection is encapsulated by `Parent`'s extent methods, but not by the methods of `Composite`. In Chapter 6, we return to this issue and show how we can extend Table 4.4 so as to accommodate the `Composite` element invariant. In Section 7.1.2 on page 193 we provide the correspondingly updated version of the Rumer Composite pattern specification.

4.3.3 Admissible invocations

To restore data type induction soundness for Rumer programs, the Matryoshka Principle guarantees the absence of transitive call-backs. A call-back, in general, can only occur if there exists a cycle in the reference structure underlying the call chain. Call-backs can be prevented by making this reference structure acyclic. As stated by System Invariant 4.1, relationship declarations in Rumer are guaranteed to be acyclic. Thus, in Rumer, call-backs can be prevented by requiring *routine invocations to propagate in the direction of a program's participants relation*.

Table 4.5 details how a less stringent version of the above requirement is implemented in Rumer. It lists, for the execution of a routine, on which instances the routine is allowed to invoke routines. For the execution of default extent constructors, mold constructors, and programmer-defined mold initializers, Table 4.5 prohibits any further routine invocations. Mold constructors and (possibly default) extent constructors have the newly instantiated instance as the current receiver instance. For the execution of the remaining routines, Table 4.5 establishes the following pattern: a routine executing on an instance o can invoke a routine *(a)* on its current receiver instance o, *(b)* on an element instance of $\mathit{BaseType}(o)$, if o is an extent instance, or on a role element instance of $\mathit{BaseType}(o)$, if o is a relationship element instance, *(c)* on an instance whose base type is a direct participant of $\mathit{BaseType}(o)$, if $\mathit{BaseType}(o)$ is a relationship, and *(d)* on a mold instance. As a result, Table 4.5 not only guarantees that routine invocations propagate in the direction of a pro-

gram's participant relation *(c)* but also allows for direct call-backs, including recursive invocations *(a)*. Furthermore, it allows routine invocations to propagate from an extent instance to an element instance of the same base type and from a relationship element instance to a role element instance of the same base type *(b)*. Table 4.5 does further not restrict invocations of mold constructors and mold initializers *(d)*. Since relationship declarations are acyclic (see System Invariant 4.1) and since extent types, element types, and role types are distinct (see Section 3.3 on page 59) and since mold constructors and mold initializers cannot invoke routines, the routine invocation scheme enforced by Table 4.5 prevents transitive call-backs. Thus, Table 4.5 gives rise to the following system invariant:

System Invariant 4.2 (No transitive call-backs): The admissibility criterion "admissible invocations" of the Matryoshka Principle guarantees the absence of transitive call-backs.

In terms of Figure 4.1, Table 4.5 guarantees that call chains are either *recursive* or propagate *inwards* (from an extent/element instance to an element/role instance of the same base type) or *downwards* (from an upper to its immediate lower layer). For example, the left `Parent` extent instance *px1* in Figure 4.1 may invoke routines on the following instances: *px1*, any `Parent` element instance that resides in *px1* or that resides in the right `Parent` extent instance *px2* in Figure 4.1, any `Parent.parent` role element instance related to a `Parent` element instance that resides in *px1* or in *px2*, the only `Component` extent instance *cx1* in Figure 4.1, any `Component` element instance that resides in *cx1*, any mold instance of base type `Parent`, `Component`, or `Composite`. The left `Parent` extent instance *px1* in Figure 4.1 could also instantiate a new `Component` extent instance by invoking `Component`'s default extent constructor. However, Table 4.5 forbids call chains to propagate horizontally or upwards. For example, the `Parent` extent instance *px1* is neither allowed to invoke a method on the `Parent` extent instance *px2* nor to instantiate a new `Parent` extent instance by invoking `Parent`'s default extent constructor, nor is it allowed to invoke any routine on an instance of base type `Composite` or base type `CompositePattern`.

A program's compliance with Table 4.5 can be checked at compile-time since a program's stratification is known at compile-time. For example, in the case of the Composite pattern program shown in Figure 3.5 on page 45, the invocation of the extent method `append()` on line 15 is admissible since it is invoked on the current `Composite` element receiver instance's tree `Parent` extent instance. This instance is of type Extent⟨Element⟨Parent⟩⟩, which is a direct participant of `Composite`. Also the invocations of the built-in query operators in the extent method `append()` of the Composite pattern program shown in Figure 3.11 on page 57 are admissible. Built-in query operators are unconstrained by the Matryoshka Principle since they are side-effect free.

Executing routine	Allowed callee
Application App	
action	Current receiver instance of type App.
	For any relationship or entity T, any instance o such that $BaseType(o) = \mathsf{T}$.
Relationship R with participant base types A a and B b	
default extent constructor	–
extent constructor	Current receiver instance of type Extent⟨Element⟨R⟩⟩.
	Any instance of type Element⟨R⟩, Element⟨R.a⟩, Element⟨R.b⟩, or any instance o such that $BaseType(o) = \mathsf{A}$ or $BaseType(o) = \mathsf{B}$.
	For any relationship or entity T, any instance of type Mold⟨T⟩.
extent method	Current receiver instance of type Extent⟨Element⟨R⟩⟩.
	Any instance of type Element⟨R⟩, Element⟨R.a⟩, Element⟨R.b⟩, or any instance o such that $BaseType(o) = \mathsf{A}$ or $BaseType(o) = \mathsf{B}$.
	For any relationship or entity T, any instance of type Mold⟨T⟩.
non-interposed mold constructor	–
non-interposed mold initializer	–
non-interposed element method	Current receiver instance of type Element⟨R⟩.
	Any instance of type Element⟨R.a⟩ or Element⟨R.b⟩, or any instance o such that $BaseType(o) = \mathsf{A}$ or $BaseType(o) = \mathsf{B}$.
	For any relationship or entity T, any instance of type Mold⟨T⟩.
interposed element method	Current receiver instance of type Element⟨R._⟩.
	Any instance o such that $BaseType(o) = \mathsf{A}$ or $BaseType(o) = \mathsf{B}$.
	For any relationship or entity T, any instance of type Mold⟨T⟩.
Entity E	
default extent constructor	–
extent constructor	Current receiver instance of type Extent⟨Element⟨E⟩⟩.
	Any instance of type Element⟨E⟩.
	For any relationship or entity T, any instance of type Mold⟨T⟩.
extent method	Current receiver instance of type Extent⟨Element⟨E⟩⟩.
	Any instance of type Element⟨E⟩.
	For any relationship or entity T, any instance of type Mold⟨T⟩.
mold constructor	–
mold initializer	–
element method	Current receiver instance of type Element⟨E⟩.
	For any relationship or entity T, any instance of type Mold⟨T⟩.

Table 4.5: Receivers on which executing routine is allowed to invoke routines. This table instantiates parameter $\mathbb{C}$ of the unified framework for visible-state verification techniques (see Section 5.1 on page 89).

4.4 Discussion

In this section, we reflect on the design decisions underlying the Matryoshka Principle and discuss how the Matryoshka Principle relates to object ownership.

4.4.1 Reflection on design decisions

The Matryoshka Principle enforces a *"lightweight"* modularization discipline. The discipline is lightweight as it only constrains *writes* to locations but not *reads* of locations. As a result, the Matryoshka Principle allows the declaration of variables or fields that point in the opposite direction of a program's participants relation[1]. As long as those references are only used for reading the locations they point to, the accesses are unconstrained by the Matryoshka Principle. However, the principle forbids any writes to the locations to which the references point as those writes may execute in inconsistent states. These restrictions reflect the primary design intent underlying the Matryoshka Principle, which is to facilitate modular reasoning about state change in sequential, relationship-based programs. If the principle were to accommodate concurrent programs as well, then it should also control read accesses to locations.

The Matryoshka Principle requires routine invocations to gradually propagate downwards a program's participants relation, but forbids "jumps" even if they conform to the direction of the participants relation. This restriction could be lifted without compromising the guarantees made by the principle. However, the support of "jumping" routine invocations would render the checking of a program's compliance with the Matryoshka Principle less modular. In the present version of the Matryoshka Principle, to check a program's compliance with the Matryoshka Principle a relationship only needs to know its immediate participant types. However, if "jumping" routine invocations were supported, then a relationship would need to know all its transitive participant types.

The stratification induced by the Matryoshka Principle is enforced at compile-time. A static enforcement of the desired heap stratification permits some liberalness with regard to the resulting stratification. In particular, it allows instances to change their "position" in the heap structure at run-time. However, a static enforcement of the heap stratification also comes at the price of a non-modular program well-formedness check, guaranteeing that a program's participants relations is acyclic (see System Invariant 4.1). In Section 7.2 on page 204, we discuss the elaboration of a module system as part of future work. Given a module system, the desired program stratification can be formulated relative to the program's modules, and the checking of the acyclicity of the program's participants relation should become modular with regard to a particular module.

[1] Any such element type variables or fields are implemented as weak references to facilitate their destruction upon executing a removal operation (see Section 3.4.2 on page 71)

4.4.2 Connections to ownership

The Matryoshka Principle leads to a stratification of a Rumer program's heap and guarantees strong encapsulation of the program's instances. From this perspective, the Matryoshka Principle bears resemblance to ownership type systems. Different flavors of ownership type systems have been developed (e.g., Ownership Types [93, 94, 95, 97] and Universe Types [12, 96, 98, 99, 100], see Section 6.4 on page 179 for further details), but they all structure a program's heap as an *ownership tree*, where each object is owned by exactly one other object, and they enforce *encapsulation policies*, which restrict the existence or use of references between objects [98].

The Matryoshka Principle most closely relates to Universe Types [12, 96, 98, 99, 100] since the Universe Type system does not constrain references that are only used for reading ("any" references) and prevents call-backs into owners from within owned objects. However, the Matryoshka Principle differs importantly from Universe Types (and other ownership type systems) in the enforced heap topology and encapsulation policy. Whereas every object in the Universe Type system has a unique owner, an entity or relationship instance in Rumer can participate in more than one relationship instance. The relationship instances in which an entity or relationship instance participates can further be of different relationship types. Moreover, entity and relationship instances can freely change their relationship participation at run-time[2]. The possibility to migrate from one owner to another ("ownership transfer"), on the other hand, is only supported by a number of ownership type systems (e.g., [102, 103]) and imposes uniqueness guarantees on references. The liberalness with regard to the resulting heap topology offered by the Matryoshka Principle requires a static enforcement of the heap topology. Ownership type systems, on the other hand, rely on the resulting topology to be enforced at run-time. As discussed in the previous section, this liberalness comes at the price of a non-modular program well-formedness check, guaranteeing that a program's participants relations is acyclic (see System Invariant 4.1). Given the differences in underlying heap topology, the encapsulation policies enforced on top of the heap topology fall out differently too. Whereas Universe Types enforce the *owner-as-modifier* discipline [96], which ensures that modifications of an object are initiated by the object's owner, the Matryoshka Principle does not impose such a restriction. Instead, the Matryoshka Principle allows modifications of an entity or relationship instance to be initiated by any of the relationship instances the very entity or relationship instance (possibly transitively) participates.

[2] As discussed in Section 3.4.2 on page 71, the Rumer removal operators are subject to referential integrity (see System Invariant 3.7 on page 74).

5 Verification technique

The guarantees made by the Matryoshka Principle facilitate the modular verification of Rumer programs. In this chapter, we introduce a visible-state verification technique for Rumer that leverages the stratification and admissibility criteria imposed on a Rumer program by the Matryoshka Principle. We first provide the necessary background information on visible-state verification techniques (Section 5.1). Then, we present our verification technique (Section 5.2) and prove its soundness (Section 5.3).

5.1 Background

The verification technique we developed for Rumer relies on the guarantees provided by the Matryoshka Principle. Since the admissibility criteria of the Matryoshka Principle (see Section 4.3 on page 78) restore data type induction soundness for Rumer programs, our verification technique applies data type induction as a proof technique. It thus falls in the category of *visible-state verification techniques* [16] that require instances to meet their invariants in the initial and final states of routine executions (i.e., the visible states) but allow instances to temporarily break their invariants during the execution of a routine.

To describe our verification technique, we use the unified framework for visible-state verification techniques introduced by Drossopoulou et al. [104]. The framework captures a visible-state verification technique in terms of seven *parameters*. These parameters are formulated relative to the execution of a routine *r* declared by a type *t* and denote either a set of invariants or a set of instances:

$\mathbb{X}_{(t,r)}$ set of invariants that are e**x**pected to hold in the initial and final state (i.e., visible states) of the execution of the routine *r* on an instance of type *t*.

$\mathbb{V}_{(t,r)}$ set of invariants that are **v**ulnerable to the execution of the routine *r* on an instance of type *t*. The executing routine *r* can break invariants either by writing to locations or by invoking routines that break invariants without re-establishing them.

$\mathbb{B}_{(t,r,s)}$ set of invariants that must be proven to hold **b**efore invoking a routine on a

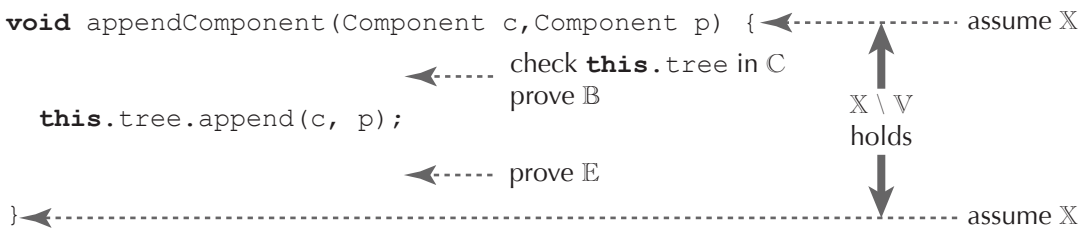

Figure 5.1: Parameters of the unified framework for visible-state verification techniques. Illustration based on [104] using element method `appendComponent()` (see Figure 3.5 on page 45).

callee in the set of instances *s* during the execution of the routine *r* on an instance of type *t*.

$\mathbb{E}_{(t,r)}$ set of invariants that must be proven to hold in the final state (i.e., at the **e**nd) of the execution of the routine *r* on an instance of type *t*.

$\mathbb{U}_{(t,r,t')}$ set of permitted receivers of type t' for an **u**pdate of their locations during the execution of the routine *r* on an instance of type *t*.

$\mathbb{D}_{(t)}$ the set of invariants that **d**epend on the locations of an instance of type *t* (and, indirectly, the set of locations on which an invariant of an instance of type *t* may depend).

$\mathbb{C}_{(t,r,t',r')}$ set of permitted receivers of type t' for an invocation (i.e., **c**all) of routine r' during the execution of the routine *r* on an instance of type *t*.

Figure 5.1 illustrates the meaning of these parameters based on the element method `appendComponent()` of the Rumer Composite pattern program in Figure 3.5 on page 45. 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 location updates and routine invocations, 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` the instance referred to by `this.tree` must be checked to be 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. Parameter $\mathbb{D}$ (not shown in Figure 5.1), lastly, constrains the effects of location updates by indicating the set of invariants that may be affected by an update.

In addition, the unified framework for visible-state verification techniques establishes *conditions* on the parameters that are sufficient to guarantee soundness of the described verification technique. This general soundness result reduces the complex task of proving soundness of a verification technique to checking a number of fairly simple conditions [104]. The unified framework for visible-state verification techniques has been elaborated in the context of object-oriented languages and focuses on object invariants.

Furthermore, it only supports the proof obligation of establishing an invariant whereas our verification technique additionally relies on the proof obligation of preserving an invariant. Thus, the general soundness result of the framework cannot be adopted for our verification technique, and we developed our own soundness proof for our verification technique. Like Summers et al. [105], we find it convenient to use the framework for illustration purposes only and describe our verification technique by instantiating the seven framework parameters. In the development of our soundness argument, however, the framework conditions proved to be very helpful. For instance, the first and second condition of the framework (page 429 in [104]), which relates the proof obligations $\mathbb{B}$ and $\mathbb{E}$, respectively, to the expected invariants $\mathbb{E}$, surfaces in the proof of parts (1) (b) and (2), respectively, of our main soundness Theorem 5.35. The third and fourth condition are only indirectly apparent. Those conditions ensure that the invariants that are invalidated by a method are in the method's vulnerable set $\mathbb{V}$. Our soundness argument relies on the concept of an invalidator to describe the set of instances whose invariants are vulnerable to the execution of a method on an instance by updating the locations of that instance. Thus, the third and fourth condition indirectly surface in part (1) (a), Case ($n+1$ is even) and ($n+1$ is odd), respectively, of our main soundness Theorem 5.35. The fifth condition of the framework has no correspondence in our soundness argument since Rumer does not support classical, object-oriented inheritance.

5.2 Proof obligations and assumptions

In this section, we elaborate our verification technique that we introduced in [75]. We define the technique in terms of the seven parameters of the unified framework for visible-state verification techniques. The parameters $\mathbb{U}$ and $\mathbb{C}$ as well as indirectly $\mathbb{D}$ capture the encapsulation properties of the programming language underlying the verification technique. For Rumer, these encapsulation properties are defined by the admissibility criteria introduced in Section 4.3 on page 78, i.e., Table 4.3, Table 4.5, and Table 4.4. In this section, we introduce the remaining parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$. To this end, we instantiate those parameters for each routine kind in Rumer. We discuss the resulting routine-specific instantiations in Section 5.2.3. Before doing so, we sketch the general pattern underlying our verification technique and introduce the differentiation between free, benign, and malign relationships.

5.2.1 General pattern

We devised our verification technique so as to leverage the stratification and encapsulation guarantees made by the Matryoshka Principle. We illustrate the general pattern underlying the instantiations of the framework parameters $\mathbb{V}$, $\mathbb{X}$, $\mathbb{E}$, and $\mathbb{B}$ for our verification technique using a synthetic Rumer program. Figure 5.2 displays a heap snapshot of this synthetic program, which comprises instances of the following type declarations: the application `App`, the relationships `A`, `B`, `C`, `D`, and `E`, and the entities `F`, `G`, and `H`. The

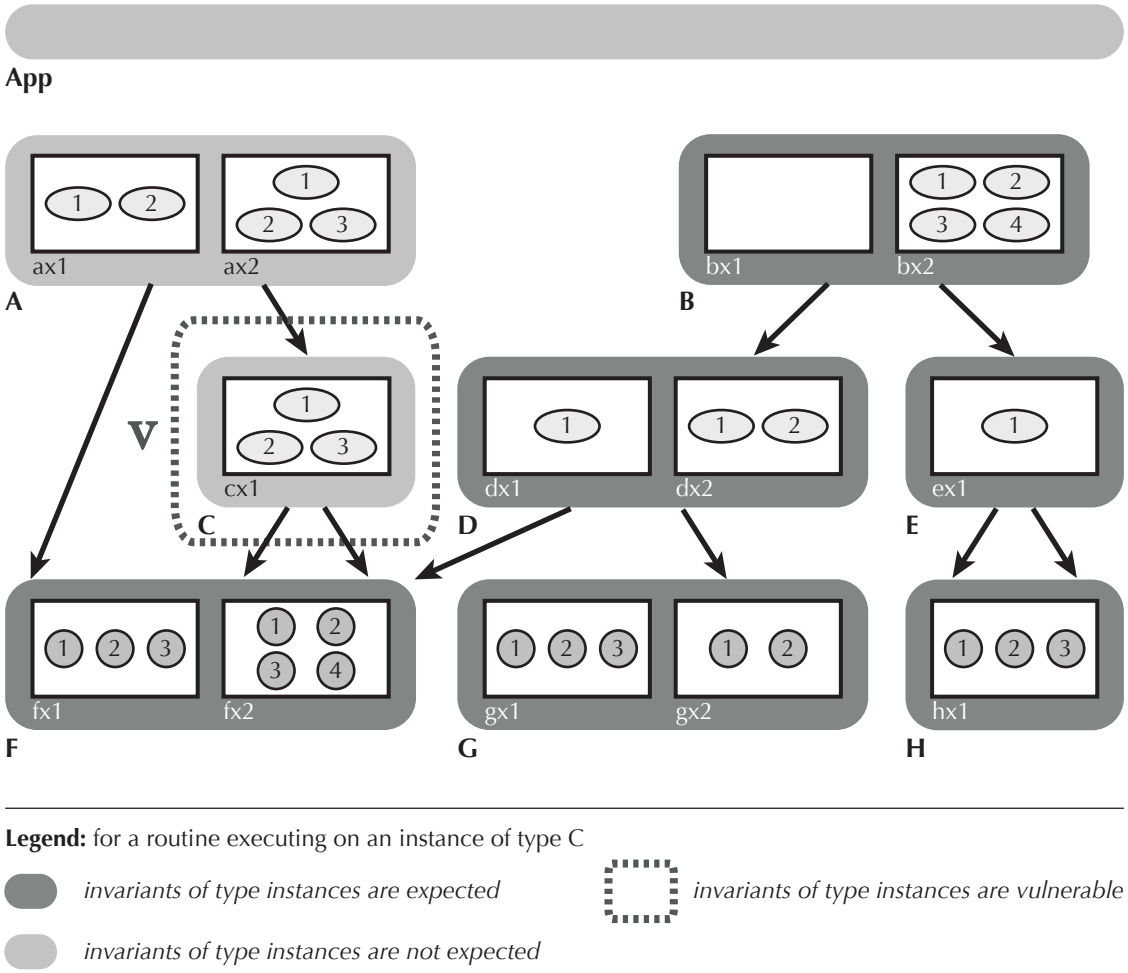

Figure 5.2: General pattern underlying vulnerable and expected invariants $\mathbb{V}$ and $\mathbb{X}$, resp., for routine execution on instance of base type C. Symbols: see legend of Figure 4.1 on page 78.

snapshot captures the execution of a routine on an instance of base type $\mathtt{C}$.

Parameter $\mathbb{V}$ relies on the guarantees made by parameters $\mathbb{U}$ ("admissible writes") and $\mathbb{D}$ ("admissible invariants"), which allow for modular reasoning about an instance's state and about the effects of state changes on invariants, respectively. As detailed by Table 4.3 on page 80, we set up parameter $\mathbb{U}$ so that a routine may only write to locations of the current receiver instance. For example, in the case of a non-interposed relationship element method, the method may only write to its relationship element receiver instance's non-interposed and interposed element fields. Furthermore, as detailed by Table 4.4 on page 82, we set up parameter $\mathbb{D}$ so that an invariant may only depend on locations of the invariant's instance. For example, in the case of a relationship extent invariant, the invariant may only depend on its relationship extent receiver instance's content or fields or on the fields of any element instances residing in its relationship extent receiver instance's content. The choice of the parameters $\mathbb{U}$ and $\mathbb{D}$ guarantee that the set of invariants $\mathbb{V}$ that are vulnerable to the execution of a routine comprises those invariants that may depend on the locations to which the routine can write. In Section 5.2.3 we detail for each routine kind what its vulnerable invariants are. To illustrate the underlying general pattern in Fig-

ure 5.2, we over-approximate the set of invariants $\mathbb{V}$ that are vulnerable to the execution of a routine on an instance of base type `C` as the set of invariants of instances of base type `C`. Figure 5.2 displays those invariants by a dotted rectangle that encloses the type whose instances' invariants are vulnerable.

Parameter $\mathbb{X}$ relies on the stratification induced by the Matryoshka Principle. We devised the parameter so that a routine can always expect the invariants of instances of those base types of which the routine-declaring base type is not a direct or transitive participant. For the routine executing on an instance of base type `C` captured in Figure 5.2, for example, these are the invariants of all instances of base types `B`, `D`, `E`, `F`, `G`, and `H`. Figure 5.2 displays the base types whose instances' invariants are expected in the visible states of a routine by dark gray boxes and, conversely, the ones that are not expect by light gray boxes. Thus, the routine in Figure 5.2 cannot expect the invariants of instances of base types `App`, `A`, and `C`. In Section 5.2.3, we detail for each routine kind what its expected invariants are. Some of those instantiations may additionally include invariants of instances of the relationship-declaring base type. As we will see, the inclusion of those invariants is mainly determined by parameters $\mathbb{C}$ ("admissible invocations") and $\mathbb{V}$.

Parameters $\mathbb{E}$ and $\mathbb{B}$, finally, rely on the parameters $\mathbb{X}$, $\mathbb{V}$, and $\mathbb{C}$. The set of invariants $\mathbb{E}$ that must be proven to hold in the final state of a routine execution is determined by the invariants $\mathbb{X}$ and $\mathbb{V}$ and comprises those invariants that are both expected in the visible states of the routine and vulnerable to the execution of the routine. The set of invariants $\mathbb{B}$ that must be proven to hold before the invocation of a routine, on the other hand, relies not only on the parameters $\mathbb{X}$ and $\mathbb{V}$ but also on parameter $\mathbb{C}$ ("admissible invocations"). As detailed by Table 4.5 on page 85, we set up parameter $\mathbb{C}$ so as to prevent transitive call-backs. Relying on this guarantee as well as on parameter $\mathbb{X}$, which guarantees that the invariants of possible callees that are of a different base type than the caller can be expected, the set of invariants $\mathbb{B}$ includes at most the invariants of instances of the same base type as the caller. For example, the set $\mathbb{B}$ must include the invariant of the caller in the case of direct call-backs.

In summary, the instantiations of the parameters $\mathbb{V}$, $\mathbb{X}$, $\mathbb{E}$, and $\mathbb{B}$ for our verification technique give rise to a scheme in which routines can expect the invariants of instances of base types of which the routine-declaring base type is not a direct or transitive participant, but cannot expect the invariants of instances of base types of which the routine-declaring base type is a direct or transitive participant. As a result, the invariants expected by the current receiver instance of a routine are left broken while sub-calls propagate downwards the participants relation, but the invariants expected by those callees are re-established by the time the sub-calls return.

As briefly mentioned in Section 5.1, our verification technique employs two kinds of proof obligations. The first kind of proof obligation is the one of *establishing* an invariant. This obligation denotes a classical proof obligation, which is also supported by the unified framework for visible-state verification techniques. It requires a routine to assert that an invariant holds in a particular state of the routine. The second kind of proof obligation

is the one of *preserving* and invariant. This obligation has been recently introduced in the context of object and class invariants in [18] and [105], respectively. It represents a weaker proof obligation as it does not require a routine to assert that an invariant holds but to show that the routine does not break the invariant under the assumption that the invariant held in the routine's initial state.

Lastly, our verification technique does not confine any built-in query operators. Built-in query operators are system-provided and have a well-defined semantics (see Section 3.2.3 on page 49). Their correct operation does not rely on existing instances satisfying their invariants. Furthermore, built-in query operators are side-effect free. Thus, our verification technique does not impose a visible-state semantics on built-in query operators.

5.2.2 Free, benign, and malign relationships

As explained previously, the set of invariants $\mathbb{V}$ that are vulnerable to the execution of a routine comprises those invariants that may depend on the locations to which the routine can write. For element methods, this set $\mathbb{V}$ comprises in general the invariant of the current receiver instance of the method as well as the invariant of the extent instance in which the current receiver instance resides. The existence of interposed relationship element fields, however, may affect the set of invariants $\mathbb{V}$ that are vulnerable to the execution of an interposed relationship element method. Since interposed relationship element fields are shared among all the relationship element instances that reside in an extent instance and that refer to the same participant instance (see Section 3.2.4 on page 51 and System Invariant 3.6 on page 71), a non-interposed relationship element method may compromise not only the invariant of its current receiver instance and the invariant of the extent instance in which the current receiver instance resides but also the invariants of all those element instances that refer to the same participant instance as the current receiver instance.

Figure 5.3 a) illustrates this issue for the declaration of a relationship R with the participant base types A and B and the role identifiers a and b, respectively. The relationship interposes the element field x into the role R.a and the element field y into the role R.b and declares a non-interposed element field z. Furthermore, the relationship declares an element invariant that depends on the fields x and z. If a non-interposed relationship element method executes, for example, on the topmost relationship element instance in Figure 5.3 a), then not only the invariant of the topmost element instance is vulnerable but also the invariant of the second topmost relationship element instance. Both relationship element instances share the topmost role element instance of the role R.a, and changes applied to the role element field x by either relationship element instance affect the invariants of both relationship element instances.

The set of vulnerable invariants $\mathbb{V}$ for a non-interposed relationship element method are only increased by the invariants of all the relationship element instances that refer to the same participant instance as the current receiver instance if the relationship declares in-

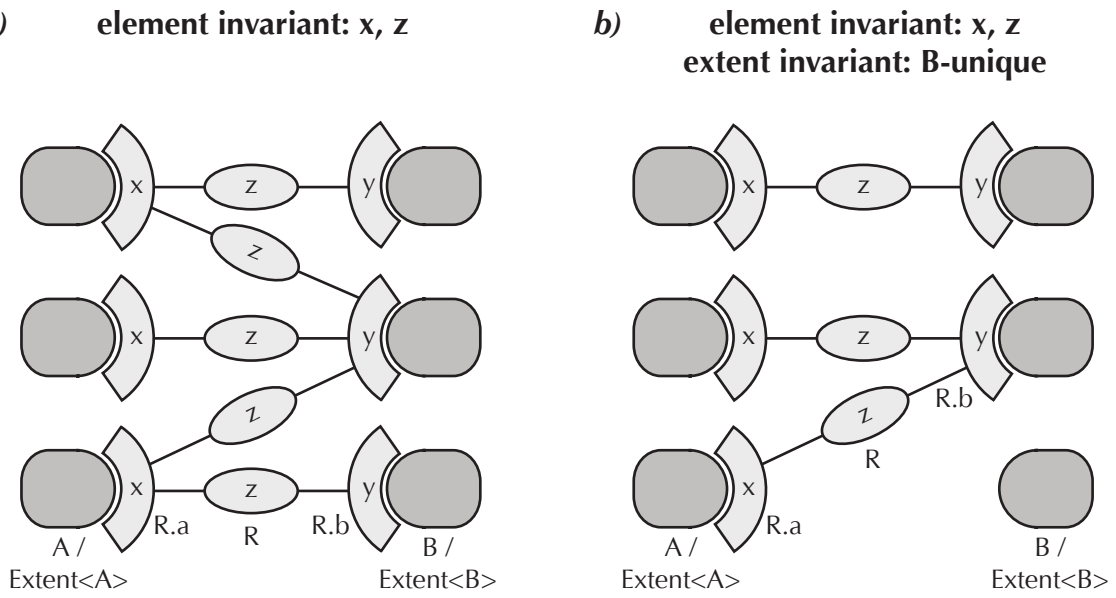

Figure 5.3: Interdependence between relationship element invariants on interposed fields and uniqueness assertions of extent invariants (see also Table 5.4): *a)* A-malign relationship, *b)* A-benign relationship. Elliptic rectangles denote relationship participant instances (i.e., entity/relationship element/extent instances), see legend of Figure 4.1 on page 78 for remaining symbols.

terposed element fields *and* imposes an element invariant on any of these fields. But even if a relationship imposes an element invariant on its interposed element fields, the set of vulnerable invariants $\mathbb{V}$ does not necessarily need to be increased. If the relationship additionally declares an extent invariant that prevents the sharing of interposed element fields among different relationship element instances, then the set $\mathbb{V}$ for a non-interposed relationship element method only comprises the invariant of the current receiver instance of the method and the invariant of the extent instance in which the current receiver instance resides. Figure 5.3 b) illustrates this scenario and augments the relationship of Figure 5.3 a) with an extent invariant that guarantees that any extent instance of type $\mathsf{Extent}\langle\mathsf{R}\rangle$ builds a right-unique relation. Given this assertion, any non-interposed relationship element method executing on an instance of the relationship of Figure 5.3 b) invalidates at most the element invariant of its current receiver instance.

The factors that finally determine the set of vulnerable invariants $\mathbb{V}$ for a non-interposed relationship element method are: the existence of an element invariant on an interposed element field and the existence of relation uniqueness assertions through extent invariants. Since interposed element fields and uniqueness assertions can both be declared for either side of the relationship, nine different relationship kinds can be distinguished. Table 5.4 provides an overview of these kinds for the same relationship as the one in Figure 5.3. The second to fifth column of Table 5.4 indicate which fields the element invariant of the relationship constrains. For example, the second column represents relationships whose element invariant constrains at most the non-interposed element field `z` and the fifth column represents relationships whose element invariant constrains at least the interposed element fields `x` and `y`. The non-interposed element field `z` is parenthe-

Kind	Element invariant				Uniqueness assertion	
	(z)	x, (z)	y, (z)	x, y, (z)	A-unique	B-unique
free	●	-	-	-	○	○
A-benign	-	●	-	-	○	●
B-benign	-	-	●	-	●	○
A-benign–B-benign	-	-	-	●	●	●
A-malign	-	●	-	-	○	-
B-malign	-	-	●	-	-	○
A-benign–B-malign	-	-	-	●	-	●
A-malign–B-benign	-	-	-	●	●	-
A-malign–B-malign	-	-	-	●	-	-

Table 5.4: Summary of different relationship kinds. Relationship R with participant base types A and B and role identifiers a and b, resp., and non-interposed element field z and interposed element fields x and y for roles R.a and R.b, resp. Symbols: '●' mandatory, '○' optional, - nonexistent.

sized to subsume the cases where the field is either constrained or not constrained by the element invariant. The sixth to last column of Table 5.4 indicate which side of the relation the extent invariant makes unique. Each row of Table 5.4 lists a particular relationship kind. The symbol '●' indicates that the respective element invariant or uniqueness assertion is mandatory, whereas the symbol '○' indicates that the respective element invariant or uniqueness assertion is optional.

As indicated by the first column of Table 5.4, we distinguish the various relationship kinds by different names. A *free* relationship is a relationship that does not impose an element invariant on its interposed element fields. This kind of relationship also subsumes any relationships that do not declare any interposed element fields. Non-interposed element methods of a free relationship invalidate at most the element invariant of their current receiver instance. A *benign* relationship is a relationship that imposes an element invariant on its interposed element field(s) and that declares an extent invariant guaranteeing that the underlying relation is unique. For example, the relationship of Figure 5.3 b) is an instance of an A-benign relationship since the element invariant constrains the field `x`, which is interposed into the role R.a, and since the extent invariant imposes a uniqueness assertion on the role R.b. Non-interposed element methods of an A-benign and/or B-benign relationship are guaranteed to invalidate at most the element invariant of their current receiver instances upon writes to the fields interposed into the roles R.a and/or R.b, respectively. We use the term *malign* for relationships whose element invariant constrains interposed element fields but whose extent invariant fails to impose uniqueness assertions on the side opposite the one targeted by member interposition. For example, the relationship of Figure 5.3 a) is an instance of an A-malign relationship since the element invariant constrains the field `x`, which is interposed into the role R.a, but the extent invariant does not impose a uniqueness assertion on the role R.b. Non-interposed element methods of an A-malign and/or B-malign relationship invalidate not only the element invariant of their current receiver instances but also the element invariants of all relationship element instances that relate to the same instances of the role R.a and/or R.b, respectively.

Parameters for: action of a application App **Current receiver:** App *app*
$\mathbb{X}$: Invariant of *app*.
For any entity or relationship S declared in App: for any instance Extent⟨Element⟨S⟩⟩ *sx*, invariant of *sx* and invariants of all instances Element⟨S⟩ *s* in *sx*.
$\mathbb{V}$: Invariant of *app*.
$\mathbb{B}$: If callee = *app*, then invariant of *app*.
$\mathbb{E}$: Invariant of *app*.

Table 5.5: Parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for application actions.

5.2.3 Parameter instantiations

The routine-specific instantiations of the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ of the unified framework for visible-state verification techniques are shown in Table 5.5, Table 5.6, Table 5.7, Table 5.8, Table 5.9, Table 5.10, Table 5.11, Table 5.12, and Table 5.13. We review each routine-specific instantiation in turn. Rumer program examples that illustrate these routine-specific instantiations are discussed in Section 7.1 on page 185.

Table 5.5 specifies the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for application actions. The parameters are specified for the execution of an application action on the singleton application receiver instance *app* of type App. Since an application invariant can only depend on application global variables, only the invariant of the current receiver instance *app* is vulnerable to the action execution. According to Figure 5.2, an application action expects the invariants of instances of those base types in its visible states of which App is not a direct or transitive participant. Since an application is the topmost abstraction of a Rumer program (see Section 4.2 on page 76), an application action expects the invariants of all element and extent instances of a program's entity and relationship declarations. Thanks to the proof obligations $\mathbb{E}$ and $\mathbb{B}$, an application action can also expect the invariant of its current receiver instance *app* to hold in the action's visible states. The proof obligation $\mathbb{B}$ to establish the invariant of the current receiver instance *app* before a direct call-back is vital to soundly support data type induction.

Table 5.6 specifies the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for default entity extent constructors and default relationship extent constructors (see Section 3.4.1 on page 70 as well as Table 3.19 and Table 3.20, resp.). The parameters are specified for a default extent constructor of an entity or relationship T that executes on a receiver instance *tx* of type Extent⟨Element⟨T⟩⟩. As indicated by Table 5.6, the invariant of the extent instance *tx* is vulnerable to the execution of the default extent constructor. This invariant is the only invariant which is vulnerable to the constructor's execution. In particular, no invariants of element instances of type Element⟨T⟩ are affected since the current extent receiver instance *tx* is empty. As implied by Figure 5.2, the default extent constructor expects in its visible states the invariants of instances of any base type S of which T is not a direct or transitive participant. In addition, the default extent constructor expects the invariants of all other extent

Parameters for: default extent constructor of an entity or relationship T **Current receiver:** Extent⟨Element⟨T⟩⟩ *tx*	
$\mathbb{X}$:	For any instance Extent⟨Element⟨T⟩⟩ *tx′* such that *tx* ≠ *tx′*: invariant of *tx′* and invariants of all instances Element⟨T⟩ *t′* in *tx′*.
	For any entity or relationship S such that S ≠ T and T is not a direct or transitive participant of S: for any instance Extent⟨Element⟨S⟩⟩ *sx*, invariant of *sx* and invariants of all instances Element⟨S⟩ *s* in *sx*.
$\mathbb{V}$:	Invariant of *tx*.
$\mathbb{B}$:	*n/a*
$\mathbb{E}$:	Invariant of *tx*.

Table 5.6: Parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for default entity and relationship extent constructors.

instances *tx′* of type Extent⟨Element⟨T⟩⟩, including the invariants of element instances *t′* residing in such extent instances *tx′*. However, the default extent constructor cannot expect the invariant of its current receiver instance *tx* since this instance has been newly instantiated and may thus not meet its invariant. The reason why the invariants of other extent instances *tx′* and of their element instances can be expected is due to parameter $\mathbb{C}$ (see Table 4.5 on page 85), which prevents routine invocations to propagate horizontally or upwards. For example, for the creation of instance `bx1` in Figure 5.2, the application `App` invokes the default relationship extent constructor `new Extent<B>()`, which executes on the relationship extent instance under construction `bx1`. During its execution, the constructor can expect the invariants of all extent instances and element instances of the types `A`, `C`, `D`, `E`, `F`, `G`, and `H` as well as the invariant of the extent instance `bx2` including the invariants of its element instances `bx2-1`, `bx-2`, `bx2-3`, and `bx2-4`. The proof obligation $\mathbb{E}$, finally, guarantees that the newly created extent instance *tx* satisfies its invariant once the constructor completes. Since default extent constructors do not invoke any routines, the proof obligation $\mathbb{B}$ is vacuous.

Table 5.7 specifies the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for programmer-defined entity extent constructors and programmer-defined relationship extent constructors. The parameters are specified for an extent constructor of an entity or relationship T that executes on a receiver instance *tx* of type Extent⟨Element⟨T⟩⟩. Programmer-defined extent constructors only marginally differ in the parameters $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ from default extent constructors. Since programmer-defined extent constructors may include statements to populate the current extent receiver instance *tx* and to modify any of the newly instantiated element instances, the invariants of any element instances *t* residing in the extent instance *tx* are vulnerable too (parameter $\mathbb{V}$). Thus, the invariant of the current extent receiver instance *tx* as well as the invariants of its element instances *t* are the only invariants that are vulnerable to the constructor's execution. In particular, no invariants of other extent instances *tx′* of type Extent⟨Element⟨T⟩⟩ are affected by any modifications of the element instances *t* since those element instances are guaranteed to reside in the current extent receiver instance *tx* only (see System Invariant 3.6 on page 71). The proof obligation $\mathbb{B}$ guards against direct call-backs and requires the invariant of the current extent receiver instance

Parameters for: extent constructor of an entity or relationship T **Current receiver:** Extent⟨Element⟨T⟩⟩ *tx*	
$\mathbb{X}$:	For any instance Extent⟨Element⟨T⟩⟩ *tx*′ such that *tx* ≠ *tx*′: invariant of *tx*′ and invariants of all instances Element⟨T⟩ *t*′ in *tx*′.
	For any entity or relationship S such that S ≠ T and T is not a direct or transitive participant of S: for any instance Extent⟨Element⟨S⟩⟩ *sx*, invariant of *sx* and invariants of all instances Element⟨S⟩ *s* in *sx*.
$\mathbb{V}$:	Invariant of *tx* and invariants of all instances Element⟨T⟩ *t* in *tx*.
$\mathbb{B}$:	If callee = *tx*, then invariant of *tx* and invariants of all instances Element⟨T⟩ *t* in *tx*.
	If callee = instance Element⟨T⟩ *t* in *tx*, then invariant of *t*.
	If callee = instance Element⟨T⟩ *t* in *tx* and T = relationship with participant base types A and B that is A-benign and/or B-benign, then B-uniqueness and/or A-uniqueness assertion of invariant of *tx*, resp.
$\mathbb{E}$:	Invariant of *tx* and invariants of all instances Element⟨T⟩ *t* in *tx*.

Table 5.7: Parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for programmer-defined entity and relationship extent constructors.

tx and the invariants of its element instances *t* to be established. Furthermore, the proof obligation makes sure that the B-uniqueness and/or A-uniqueness assertions of *tx*'s extent invariant are established, should the callee be an element instance *t* that resides in *tx* and should T be an A-benign and/or B-benign relationship (see Table 5.4). The proof obligation $\mathbb{E}$, finally, guarantees that the newly created extent instance *tx* and its element instances *t* satisfy their invariants once the constructor completes.

Table 5.8 specifies the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for entity extent methods and relationship extent methods. The parameters are specified for an extent method of an entity or relationship T that executes on a receiver instance *tx* of type Extent⟨Element⟨T⟩⟩. Extent methods and programmer-defined extent constructors only differ in the parameter $\mathbb{X}$. As opposed to an extent constructor, an extent method additionally expects the invariant of its current receiver instance *tx* as well as the invariants of the element instances *t* residing in *tx* (parameter $\mathbb{X}$). This assumption relies on parameter $\mathbb{C}$ but also on the proof obligations of extent constructors, which guarantee that newly created extent instances satisfy their invariants.

Table 5.9 specifies the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for element methods of entities and free relationships (see Table 5.4). The parameters are specified for an element method of an entity or free relationship T that executes on a receiver instance *t* of type Element⟨T⟩. As expected, the invariant of the current element receiver instance *t* as well as the invariant of the extent instance *tx* in which *t* resides are vulnerable to the execution of the element method (parameter $\mathbb{V}$). The invariant of the current element receiver instance *t* is also expected in the visible states of the element method (parameter $\mathbb{X}$). In addition, an element method expects in its visible states the invariants of instances of any base type S of which T is not a direct or transitive participant. As opposed to an extent method, an element method cannot expect the invariant of any instance of type Extent⟨Element⟨T⟩⟩ (including the extent instance *tx* in which the current receiver instance *t* resides) nor of

Parameters for: extent method of an entity or relationship T **Current receiver:** Extent⟨Element⟨T⟩⟩ *tx*	
$\mathbb{X}$:	For any instance Extent⟨Element⟨T⟩⟩ *tx*′: invariant of *tx*′ and invariants of all instances Element⟨T⟩ *t*′ in *tx*′.
	For any entity or relationship S such that S ≠ T and T is not a direct or transitive participant of S: for any instance Extent⟨Element⟨S⟩⟩ *sx*, invariant of *sx* and invariants of all instances Element⟨S⟩ *s* in *sx*.
$\mathbb{V}$:	Invariant of *tx* and invariants of all instances Element⟨T⟩ *t* in *tx*.
$\mathbb{B}$:	If callee = *tx*, then invariant of *tx* and invariants of all instances Element⟨T⟩ *t* in *tx*.
	If callee = instance Element⟨T⟩ *t* in *tx*, then invariant of *t*.
	If callee = instance Element⟨T⟩ *t* in *tx* and T = relationship with participant base types A and B that is A-benign and/or B-benign, then B-uniqueness and/or A-uniqueness assertion of invariant of *tx*, resp.
$\mathbb{E}$:	Invariant of *tx* and invariants of all instances Element⟨T⟩ *t* in *tx*.

Table 5.8: Parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for entity and relationship extent methods.

any other element instances of type Element⟨T⟩. This specificity is due to parameters $\mathbb{U}$, $\mathbb{C}$, and $\mathbb{B}$ of extent methods, which allow a current extent receiver instance to leave its own invariant as well as the invariants of its element instances broken for the duration of routine invocations on element instances of the same base type as the extent instance but residing in a different extent instance. For example, an extent method executing on the extent instance `fx1` in Figure 5.2 may assign to the fields of its element instances `fx1-1` and `fx1-3` and then invoke an element method on the element instance `fx2-4`. Since the callee element instance `fx2-4` resides in the extent instance `fx2` and not in the caller extent instance `fx1`, neither the invariant of the extent instance `fx1` nor the invariants of the element instances `fx1-1` and `fx1-3` have to be re-established before the invocation of the element method. As a result, the element method cannot expect those invariants to hold. In its final state (parameter $\mathbb{E}$), an element method must establish the invariant of its current element receiver instance *t*. In addition, the element method must show to preserve the invariant of the extent instance *tx* in which *t* resides between the initial and final state of the method. This proof obligation accounts for the fact that an element method may break an extent invariant but may not be in the position to re-establish that invariant. If an element method cannot show to preserve an extent invariant, its corresponding code must be moved to an extent method. The proof obligation $\mathbb{B}$, finally, guards the current receiver instance *t* against direct call-backs.

Table 5.10 and Table 5.11 specify the parameter $\mathbb{X}$ and the parameters $\mathbb{V}$ and $\mathbb{E}$ for non-interposed element methods of benign and malign relationships, respectively. The parameters are specified for an non-interposed element method of relationship R with the participant base types A and B and the role identifiers a and b, respectively, where the method executes on a receiver instance *r* of type Element⟨R⟩. Since the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for element methods of benign or malign relationships are largely congruent with the ones for element methods of free relationships, Table 5.10 and Table 5.11 only list the parameters that are affected by a relationship's benignity and malignity, respec-

Parameters for: element method of an entity or free relationship T **Current receiver:** Element⟨T⟩ *t*
$\mathbb{X}$: Invariant of *t*.
For any entity or relationship S such that S ≠ T and T is not a direct or transitive participant of S: for any instance Extent⟨Element⟨S⟩⟩ *sx*, invariant of *sx* and invariants of all instances Element⟨S⟩ *s* in *sx*.
$\mathbb{V}$: Invariant of *t*.
Invariant of instance Extent⟨Element⟨T⟩⟩ *tx* in which *t* resides.
$\mathbb{B}$: If callee = *t*, then invariant of *t*.
$\mathbb{E}$: Invariant of *t*.
Preservation of invariant of instance Extent⟨Element⟨T⟩⟩ *tx* in which *t* resides.

Table 5.9: Parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for entity and free, non-interposed relationship element methods.

Adjunct parameter for: element method of an A-benign relationship R **Participants:** base types A and B and role identifiers a and b, resp. **Current receiver:** Element⟨R⟩ *r*
$\mathbb{X}^+$: B-uniqueness assertion of invariant of instance Extent⟨Element⟨R⟩⟩ *rx* in which *r* resides.

Table 5.10: Adjunct for parameter $\mathbb{X}$ for unilaterally benign, non-interposed relationship element methods. This parameter conjoins with the parameters listed in Table 5.9 and, if the relationship is inversely benign or malign with the parameters listed in Table 5.10 or Table 5.11, resp.

tively. Furthermore, to accommodate the fact that a relationship's benignity or malignity is dependent on a relationship's side, Table 5.10 and Table 5.11 only specify the parameter values from the point of view of one side of the relationship. Thus, to determine the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for an non-interposed element method of a relationship, the parameters in Table 5.9 must be conjoined, for each side of the relationship that is benign or malign, with the parameters in Table 5.10 or Table 5.10, respectively. For example, the parameters for the A-benign relationship of Figure 5.3 b) are determined by conjoining the parameters in Table 5.9 with the parameters in Table 5.10. As discussed in Section 5.2.2, benign relationships rely on the underlying relation's uniqueness, and, thus, the adjunct parameter $\mathbb{X}^+$ in Table 5.10 includes the side-specific uniqueness assertion of the invariant of the extent instance *rx* in which the current receiver instance *r* resides. Malign relationships, on the other hand, may invalidate the invariants of all relationship element instances r' that relate to the same role element instance as the current relationship element receiver instance *r*.[1] Thus, the adjunct parameters $\mathbb{V}^+$ and $\mathbb{E}^+$ in Table 5.11 include the invariants of those instances r' and require the non-interposed relationship element method to preserve the invariants of those instances r' between the initial and final state of the method, respectively.

Table 5.12 specifies the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for interposed relationship element

[1] According to System Invariant 3.6 on page 71, the relationship element instances r' are guaranteed to reside in the same extent instance *rx* as the relationship element instance *r*.

Adjunct parameter for: element method of a B-malign relationship R **Participants:** base types A and B and role identifiers a and b, resp. **Current receiver:** Element⟨R⟩ *r*
$\mathbb{V}^+$: Invariants of all instances Element⟨R⟩ $r' \neq r$ such that $r'.b = r.b$.
$\mathbb{E}^+$: Preservation of invariants of all instances Element⟨R⟩ $r' \neq r$ such that $r'.b = r.b$.

Table 5.11: Adjuncts for parameters $\mathbb{V}$ and $\mathbb{E}$ for unilaterally malign, non-interposed relationship element methods. These parameters conjoin with the parameters listed in Table 5.9 and, if the relationship is inversely benign or malign with the parameters listed in Table 5.10 or Table 5.11, resp.

Parameters for: element method of a role R.a declared by relationship R **Current receiver:** Element⟨R.a⟩ *ra*	
$\mathbb{X}$:	For any entity or relationship S such that S ≠ R and R is not a direct or transitive participant of S: for any instance Extent⟨Element⟨S⟩⟩ *sx*, invariant of *sx* and invariants of all instances Element⟨S⟩ *s* in *sx*.
$\mathbb{V}$:	Invariants of all instances Element⟨R⟩ *r* such that *r.a* = *ra*.
	Invariant of instance Extent⟨Element⟨R⟩⟩ *rx* in which all instances Element⟨R⟩ *r* with *r.a* = *ra* reside.
$\mathbb{B}$:	–
$\mathbb{E}$:	Preservation of invariants of all instances Element⟨R⟩ *r* such that *r.a* = *ra*.
	Preservation of invariant of instance Extent⟨Element⟨R⟩⟩ *rx* in which all instances Element⟨R⟩ *r* with *r.a* = *ra* reside.

Table 5.12: Parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for interposed relationship element methods.

methods. The parameters are specified for an element method of role R.a of relationship R that executes on a receiver instance *ra* of type Element⟨R.a⟩. Those parameters are similar to the ones of non-interposed element methods of malign relationships with the difference that no invariants can be imposed on role element instances. As a result, an interposed relationship element method only expects in its visible states the invariants of instances of any base type S of which R is not a direct or transitive participant. The invariants vulnerable to the execution of the method are the invariants of the relationship element instances *r* that relate to the current role element receiver instance *ra* as well as the invariant of the relationship extent instance *rx* in which those instances *r* reside.[2] The invariants of those relationship element instances *r* and the invariant of the relationship extent instance *rx* must be shown to be preserved between the initial and final state of the method. The proof obligation $\mathbb{B}$ is vacuous since no invariants for role element instances exist, which would need to be guarded against direct call-backs.

Table 5.13 specifies the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for constructors and programmer-defined initializers of entity molds and relationship molds. The parameters are specified for a mold constructor or mold initializer of entity or relationship T that executes on a receiver instance *t* of type Mold⟨T⟩. Those parameters differ importantly from the pa-

[2] System Invariant 3.6 on page 71 guarantees that the relationship element instances *r* reside all in the same relationship extent instance *rx*.

Parameters for: mold constructor or initializer of an entity or relationship T **Current receiver:** Mold⟨T⟩ *t*
$\mathbb{X}$: None.
$\mathbb{V}$: None.
$\mathbb{B}$: *n/a*
$\mathbb{E}$: None.

Table 5.13: Parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for entity and non-interposed relationship mold constructors and programmer-defined initializers.

rameters of the routines introduced earlier. As indicated by Table 5.13, mold constructors and initializers neither expect any invariants to hold in their visible states (parameter $\mathbb{X}$), nor violate any invariants during their execution (parameter $\mathbb{V}$), nor require to prove certain invariants to hold in their final state (parameter $\mathbb{E}$). The proof obligation $\mathbb{B}$ is vacuous since mold constructors and initializers are not allowed to invoke any routines (see Table 4.5 on page 85). The choice of the parameters are due to two reasons: *(i)* that mold constructors and mold initializers can only write to fields of mold instances (see parameter $\mathbb{U}$ in Table 4.3 on page 80) and *(ii)* that no invariants can depend on fields of mold instances (see parameter $\mathbb{D}$ in Table 4.4 on page 82). As a result, invariants are neither imposed on mold instances nor violated by the executions of mold constructors and initializers. The instantiations of the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ in Table 5.13 effectively mean that mold constructors and mold initializers do not obey a visible-state semantics.

The proof obligations $\mathbb{B}$ and $\mathbb{E}$ of the verification technique introduced in this section are all relative to the type of a routine's current receiver instance. As a result, types can be verified independently from each, facilitating the *modular* verification of a Rumer program.

5.3 Soundness

To prove our verification technique sound, we must show that the proof obligations $\mathbb{B}$ and $\mathbb{E}$ imposed on the routines of a Rumer program are sufficient to allow those routines to assume the expected invariants $\mathbb{X}$ in their visible states. Before stating the main soundness Theorem 5.35, we introduce auxiliary definitions, propositions, lemmas, and corollaries on which the soundness theorem and its proof rely.

The Matryoshka Principle gives rise to an important lemma that allows us to relate the call stack of an executing Rumer program to the set of expected invariants of an executing routine. Intuitively, this lemma is expressed by Figure 5.2. We first introduce the notion of a *call stack* and its *well-formedness* condition and then provide the lemma.

Definition 5.1 (Call stack): A call stack is a sequence of stack frames. A stack frame corresponds to the invocation of a routine *r* on an instance *o* and is represented by the pair

(o,r), consisting of the current receiver instance o and the routine name r. A call stack is formally defined as:

$$\sigma = \begin{cases} \epsilon & \text{if empty stack,} \\ \sigma \circ (o, r) & \text{otherwise } (\circ \text{ denotes push operator}). \end{cases}$$

Occasionally, we find it convenient to subscript a stack frame with its index in the call stack, such that the stack frames $(o', r')_l$ and $(o, r)_{l-1}$ denote a callee stack frame and its caller stack frame, respectively.

The well-formedness condition on a call stack relies on the notion of a base type (see Definition 3.1 on page 62) and guarantees that the receiver instances' base types of any consecutive stack frames conform to the reflexive closure of a program's participants relation. Furthermore, it guarantees, for routine invocation across stratification layers, that there exists no receiver instance of the same base type as the callee's base type on the call stack. The well-formedness condition does not apply to mold instances since parameter $\mathbb{C}$ (see Table 4.5 on page 85) leaves the propagation of invocations of mold constructors and initializers unconstrained.

Definition 5.2 (Well-formed call stack): A call stack σ is well-formed if and only if for any consecutive stack frames $(o', r')_l$ and $(o, r)_{l-1}$ contained in σ, where $(o', r')_l$ denotes the callee stack frame and $(o, r)_{l-1}$ denotes the caller stack frame, if the callee o' is not a mold instance, then it either holds that $\mathit{BaseType}(o') = \mathit{BaseType}(o)$ or that $\mathit{BaseType}(o')$ is a direct participant of $\mathit{BaseType}(o)$, according to the participants relation of the program. In the latter case, it holds further that there exists no preceding stack frame $(o'', r'')_k$ such that $0 \leqslant k < l - 1$ and that $\mathit{BaseType}(o'') = \mathit{BaseType}(o')$.

The following proposition guarantees that the execution of a Rumer program only produces well-formed call-stacks:

Proposition 5.3 (Rumer call stacks are well-formed): Any call stack σ generated by the execution of a successfully type checked and verified Rumer program is well-formed.

Proof 5.4: The proof of Proposition 5.3 proceeds by induction over the execution of a Rumer program where the interesting cases are those cases where stack frames are added to and removed from σ. Those cases depend on parameter $\mathbb{C}$ (see Table 4.5 on page 85) and on the acyclicity of a program's participants relation (see System Invariant 4.1 on page 77). Parameter $\mathbb{C}$ guarantees that a callee's base type is either the same as the caller's base type or a direct participant of the caller's base type. A program's participants relation's acyclicity guarantees that a type cannot be its own direct or transitive participant and hence, for routine invocations across stratification layers, that there exists no previous stack frame with a receiver instance of the same base type as the callee's base type. □

We can finally relate the set of expected invariants of a callee to the set of expected invariants of its caller:

Lemma 5.5 (Call stack and expected invariants): For any well-formed call stack σ and for any consecutive stack frames $(o', r')_l$ and $(o, r)_{l-1}$ contained in σ, where $(o', r')_l$ denotes the callee stack frame and $(o, r)_{l-1}$ denotes the caller stack frame, if $\mathit{BaseType}(o') \neq \mathit{BaseType}(o)$, then $\mathbb{X}_{(o',r')} \subseteq \mathbb{X}_{(o,r)} \setminus \{\mathit{inv}_{o''} \mid \mathit{BaseType}(o'') = \mathit{BaseType}(o)\}$.

Proof 5.6: We prove Lemma 5.5 by considering the following two cases:

(o' is not a mold instance): Then, Lemma 5.5 follows directly from the routine-specific instantiations of framework parameter $\mathbb{X}$ (see Table 5.5, Table 5.6, Table 5.7, Table 5.8, Table 5.9, Table 5.10, Table 5.11, and Table 5.12) and from Proposition 5.3 and Definition 5.2. Depending on the routine, parameter $\mathbb{X}$ subsumes, at minimum, the invariants of all element and extent instances of a base type of which the current receiver's base type is not a direct or transitive participant and, at maximum, these invariants plus the invariants of all element and extent instances of the current receiver's base type. By Proposition 5.3 and Definition 5.2, we know, for any routine invocation that crosses stratification layers, that the callee's base type must be a (transitive) participant of (any of) its (transitive) caller(s)'s base type(s) and that the callee's base type is different from the base type(s) of (any of) its (transitive) caller(s). Since a base type can be a direct participant of several base types, the set of base types of which a callee's base type is a transitive participant must be a superset of the set of base types of which its caller's base type is a transitive participant. Thus, the maximal set of invariants expected by an invoked routine must be a subset of the set of invariants expected by its calling routine minus the invariants of all element and extent instances of the calling routine's receiver's base type.

(o' is a mold instance): Then, r' must be a mold constructor or initializer. According to Table 5.13, $\mathbb{X}_{(o',r')}$ is the empty set, and, thus, Lemma 5.5 holds trivially. □

Lemma 5.5 indicates for routine invocations that cross stratification layers that the callee's expected invariants are a subset of the caller's expected invariants. We make use of this subset relationship in the soundness proof as it allows us to conclude that the expected invariants of the callee hold if we can show that the ones of the caller hold. Lemma 5.5 does not hold for routine invocations that remain in the same stratification layer since, in some of these cases, the set of expected invariants of a callee may actually be bigger than the set of expected invariants of its caller. This is the case, for example, if a programmer-defined extent constructor invokes an extent method on its current extent receiver instance. As opposed to the extent constructor, the extent method expects the invariant of the extent instance under construction to hold in the extent method's visible states.

The main soundness Theorem 5.35 relies on the notion of *semi-visible states* of a routine execution that was introduced by Summers et al. [105]. The semi-visible states of a routine execution capture significant "mid-points" of the execution that are important for reasoning about the soundness of the verification technique in the presence of routine invocations.

Definition 5.7 (Semi-visible states): For any routine execution r on an instance o, the semi-visible states of r's execution are the following states:

- the initial state of r's execution
- the pre-states and post-states of direct routine invocations made during r's execution
- the final state of r's execution.

If the semi-visible states are numbered from zero, starting at the initial state, then routine invocations are made between any odd semi-visible state (except for the final state) and its subsequent even semi-visible state. Between any even semi-visible state and its subsequent odd semi-visible state no routines are invoked and control remains within the receiver o.

To show that the proof obligations of a routine are sufficient to allow the routine to assume the expected invariants $\mathbb{X}$ in its visible states, the main soundness Theorem 5.35 reasons about the invariants that may potentially be lost between the semi-visible states of the routine execution. Before defining the notion of a lost invariant, we define the notions of a fresh instance and generating routine and introduce the concept of an invalidator. Furthermore, we provide the definitions of the two kinds of proof obligations our verification technique employs.

A *fresh instance* is defined as follows:

Definition 5.8 (Fresh instance): An instance o is fresh in a semi-visible state s_i if o is the current receiver instance of a (possibly default) extent constructor or mold constructor r and s_i is r's initial state.

A *generating routine* is a routine that instantiates new element or extent instances and is defined as follows:

Definition 5.9 (Generating routine): A routine r that executes on an instance o is a generating routine in any of the following cases:

- If r is a (possibly default) extent constructor. Then, r's current extent receiver instance o is said to be generated by r. The generated instance o is fresh in r's initial state $s_{initial}$ and exists in state $s_{initial+1}$.
- If r is an extent constructor or extent method that includes invocations of the built-in addition operator (see Table 3.17 on page 68 and Table 3.18 on page 69). Then, the element instances that are instantiated by those invocations are said to be generated by r. Any element instance generated by r in this way does not exist in the semi-visible state s_i that precedes the invocation of the built-in addition operator, but exists in state s_{i+1}.

Next, we define the proof obligations of *establishing* an invariant and *preserving* an invariant which our verification technique employs:

Definition 5.10 (Establishing an invariant): For a routine r that executes on an instance

o, the proof obligation to establish the invariant $inv_{o'}$ of an instance o' in the semi-visible state s_i of r requires to show that the invariant $inv_{o'}$ holds in s_i.

Definition 5.11 (Preserving an invariant): For a routine r that executes on an instance o, the proof obligation to preserve the invariant $inv_{o'}$ of an instance o' between the semi-visible states s_i and s_j (where $i \leqslant j$) of r requires to show that if the instance o' exists but is not fresh in s_i and its invariant $inv_{o'}$ holds in s_i, then the invariant $inv_{o'}$ holds in s_j .

The following proposition points out that generated element instances always reside in the current extent receiver instance of their generating routine:

Proposition 5.12 (Containment of generated element instances): Any element instance generated by a generating routine r resides in the current receiver instance of r or, if the generated instance is a role element instance, is related by relationship element instances that reside in the current receiver instance of r.

Proof 5.13: According to Definition 5.9, r must be an extent constructor or extent method, and the element instances are generated by invocations of the built-in addition operator during r's execution. We know by Table 4.3 on page 80 that invocations of the built-in addition operator (see Table 3.17 on page 68 and Table 3.18 on page 69) are only allowed to be invoked on the current receiver instance of the invoking routine. As a result, the newly instantiated entity and relationship element instances reside in r's current receiver instance and, if r's current receiver instance is a relationship extent instance, the newly instantiated role element instances are related by relationship element instances that reside in r's current receiver instance. □

The following definition introduces the concept of an *invalidator*. This concept allows us to describe the framework parameters $\mathbb{U}$ and $\mathbb{D}$ from the perspective of a specific instance.

Definition 5.14 (Invalidator): For any instance o, its invalidators are those instances that can write to locations on which the invariant of o depends.

For the verification technique presented in this chapter, the invalidators are as follows:

Proposition 5.15 (Set of Invalidators): For the possible instances in a successfully type checked and verified Rumer program the invalidators for such an instance o are:

- For an application instance o: the application instance o.
- For a relationship extent instance o: the relationship extent instance o, all relationship element instances residing in o, and all role element instances related by the relationship element instances residing in o.
- For a relationship element instance o that is neither A-malign nor B-malign (first four rows in Table 5.4): the relationship element instance o, o's role element instances, and the relationship extent instance in which o resides.
- For a relationship element instance o of a relationship R with participant base types A and B and role identifiers a and b, respectively, that is A-malign and/or B-malign (last

five rows in Table 5.4): the relationship element instance o, o's role element instances, all relationship element instances that reside in the relationship extent instance of o and that have the role element instance $o.a$ and/or $o.b$, respectively, as participant, and the relationship extent instance in which o resides.
- For a role element instance o: *n/a*.
- For an entity extent instance o: the entity extent instance o and all entity element instances residing in o.
- For an entity element instance o: the entity element instance o and the entity extent instance in which o resides.

Proof 5.16: An instance's invalidators follow directly from the parameters $\mathbb{U}$ (see Table 4.3 on page 80) and $\mathbb{D}$ (see Table 4.4 on page 82), from the fact that extent instances are mutually disjoint (see System Invariant 3.6 on page 71), and from the differentiation between free, benign, and malign relationships (see Table 5.4). Parameters $\mathbb{U}$ and $\mathbb{D}$ delimit, relative to a current receiver instance, the set of invariants that may depend on the locations to which the routine can write. System Invariant 3.6 and Table 5.4 provide further guarantees on heap separation and thus diminish the set of possibly vulnerable invariants. System Invariant 3.6 guarantees that an element instance cannot reside in several different extent instances and, thus, that modifications of an element instance by the extent instance in which the element instance resides compromise at most the extent invariant of that same extent instance. Table 5.4 guarantees for free and benign relationships that their element instances do not share interposed fields and, thus, that modifications of an element instance compromise at most the element invariant of that same element instance. □

The following corollary follows from Proposition 5.15 and makes explicit that mold instances are not invalidators for any other instances:

Corollary 5.17 (Molds are not invalidators): There exist no instances for whose invariants a mold instance can be an invalidator.

Proof 5.18: Corollary 5.17 is immediate from Proposition 5.15. □

We can finally introduce the notion of a *lost invariant*:

Definition 5.19 (Lost invariants): For any routine execution r on an instance o and any two semi-visible states s_i and s_j (where $i \leqslant j$) of r, an invariant $inv_{o'}$ of an instance o' is lost between s_i and s_j in either of the following ways:

- By invalidation: $inv_{o'}$ held in s_i, but no longer holds in s_j.
- By instantiation: o' was either fresh in s_i or did not exist in s_i, but exists in s_j, and $inv_{o'}$ does not hold in s_j.

Some routines in Rumer impose the proof obligation to show preservation of certain invariants (see Table 5.9, Table 5.11, and Table 5.12). The following lemma relates the

notion of preserving an invariant to the one of losing an invariant. It shows in particular that invariants can still be lost between two semi-visible states even though they are preserved between those states.

Lemma 5.20 (Invariant preservation and lost invariants): For any routine execution r on an instance o and any two semi-visible states s_i and s_j (where $i \leqslant j$) of r, if an invariant $inv_{o'}$ of an instance o' is preserved between s_i and s_j, then it can only be lost between s_i and s_j if the instance o' was either fresh in s_i or did not exist in s_i.

Proof 5.21: According to Definition 5.19, an invariant $inv_{o'}$ of an instance o' can be lost between two semi-visible states s_i and s_j in one of the following two ways: *(i)* o' exists in both states and satisfies its invariant in s_i but not in s_j or *(ii)* o' only exists in s_j and does not satisfy its invariant in s_j. In the first case *(i)*, the invariant $inv_{o'}$ cannot be lost between s_i and s_j since it holds in s_i and since its preservation (see Definition 5.11) guarantees that it also holds in s_j. In the second case *(ii)*, however, the invariant $inv_{o'}$ may indeed be lost between s_i and s_j. If the instance o' does not exist in s_i or is fresh in s_i, then the preservation of $inv_{o'}$ (see Definition 5.11) between s_i and s_j is too weak a guarantee to actually assert that $inv_{o'}$ also holds in s_j. □

The following lemma provides an upper bound on the invariants lost between two consecutive semi-visible states between which control remains within a routine. It combines Definition 5.14, Proposition 5.15, and Definition 5.19:

Lemma 5.22 (Invalidators and lost invariants): For any routine execution r on an instance o, consider any consecutive semi-visible states s_i and s_{i+1} of r's execution between which no routine is invoked (i.e., s_i is even and thus control remains within r). Then, at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_i and s_{i+1}.

Proof 5.23: According to Definition 5.19, invariants can only be lost between s_i and s_{i+1} either by invalidation or instantiation. We consider each case separately:

(invalidation): Since Lemma 5.22 only considers consecutive semi-visible states s_i and s_{i+1} of r's execution between which control remains within the current receiver instance o, an invariant of an instance o' can only be invalidated if o writes to any of the locations of o' between s_i and s_{i+1}. According to Definition 5.14, o is an invalidator for such an instance o'.

(instantiation): Since Lemma 5.22 only considers consecutive semi-visible states s_i and s_{i+1} of r's execution between which control remains within the current receiver instance o, an invariant can only be lost due to instantiation if r is a generating routine and if the invariant's instance is generated by the routine. According to Definition 5.9, r must either be *(i)* a (possibly default) extent constructor with the initial state s_i or *(ii)* an extent constructor or extent method that includes invocations of the built-in addition operators in-between the semi-visible states s_i and s_{i+1}. The possibly lost

invariants due to instantiation are: for the first case *(i)*, the invariant of r's current receiver instance, and, for the second case *(ii)*, the invariants of all those entity and relationship element instances that are instantiated through invocations of the built-in addition operators between s_i and s_{i+1}. In the first case *(i)*, we know by Proposition 5.15 that r's current receiver instance is also its own invalidator. In the second case *(ii)*, we know by Proposition 5.12 that the newly instantiated entity and relationship element instances reside in r's current receiver instance, which is also an invalidator for those instances according to Proposition 5.15. In the second case *(ii)*, the instantiation of new element instances may also lead to losing invariants due to invalidation. In the case of relationship element instantiation, the invariants of all those relationship element instances that reside in the current relationship extent receiver instance o and that share any role element instances with the newly instantiated relationship element instances may be invalidated by writes of the relationship addition operator to the role element fields (see Table 3.18 on page 69). As shown previously in the first case of this proof, o is an invalidator for any such instances. □

The following proposition refines Lemma 5.22 with regard to the uniqueness assertions of extent invariants. In particular, the proposition guarantees for the execution of a non-interposed relationship element method that the uniqueness assertions of the invariant of the relationship extent instance in which the current relationship element receiver instance resides also hold in the final state of the method execution provided that they held in the initial state:

Proposition 5.24 (Invalidation of uniqueness assertions): For the execution of any non-interposed relationship element method r on any relationship element instance o and for all semi-visible states s_i of r's execution, the uniqueness assertions of the invariant of the relationship extent instance ox in which o resides cannot be invalidated between the semi-visible states s_0 and s_i.

Proof 5.25: A routine can only invalidate the uniqueness assertions of the invariant of a relationship extent instance between the routine's semi-visible states s_0 and s_i by writing to the content of the extent instance or by (possibly transitively) invoking a routine that writes to the content of the extent instance. According to Table 4.3 on page 80, only the extent instance itself is allowed to directly write to its content. Furthermore, according to parameter $\mathbb{C}$ (see Table 4.5 on page 85) and by acyclicity of a program's participants relation (see System Invariant 4.1 on page 77), an element instance is not allowed to (possibly transitively) invoke a method on the extent instance in which the element instance resides. Thus, a non-interposed relationship element method cannot invalidate the uniqueness assertions of the invariant of the relationship extent instance in which the current relationship element receiver instance resides between the method's semi-visible states s_0 and s_i. □

To state the main soundness Theorem 5.35 we find it further convenient to introduce the concept of a *guarantor*. Such a guarantor allows us to describe the framework parameter

$\mathbb{E}$ from the perspective of the instance whose invariant gets established in relation to the instances that are responsible for the establishment.

Definition 5.26 (Guarantor): For any instance o, its guarantors are those instances that guarantee to establish the invariant of o in the final state of any routine executing on such a guarantor.

For our verification technique, the guarantors are as follows:

Proposition 5.27 (Set of guarantors): For the possible instances in a successfully type checked and verified Rumer program the guarantors for such an instance o are:

- For an application instance o: the application instance o.
- For a relationship extent instance o: the relationship extent instance o.
- For a relationship element instance o: the relationship element instance o and the relationship extent instance in which o resides.
- For a role element instance o: *n/a*.
- For an entity extent instance o: the entity extent instance o.
- For an entity element instance o: the entity element instance o and the entity extent instance in which o resides.

Proof 5.28: An instance's guarantors follow directly from parameter $\mathbb{E}$ (see corresponding routine-specific instantiations in Table 5.5, Table 5.6, Table 5.7, Table 5.8, Table 5.9, Table 5.10, Table 5.11, and Table 5.12) as well as from the fact that extent instances are mutually disjoint (see System Invariant 3.6 on page 71). The latter guarantees that element instances reside in exactly one extent instance. □

The following corollary follows from Proposition 5.27 and makes explicit that mold instances are not guarantors for any other instances:

Corollary 5.29 (Molds are not guarantors): There exist no instances for whose invariants a mold instance can be a guarantor.

Proof 5.30: Corollary 5.29 is immediate from Proposition 5.27. □

The following corollary points out that a guarantor for an instance is always also an invalidator for that instance:

Corollary 5.31 (Guarantors are invalidators): For any instance o, if o' is a guarantor for o, then o' also is an invalidator for o.

Proof 5.32: Corollary 5.31 is immediate from Proposition 5.15 and Proposition 5.27. □

In general, the converse of Corollary 5.31 does not hold. However, the following lemma indicates that under certain circumstances invalidators for instances are also guarantors for these instances.

Lemma 5.33 (Some invalidators are guarantors): For any routine execution r on an instance o and for any instance o' that is instantiated during r's execution (i.e., o' was either fresh in r's initial state or did not exist in r's initial state, but exists in r's final state), if o is an invalidator for o', then o must also be a guarantor for o'.

Proof 5.34: An instance o' can only be instantiated during the execution of a routine r on an instance o if r is a generating routine that generates o' (see Definition 5.9) or if r directly or transitively invokes a generating routine that generates o'. We consider each case separately:

(r is generating routine): According to Definition 5.9, the generated instance o' must either be *(i)* an extent instance or *(ii)* an element instance. In the first case *(i)*, r must be a (possibly default) extent constructor and $o = o'$. According to Proposition 5.15 and Proposition 5.27, o is both its own invalidator and guarantor. In the second case *(ii)*, r must be an extent constructor or extent method, and, according to Proposition 5.12, o' resides in o or, if o' is a role element instance, is related by relationship element instances that reside in o. According to Proposition 5.15, o is an invalidator for all the element instances that reside in it, and, according to Proposition 5.27, o is also a guarantor for any of these element instances.

(r is caller of generating routine): If the generated instance o' is an extent instance, then r must directly or transitively invoke a (possibly default) extent constructor on the instance o'. According to Proposition 5.15, the only invalidators for o' are o' itself and all element instances residing in o', and, if o' is a relationship extent instance, all role element instances related by the relationship element instances residing in o'. According to Definition 3.1 on page 62, these invalidators are all of the same base type: $\mathit{BaseType}(o')$. Furthermore, we know by parameter $\mathbb{C}$ (see Table 4.5 on page 85) that either $o' = o$ or that the base type of the callee o' is a direct or transitive participant of the base type of the caller o. However, since o' is fresh in the constructor's initial state, $o \neq o'$. By acyclicity of a program's participants relation (see System Invariant 4.1 on page 77), we know further that $\mathit{BaseType}(o') \neq \mathit{BaseType}(o)$. Since invalidators for o' must be of $\mathit{BaseType}(o')$, o cannot be an invalidator for o', and Lemma 5.33 follows vacuously. Otherwise, the generated instance o' must be an element instance that resides in an extent instance o'', and r must directly or transitively invoke an extent constructor or extent method on the instance o''. According to Proposition 5.15, the only invalidators for such an element instance o' is o', the extent instance o'' in which o' resides, and, if o' is a relationship element instance, the role element instances related by o' (and, if o' is an element instance of an A-malign and/or B-malign relationship R with participant base types A and B and role identifiers a and b, respectively, the relationship element instances that reside in the extent instance o'' and that have the role element instance $o'.a$ and/or $o'.b$, respectively, as participant). Of these invalidators, only o' itself and its extent instance o'' are guarantors for o'. According to Definition 3.1 on page 62, these invalidators and guarantors are all of the same base type: $\mathit{BaseType}(o'')$. Furthermore, we know by parameter $\mathbb{C}$ (see Table 4.5 on

page 85) that either $o'' = o$ or that the base type of the callee o'' is a direct or transitive participant of the base type of the caller o. If $o'' = o$, then we know that o is both an invalidator and guarantor for o' since o'' is both an invalidator and guarantor for o'. If the base type of the callee o'' is a direct or transitive participant of the base type of the caller o, we can conclude by acyclicity of a program's participants relation that $\mathit{BaseType}(o'') \neq \mathit{BaseType}(o)$. Since invalidators for o' must be of $\mathit{BaseType}(o'')$, o cannot be an invalidator for o', and Lemma 5.33 follows vacuously □

The main soundness theorem for our verification technique finally becomes:

Theorem 5.35 (Soundness): For any routine execution r on an instance o in a successfully type checked and verified Rumer program, if the invariants $\mathbb{X}_{(o,r)}$ hold in the initial state of r's execution, then:

(1) in all semi-visible states s_i of r's execution:
   (a) at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_i;
   (b) if s_i is a pre-state of a direct routine invocation r' on an instance o', the invariants $\mathbb{X}_{(o',r')}$ hold;

(2) in the final state s_{final} of r's execution: the invariants of those instances for which o is an invalidator are preserved (since s_0) and also established for any of these instances for which o is a guarantor, and the invariants $\mathbb{X}_{(o,r)}$ hold.

Proof 5.36: We prove Theorem 5.35 by strong induction on the number k of transitive routine invocations made during the execution of a routine r. Let k be an arbitrary natural number of transitive routine invocations. By induction, we can assume that the theorem holds for any natural number j of transitive routine invocations such that $j < k$ and we must show that it also holds for k transitive routine invocations. We consider each routine kind in Rumer separately (see Table 4.5 on page 85 for an overview of different routine kinds):

Case 1 (Application action): For the execution of an application action r on the singleton application instance o, we assume that the invariants $\mathbb{X}_{(o,r)}$ hold in r's initial state and show that consequences (1) and (2) hold for k transitive routine invocations made during r's execution:

(1): To prove (1), we proceed by secondary weak induction on the index i of semi-visible states of r's execution:

(i = 0): Base case:
   (a) Immediate since no invariants can yet have been lost.
   (b) Vacuous since the initial state is not a pre-state of a routine invocation.

(i = n + 1): Let n be an arbitrary natural number. By inner induction, we can assume that (1) holds in the n-th semi-visible state s_n and we must show that it holds in state s_{n+1}. There are two cases to be considered:

($n+1$ is odd): Control remains within r between s_n and s_{n+1}.

(a) By inner induction, we know that (1) (a) holds in s_n and, thus, that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_n. By Lemma 5.22, we know further that at most the invariants of those instances for which o is an invalidator are lost between s_n and s_{n+1}. Consequently, we can conclude that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}.

(b) If s_{n+1} is a pre-state of a direct routine invocation r' on an instance o', then, according to parameter $\mathbb{C}$ (see Table 4.5 on page 85), we know that the callee o' must be any of the following instances, for any relationship or entity T:

($o' = o$): The invoked routine r' must be an application action. According to Table 5.5, this routine expects the invariant of the current application receiver instance o' as well as the invariants of all extent instances and element instances of entities or relationships declared in a Rumer program. Since $o' = o$ and since both the called routine and the calling routine are of the same kind, the expected invariants $\mathbb{X}_{(o',r')}$ are exactly the invariants $\mathbb{X}_{(o,r)}$. The invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, this is the invariant of the application receiver instance o only. Given the proof obligation $\mathbb{B}_{(o,r)}$ (see Table 5.5) to re-establish the invariant of the application receiver instance o, we are guaranteed that the invariants $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

($o' =$ instance of base type T, but not mold instance): The base type of the callee o' is a direct or transitive participant of the base type of the caller o. By Proposition 5.3 and Definition 5.2 it thus holds that $\mathit{BaseType}(o') \neq \mathit{BaseType}(o)$. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, this is the invariant of the application receiver instance o only. Since the invariants $\mathbb{X}_{(o,r)}$ are known to hold in s_0 by assumption, we thus know that the invariants $\mathbb{X}_{o,r} \setminus \{\mathit{inv}_{o''} \mid \mathit{BaseType}(o'') = \mathit{BaseType}(o)\}$ hold in s_{n+1}. Since $\mathit{BaseType}(o') \neq \mathit{BaseType}(o)$ and since the invariants $\mathbb{X}_{o,r} \setminus \{\mathit{inv}_{o''} \mid \mathit{BaseType}(o'') = \mathit{BaseType}(o)\}$ hold in s_{n+1}, we conclude by Lemma 5.5 that the invariants $\mathbb{X}_{o',r'}$ hold in s_{n+1}.

($o' =$ instance of mold type): The invoked routine r' must be a mold constructor or mold initializer. According to Table 5.13, these routines do not expect any invariants to hold in their initial and final

states, and, thus, the invariants $\mathbb{X}_{o',r'}$ trivially hold in s_{n+1}.

(n + 1 is even): A direct routine invocation r' on an instance o' is made between s_n and s_{n+1}.

(a) By inner induction, we know that (1) holds in s_n and, thus, that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_n and that the invariants $\mathbb{X}_{o',r'}$ hold in s_n. Since there must be strictly fewer than k routine invocations made during the execution of r', we know, by outer induction, that Theorem 5.35 holds for the execution of r'. Since the invariants $\mathbb{X}_{o',r'}$ hold in s_n, we know further that (1) and (2) hold for r'. In particular, we know that at most the invariants of those instances for which o' is an invalidator are lost between s_n and s_{n+1} and that the invariants of those instances for which o' is an invalidator are preserved between s_n and s_{n+1} and also established in s_{n+1} for any of these instances for which o' is a guarantor. According to Lemma 5.20, we know further that of the invariants for which o' is an invalidator, since they are preserved by r', only those invariants can be lost between s_n and s_{n+1} by r' whose instance was either fresh in s_n or did not exist in s_n. By Lemma 5.33, however, we know that for the instances that were either fresh in s_n or did not exist in s_n, o' is not only an invalidator but also a guarantor. Since the invariants of any of these instances for which o' is a guarantor are established in s_{n+1}, no invariants can be lost between s_n and s_{n+1}, and, thus, we can conclude that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}.

(b) Vacuous since state s_{n+1} is not a pre-state of a routine invocation.

(2): As shown previously in part (1), we know that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{final}. According to Proposition 5.15, this is the invariant of the application receiver instance o only. The proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.5) ensures that the invariant of o is established in s_{final} and, thus, guaranteed to be preserved between s_0 and s_{final}. According to Proposition 5.27, the application receiver instance o is also its only guarantor. Since the invariant of o is guaranteed to be established in s_{final} by the proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.5), the invariants of all instances for which o is a guarantor are guaranteed to be established in s_{final}. According to Table 5.5, the invariant of the current application receiver instance o as well as the invariants of all extent instances and element instances of entities or relationships declared in a Rumer program are expected to hold in the visible states of r's execution. By assumption, we know that the invariants $\mathbb{X}_{(o,r)}$ hold in s_0. Since the invariant of o is the only invariant that can be lost between the semi-visible states s_0 and s_{final} and since the invariant of o is guaranteed to be preserved between s_0 and s_{final} and established in s_{final}, the invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_{final}.

Case 2 (Default relationship extent constructor): For the execution of a default relationship extent constructor r on a relationship extent instance o, we assume that the invariants $\mathbb{X}_{(o,r)}$ hold in r's initial state and show that consequences (1) and (2) hold for k transitive routine invocations made during r's execution:

(1): Since a default relationship constructor cannot invoke any other routines (see Table 4.5 on page 85), we prove (1) by investigating the only two semi-visible states of r's execution:

(Initial state):

(a) Immediate since no invariants can yet have been lost.

(b) Vacuous since the initial state is not a pre-state of a routine invocation.

(Final state):

(a) Since s_{final} is the immediate successor state of s_0 and, thus, control remains within r between s_0 and s_{final}, we know by Lemma 5.22 that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{final}.

(b) Vacuous since the final state is not a pre-state of a routine invocation.

(2): As shown previously in part (1), we know that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{final}. According to Proposition 5.15, the relationship extent receiver instance o is an invalidator for itself and for all the relationship element instances that reside in o. However, since the relationship extent receiver instance of a default relationship extent constructor is empty (see Table 3.20 on page 71), the relationship extent receiver instance o is its only invalidator, and, thus, its invariant is the only invariant that can get lost between the semi-visible states s_0 and s_{final}. The proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.6) ensures that the invariant of o is established in s_{final} and, thus, guaranteed to be preserved between s_0 and s_{final}. According to Proposition 5.27, the relationship extent receiver instance o is a guarantor for itself as well as for all the relationship element instances that reside in o. However, since the relationship extent receiver instance of a default relationship extent constructor is empty (see Table 3.20 on page 71), the relationship extent receiver instance o is its only guarantor. Since the invariant of o is guaranteed to be established in s_{final} by the proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.6), the invariants of all instances for which o is a guarantor are guaranteed to be established in s_{final}. According to Table 5.6, the invariant of o is not expected to hold in the visible states of r's execution. By assumption, we know that the invariants $\mathbb{X}_{(o,r)}$ hold in s_0. Since the invariant of o is the only invariant that can be lost between the semi-visible states s_0 and s_{final}, the invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_{final}.

Case 3 (Default entity extent constructor): The proof is analogous to the proof of a default relationship extent constructor (see Case 2) since default relationship extent constructors and default entity extent constructors have analogous instantiations for all the framework parameters.

Case 4 (Relationship extent constructor): For the execution of a relationship extent constructor r on a relationship extent instance o, we assume that the invariants $\mathbb{X}_{(o,r)}$ hold in r's initial state and show that consequences (1) and (2) hold for k transitive routine invocations made during r's execution. The proof is in parts identical to the proof of Case 1. Therefore, we only prove the parts that are specific to the execution of a relationship extent constructor and refer to the proof of Case 1 otherwise. We consider an extent constructor of a relationship R with the participant base types A and B and the role identifiers a and b, respectively:

(1): To prove (1), we proceed by secondary weak induction on the index i of semi-visible states of r's execution:

(i = 0): See proof of Case 1.

(i = n + 1): Let n be an arbitrary natural number. By inner induction, we can assume that (1) holds in the n-th semi-visible state s_n and we must show that it holds in state s_{n+1}. There are two cases to be considered:

(n + 1 is odd): Control remains within r between s_n and s_{n+1}.

(a) See proof of Case 1.

(b) If s_{n+1} is a pre-state of a direct routine invocation r' on an instance o', then, according to parameter $\mathbb{C}$ (see Table 4.5 on page 85), we know that the callee o' must be any of the following instances:

($o' = o$): The invoked routine r' must be a relationship extent method. According to Table 5.8, this routine expects the invariants of all extent instances and element instances of R as well as the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant. Since $o' = o$ and since both the called routine and the calling routine are declared by the same base type, the expected invariants of the called routine (see Table 5.8) relate to the ones of the calling routine (see Table 5.7) as follows: $\mathbb{X}_{(o',r')} = \mathbb{X}_{(o,r)} \cup \{inv_o\} \cup \{inv_{o''} \mid o'' \in o\}$. The invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o. Given the proof obligation $\mathbb{B}_{(o,r)}$ (see Table 5.7) to re-establish the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o, we are guaranteed that the invariants $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

(o' = instance of type Element⟨R⟩): The invoked routine r' must be a non-interposed relationship element method. In accordance with the

different relationship kinds (see Table 5.4), we consider the following cases:

(R = free): According to Table 5.9, the called method expects the invariant of the current relationship element receiver instance o' as well as the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant. Since both the called routine and the calling routine are declared by the same base type, the expected invariants of the called routine (see Table 5.9) relate to the ones of the calling routine (see Table 5.7) as follows: $\mathbb{X}_{(o',r')} = (\mathbb{X}_{(o,r)} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\}) \cup \{inv_{o'}\}$. The invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption, and it holds that $inv_{o'} \in \mathbb{X}_{(o,r)}$ if $o' \notin o$. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o. Given the proof obligation $\mathbb{B}_{(o,r)}$ (see Table 5.7) to re-establish the invariant of the relationship element receiver instance o' if o' resides in o, we are guaranteed that the invariants $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

(R = A-malign): Identical to Case (R = free) since malignity does not affect the set of expected invariants (see Table 5.11).

(R = B-malign): Identical to Case (R = free) since malignity does not affect the set of expected invariants (see Table 5.11).

(R = A-malign–B-malign): Identical to Case (R = free) since malignity does not affect the set of expected invariants (see Table 5.11).

(R = A-benign): According to Table 5.9 and Table 5.10, the called method expects the invariant of the current relationship element receiver instance o', the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant, as well as the B-uniqueness assertion of the invariant of the extent instance ox' in which o' resides. Since both the called routine and the calling routine are declared by the same base type, the expected invariants and uniqueness assertions of the called routine (see Table 5.9 and Table 5.10) relate to the expected invariants of the calling routine (see Table 5.7) as follows: $\mathbb{X}_{(o',r')} = (\mathbb{X}_{(o,r)} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\}) \cup \{inv_{o'}\} \cup \{B\text{-}unique_{ox'}\}$. The invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption, and it holds that $inv_{o'} \in \mathbb{X}_{(o,r)} \wedge inv_{ox'} \in \mathbb{X}_{(o,r)}$ if $o' \notin o$. As shown previously in part (1) (a), we know further that at most the invariants of those instances for

which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o. Given the proof obligation $\mathbb{B}_{(o,r)}$ (see Table 5.7) to re-establish the invariant of the relationship element receiver instance o' as well as the B-uniqueness assertion of the invariant of o if o' resides in o, we are guaranteed that the invariants and uniqueness assertions $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

- **(R = A-benign–B-malign):** Identical to previous case since malignity does not affect the set of expected invariants (see Table 5.11).
- **(R = B-benign):** Analogous to Case (R = A-benign) but with inverted sides.
- **(R = A-malign–B-benign):** Identical to previous case since malignity does not affect the set of expected invariants (see Table 5.11).
- **(R = A-benign–B-benign):** Combination of Case (R = A-benign) and Case (R = B-benign).

(o' = instance of type Element⟨R.a⟩): The invoked routine r' must be an interposed relationship element method. According to Table 5.12, this routine expects the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant. Since both the called routine and the calling routine are declared by the same base type, the expected invariants of the called routine (see Table 5.12) relate to the ones of the calling routine (see Table 5.7) as follows: $\mathbb{X}_{(o',r')} = (\mathbb{X}_{(o,r)} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\})$. The invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o. Thus, we are guaranteed that the invariants $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

(o' = instance of type Element⟨R.b⟩): Identical to previous case.

(o' = instance of base type A, but not mold instance): The base type of the callee o' must be a direct participant of the base type of the caller o. By Proposition 5.3 and Definition 5.2 it thus holds that $BaseType(o') \neq BaseType(o)$. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the invariant of the relationship extent receiver instance o as well as the invariants

of the relationship element instances residing in o. Since the invariants $\mathbb{X}_{(o,r)}$ are known to hold in s_0 by assumption, we thus know that the invariants $\mathbb{X}_{o,r} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\}$ hold in s_{n+1}. Since $BaseType(o') \neq BaseType(o)$ and since the invariants $\mathbb{X}_{o,r} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\}$ hold in s_{n+1}, we conclude by Lemma 5.5 that the invariants $\mathbb{X}_{o',r'}$ hold in s_{n+1}.

(o' = instance of base type B, but not mold instance): Identical to previous case.

(o' = instance of mold type): See proof of Case 1.

(n + 1 is even): See proof of Case 1.

(2): As shown previously in part (1) (a), we know that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{final}. According to Proposition 5.15, these are the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o. The proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.7) ensures that the invariant of the relationship extent receiver instance o as well as the invariants of all relationship element instances residing in o are established in s_{final} and, thus, guaranteed to be preserved between s_0 and s_{final}. According to Proposition 5.27, the relationship extent receiver instance o is a guarantor for the relationship extent receiver instance o as well as for the invariants of all relationship element instances residing in o. Since the invariant of the relationship extent receiver instance o as well as the invariants of all relationship element instances residing in o are guaranteed to be established in s_{final} by the proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.7), the invariants of all instances for which o is a guarantor are guaranteed to be established in s_{final}. According to Table 5.7, the following invariants $\mathbb{X}_{(o,r)}$ are expected to hold in the visible states of r's execution: the invariants of all extent instances and element instances of R, except for the relationship extent receiver instance o and the invariants of the relationship element instances residing in o, as well as the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant. By assumption, we know that the invariants $\mathbb{X}_{(o,r)}$ hold in s_0. Since the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o are the only invariants that can be lost between the semi-visible states s_0 and s_{final}, the invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_{final}.

Case 5 (Entity extent constructor): For the execution of an entity extent constructor r on an entity extent instance o, we assume that the invariants $\mathbb{X}_{(o,r)}$ hold in r's initial state and show that consequences (1) and (2) hold for k transitive routine invocations made during r's execution. The proof is in parts identical to the proof of Case 1. Furthermore, the proof is in parts analogous to the one of a relationship extent constructor (Case 4) since the two constructors only differ in parameter $\mathbb{C}$, but have analogous instantiations for the remaining framework parameters. Therefore, we only prove the parts that are specific to the execution of a entity extent constructor and refer to the proof of Case 1

for the generic parts of the proof and to the proof of Case 4 for the parts of the proof that are analogous to the one of a relationship extent constructor. We consider an extent constructor of an entity E:

(1): To prove (1), we proceed by secondary weak induction on the index *i* of semi-visible states of *r*'s execution:

(i = 0): See proof of Case 1.

(i = n + 1): Let *n* be an arbitrary natural number. By inner induction, we can assume that (1) holds in the *n*-th semi-visible state s_n and we must show that it holds in state s_{n+1}. There are two cases to be considered:

(n + 1 is odd): Control remains within *r* between s_n and s_{n+1}.

(a) See proof of Case 1.

(b) If s_{n+1} is a pre-state of a direct routine invocation r' on an instance o', then, according to parameter $\mathbb{C}$ (see Table 4.5 on page 85), we know that the callee o' must be any of the below listed instances. For some of the cases, the proofs are analogous to the corresponding ones of a relationship extent constructor (see Case 4) since they only consider the framework parameters $\mathbb{X}$ and $\mathbb{B}$ as well as the concept of an invalidator (parameters $\mathbb{U}$ and $\mathbb{D}$). As pointed out previously, the two constructors differ in the framework parameter $\mathbb{C}$, but have analogous instantiations for the remaining framework parameters.

($o' = o$): Analogous to proof of Case 4.

(o' = instance of type Element⟨E⟩): Analogous to proof of Case 4, Subcase (R = free).

(o' = instance of mold type): See proof of Case 1.

(n + 1 is even): See proof of Case 1.

(2): This proof is analogous to the one of a relationship extent constructor (see Case 4) since the proof only considers the framework parameters $\mathbb{E}$ and $\mathbb{X}$ as well as the concepts of an invalidator (parameters $\mathbb{U}$ and $\mathbb{D}$) and guarantor (parameter $\mathbb{E}$). As pointed out previously, the two constructors differ in the framework parameter $\mathbb{C}$, but have analogous instantiations for the remaining framework parameters.

Case 6 (Relationship extent method): For the execution of a relationship extent method *r* on a relationship extent instance *o*, we assume that the invariants $\mathbb{X}_{(o,r)}$ hold in *r*'s initial state and show that consequences (1) and (2) hold for *k* transitive routine invocations made during *r*'s execution. The proof is in parts identical to the proof of Case 1. Furthermore, the proof resembles the one of a relationship extent constructor (Case 4) as relationship extent constructors and relationship extent methods share the same framework parameters $\mathbb{V}$, $\mathbb{B}$, $\mathbb{E}$, $\mathbb{C}$, $\mathbb{U}$, and $\mathbb{D}$. However, since the two routines differ in the framework parameter $\mathbb{X}$ and since every routine-specific part of the proof relies on this parameter, we provide the full proof of a relationship extent method and refer to the proof of Case 1 for the

generic parts of the proof. We consider an extent constructor of a relationship R with the participant base types A and B and the role identifiers a and b, respectively:

(1): To prove (1), we proceed by secondary weak induction on the index i of semi-visible states of r's execution:

(i = 0): See proof of Case 1.

(i = n + 1): Let n be an arbitrary natural number. By inner induction, we can assume that (1) holds in the n-th semi-visible state s_n and we must show that it holds in state s_{n+1}. There are two cases to be considered:

(n + 1 is odd): Control remains within r between s_n and s_{n+1}.

(a) See proof of Case 1.

(b) If s_{n+1} is a pre-state of a direct routine invocation r' on an instance o', then, according to parameter $\mathbb{C}$ (see Table 4.5 on page 85), we know that the callee o' must be any of the following instances:

($o' = o$): The invoked routine r' must be a relationship extent method. According to Table 5.8, this routine expects the invariants of all extent instances and element instances of R as well as the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant. Since $o' = o$ and since both the called routine and the calling routine are of the same kind, the expected invariants $\mathbb{X}_{(o',r')}$ are exactly the invariants $\mathbb{X}_{(o,r)}$. The invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o. Given the proof obligation $\mathbb{B}_{(o,r)}$ (see Table 5.8) to re-establish the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o, we are guaranteed that the invariants $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

(o' = instance of type Element⟨R⟩): The invoked routine r' must be a non-interposed relationship element method. In accordance with the different relationship kinds (see Table 5.4), we consider the following cases:

(R = free): According to Table 5.9, the called method expects the invariant of the current relationship element receiver instance o' as well as the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant. Since both the called routine and the calling routine are declared by the same base type, the expected

invariants of the called routine (see Table 5.9) relate to the ones of the calling routine (see Table 5.8) as follows: $\mathbb{X}_{(o',r')} = (\mathbb{X}_{(o,r)} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\}) \cup \{inv_{o'}\}$. The invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption, and it holds that $inv_{o'} \in \mathbb{X}_{(o,r)}$. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o. Given the proof obligation $\mathbb{B}_{(o,r)}$ (see Table 5.8) to re-establish the invariant of the relationship element receiver instance o' if o' resides in o, we are guaranteed that the invariants $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

(R = A-malign): Identical to Case (R = free) since malignity does not affect the set of expected invariants (see Table 5.11).

(R = B-malign): Identical to Case (R = free) since malignity does not affect the set of expected invariants (see Table 5.11).

(R = A-malign–B-malign): Identical to Case (R = free) since malignity does not affect the set of expected invariants (see Table 5.11).

(R = A-benign): According to Table 5.9 and Table 5.10, the called method expects the invariant of the current relationship element receiver instance o', the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant, as well as the B-uniqueness assertion of the invariant of the extent instance ox' in which o' resides. Since both the called routine and the calling routine are declared by the same base type, the expected invariants and uniqueness assertions of the called routine (see Table 5.9 and Table 5.10) relate to the expected invariants of the calling routine (see Table 5.8) as follows: $\mathbb{X}_{(o',r')} = (\mathbb{X}_{(o,r)} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\}) \cup \{inv_{o'}\} \cup \{B\text{-}unique_{ox'}\}$. The invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption, and it holds that $inv_{o'} \in \mathbb{X}_{(o,r)} \wedge inv_{ox'} \in \mathbb{X}_{(o,r)}$. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o. Given the proof obligation $\mathbb{B}_{(o,r)}$ (see Table 5.8) to re-establish the invariant of the relationship element receiver instance o' as well as the B-uniqueness assertion of the invariant of o if o' resides in o, we are guaranteed that the invariants and uniqueness assertions $\mathbb{X}_{(o',r')}$

hold in s_{n+1}.

(R = A-benign–B-malign): Identical to previous case since malignity does not affect the set of expected invariants (see Table 5.11).

(R = B-benign): Analogous to Case (R = A-benign) but with inverted sides.

(R = A-malign–B-benign): Identical to previous case since malignity does not affect the set of expected invariants (see Table 5.11).

(R = A-benign–B-benign): Combination of Case (R = A-benign) and Case (R = B-benign).

(o' = instance of type Element⟨R.a⟩): The invoked routine r' must be an interposed relationship element method. According to Table 5.12, this routine expects the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant. Since both the called routine and the calling routine are declared by the same base type, the expected invariants of the called routine (see Table 5.12) relate to the ones of the calling routine (see Table 5.8) as follows: $\mathbb{X}_{(o',r')} = (\mathbb{X}_{(o,r)} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\})$. The invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o. Thus, we are guaranteed that the invariants $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

(o' = instance of type Element⟨R.b⟩): Identical to previous case.

(o' = instance of base type A, but not mold instance): The base type of the callee o' must be a direct participant of the base type of the caller o. By Proposition 5.3 and Definition 5.2 it thus holds that $BaseType(o') \neq BaseType(o)$. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o. Since the invariants $\mathbb{X}_{(o,r)}$ are known to hold in s_0 by assumption, we thus know that the invariants $\mathbb{X}_{o,r} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\}$ hold in s_{n+1}. Since $BaseType(o') \neq BaseType(o)$ and since the invariants $\mathbb{X}_{o,r} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\}$ hold in s_{n+1}, we conclude by Lemma 5.5 that the invariants $\mathbb{X}_{o',r'}$ hold in s_{n+1}.

(o' = instance of base type B, but not mold instance): Identical to previous case.

(o' = instance of mold type): See proof of Case 1.

(n + 1 is even): See proof of Case 1.

(2): As shown previously in part (1) (a), we know that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{final}. According to Proposition 5.15, these are the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o. The proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.8) ensures that the invariant of the relationship extent receiver instance o as well as the invariants of all relationship element instances residing in o are established in s_{final} and, thus, guaranteed to be preserved between s_0 and s_{final}. According to Proposition 5.27, the relationship extent receiver instance o is a guarantor for the relationship extent receiver instance o as well as for the invariants of all relationship element instances residing in o. Since the invariant of the relationship extent receiver instance o as well as the invariants of all relationship element instances residing in o are guaranteed to be established in s_{final} by the proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.8), the invariants of all instances for which o is a guarantor are guaranteed to be established in s_{final}. According to Table 5.8, the following invariants $\mathbb{X}_{(o,r)}$ are expected to hold in the visible states of r's execution: the invariants of all extent instances and element instances of R as well as the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant. By assumption, we know that the invariants $\mathbb{X}_{(o,r)}$ hold in s_0. Since the invariant of the relationship extent receiver instance o as well as the invariants of the relationship element instances residing in o are the only invariants that can be lost between the semi-visible states s_0 and s_{final} and since those invariants are guaranteed to be preserved between s_0 and s_{final} and established in s_{final}, the invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_{final}.

Case 7 (Entity extent method): For the execution of an entity extent method r on an entity extent instance o, we assume that the invariants $\mathbb{X}_{(o,r)}$ hold in r's initial state and show that consequences (1) and (2) hold for k transitive routine invocations made during r's execution. The proof is in parts identical to the proof of Case 1. Furthermore, the proof is in parts analogous to the one of a relationship extent method (Case 6) since the two methods only differ in the framework parameter $\mathbb{C}$, but have analogous instantiations for the remaining framework parameters. Moreover, the proof resembles the one of an entity extent constructor (Case 5) as entity extent constructors and entity extent methods share the same framework parameters $\mathbb{V}$, $\mathbb{B}$, $\mathbb{E}$, $\mathbb{C}$, $\mathbb{U}$, and $\mathbb{D}$, and only differ in the framework parameter $\mathbb{X}$. However, since every routine-specific part of the proof relies on the parameter $\mathbb{X}$, we cannot leverage the resemblance of entity extent methods to entity extent constructors for the proof. Therefore, the proof of an entity extent method only relies on the proofs of an application action (Case 1) and a relationship extent method (Case 6). We refer to the former for the generic parts of the proof and to the latter for the parts of the proof that are analogous to the one of a relationship extent method. We consider an extent method of an entity E:

(1): To prove (1), we proceed by secondary weak induction on the index *i* of semi-visible states of *r*'s execution:

(i = 0): See proof of Case 1.

(i = n + 1): Let *n* be an arbitrary natural number. By inner induction, we can assume that (1) holds in the *n*-th semi-visible state s_n and we must show that it holds in state s_{n+1}. There are two cases to be considered:

(n + 1 is odd): Control remains within *r* between s_n and s_{n+1}.

(a) See proof of Case 1.

(b) If s_{n+1} is a pre-state of a direct routine invocation r' on an instance o', then, according to parameter $\mathbb{C}$ (see Table 4.5 on page 85), we know that the callee o' must be any of the below listed instances. For some of the cases, the proofs are analogous to the corresponding ones of a relationship extent method (see Case 6) since they only consider the framework parameters $\mathbb{X}$ and $\mathbb{B}$ as well as the concept of an invalidator (parameters $\mathbb{U}$ and $\mathbb{D}$). As pointed out previously, the two methods differ in the framework parameter $\mathbb{C}$, but have analogous instantiations for the remaining framework parameters.

($o' = o$): Analogous to proof of Case 6.

(o' = instance of type Element⟨E⟩): Analogous to proof of Case 6, Subcase (R = free).

(o' = instance of mold type): See proof of Case 1.

(n + 1 is even): See proof of Case 1.

(2): This proof is analogous to the one of a relationship extent method (see Case 6) since the proof only considers the framework parameters $\mathbb{E}$ and $\mathbb{X}$ as well as the concepts of an invalidator (parameters $\mathbb{U}$ and $\mathbb{D}$) and guarantor (parameter $\mathbb{E}$). As pointed out previously, the two methods differ in the framework parameter $\mathbb{C}$, but have analogous instantiations for the remaining framework parameters.

Case 8 (Mold constructor or initializer): We collate the proofs of non-interposed relationship mold constructors and initializers as well as of entity mold constructors and initializers since these routines share the same framework parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, $\mathbb{E}$, and $\mathbb{C}$ and since their receiver instances are invalidators (determined by the framework parameters $\mathbb{U}$ and $\mathbb{D}$) for the same set of instances. For the execution of a mold constructor or mold initializer *r* on a mold instance *o*, we assume that the invariants $\mathbb{X}_{(o,r)}$ hold in *r*'s initial state and show that consequences (1) and (2) hold for *k* transitive routine invocations made during *r*'s execution.

(1): Since a mold constructor or initializer cannot invoke any other routines (see Table 4.5 on page 85), we prove (1) by investigating the only two semi-visible states of *r*'s execution:

(Initial state):

(a) Immediate since no invariants can yet have been lost.

(b) Vacuous since the initial state is not a pre-state of a routine invocation.

(Final state):

(a) Since s_{final} is the immediate successor state of s_0 and, thus, control remains within r between s_0 and s_{final}, we know by Lemma 5.22 that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{final}.

(b) Vacuous since the final state is not a pre-state of a routine invocation.

(2): According to Corollary 5.17 and Corollary 5.29, the mold receiver instance o is neither an invalidator nor a guarantor for any instances. Consequently, the invariants of all instances for which o is an invalidator and guarantor are trivially preserved between s_0 and s_{final} and trivially established in s_{final}, respectively. According to Table 5.13, $\mathbb{X}_{(o,r)}$ is the empty set, and, thus, the invariants $\mathbb{X}_{(o,r)}$ hold trivially in s_{final}.

Case 9 (Non-interposed relationship element method): For the execution of a non-interposed relationship element method r on a relationship element instance o, we assume that the invariants $\mathbb{X}_{(o,r)}$ hold in r's initial state and show that consequences (1) and (2) hold for k transitive routine invocations made during r's execution. The proof is in parts identical to the proof of Case 1. Therefore, we only prove the parts that are specific to the execution of a non-interposed relationship element method and refer to the proof of Case 1 otherwise. We consider a non-interposed relationship element method of a relationship R with the participant base types A and B and the role identifiers a and b, respectively:

(1): To prove (1), we proceed by secondary weak induction on the index i of semi-visible states of r's execution:

(i = 0): See proof of Case 1.

(i = n + 1): Let n be an arbitrary natural number. By inner induction, we can assume that (1) holds in the n-th semi-visible state s_n and we must show that it holds in state s_{n+1}. There are two cases to be considered:

(n + 1 is odd): Control remains within r between s_n and s_{n+1}.

(a) See proof of Case 1.

(b) If s_{n+1} is a pre-state of a direct routine invocation r' on an instance o', then, according to parameter $\mathbb{C}$ (see Table 4.5 on page 85), we know that the callee o' must be any of the following instances:

($o' = o$): The invoked routine r' must be a non-interposed relationship element method. In accordance with the different relationship kinds (see Table 5.4), we consider the following cases:

(R = free): According to Table 5.9, the called method expects the invariant of the current relationship element receiver instance o'

as well as the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant. Since $o' = o$ and since both the called routine and the calling routine are of the same kind, the expected invariants $\mathbb{X}_{(o',r')}$ are exactly the invariants $\mathbb{X}_{(o,r)}$. The invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the invariant of the relationship element receiver instance o as well as the invariant of the relationship extent instance in which o resides. Given the proof obligation $\mathbb{B}_{(o,r)}$ (see Table 5.9) to re-establish the invariant of the relationship element receiver instance o, we are guaranteed that the invariants $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

(R = A-benign): According to Table 5.9 and Table 5.10, the called method expects the invariant of the current relationship element receiver instance o', the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant, as well as the B-uniqueness assertion of the invariant of the extent instance in which o' resides. Since $o' = o$ and since both the called routine and the calling routine are of the same kind, the expected invariants and uniqueness assertions $\mathbb{X}_{(o',r')}$ are exactly the invariants and uniqueness assertions $\mathbb{X}_{(o,r)}$. The invariants and uniqueness assertions $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the invariant of the relationship element receiver instance o as well as the invariant of the relationship extent instance in which o resides. By the proof obligation $\mathbb{B}_{(o,r)}$ (see Table 5.9), we are guaranteed that the invariant of the relationship element receiver instance o is re-established in s_{n+1}. By Proposition 5.24, we are furthermore guaranteed that the B-uniqueness assertion of the invariant of the extent instance in which o resides is not invalidated between the semi-visible states s_0 and s_{n+1} and, thus, that this assertion still holds in s_{n+1}. Consequently, we are guaranteed that the invariants and uniqueness assertions $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

(R = B-benign): Analogous to Case (R = A-benign) but with inverted sides.

(R = A-benign–B-benign): Combination of Case (R = A-benign)

and Case (R = B-benign).

(R = A-malign): According to Table 5.9 and Table 5.11, the called method expects the invariant of the current relationship element receiver instance o' as well as the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant. Since $o' = o$ and since both the called routine and the calling routine are of the same kind, the expected invariants $\mathbb{X}_{(o',r')}$ are exactly the invariants $\mathbb{X}_{(o,r)}$. The invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the following invariants: the invariant of the relationship element receiver instance o, the invariants of all relationship element instances of R that reside in the same relationship extent instance as o and that have the role element instance $o.a$ as participant, and the invariant of the relationship extent instance in which o resides. Given the proof obligation $\mathbb{B}_{(o,r)}$ (see Table 5.11) to re-establish the invariant of the relationship element receiver instance o, we are guaranteed that the invariants $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

(R = B-malign): Analogous to Case (R = A-malign) but with inverted sides.

(R = A-malign–B-malign): Combination of Case (R = A-malign) and Case (R = B-malign).

(R = A-benign–B-malign): Combination of Case (R = A-benign) and Case (R = B-malign).

(R = A-malign–B-benign): Combination of Case (R = A-malign) and Case (R = B-benign).

(o' = instance of type Element⟨R.a⟩): The invoked routine r' must be an interposed relationship element method. According to Table 5.12, this routine expects the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant. Since both the called routine and the calling routine are declared by the same base type, the expected invariants of the called routine (see Table 5.12) relate to the ones of the calling routine (see Table 5.9) as follows: $\mathbb{X}_{(o',r')} = (\mathbb{X}_{(o,r)} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\})$. The invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the following invariants: the

invariant of the relationship element receiver instance o, the invariant of the relationship extent instance in which o resides, and, if R is A-malign and/or B-malign (last five rows in Table 5.4), the invariants of all relationship element instances of R that reside in the same relationship extent instance as o and that have the role element instance $o.a$ and/or $o.b$, respectively, as participant. Since these possibly lost invariants are all invariants of instances of $BaseType(o)$, we are guaranteed that the invariants $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

(o' = instance of type Element⟨R.b⟩): Identical to previous case.

(o' = instance of base type A, but not mold instance): The base type of the callee o' must be a direct participant of the base type of the caller o. By Proposition 5.3 and Definition 5.2 it thus holds that $BaseType(o') \neq BaseType(o)$. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the following invariants: the invariant of the relationship element receiver instance o, the invariant of the relationship extent instance in which o resides, and, if R is A-malign and/or B-malign (last five rows in Table 5.4), the invariants of all relationship element instances of R that reside in the same relationship extent instance as o and that have the role element instance $o.a$ and/or $o.b$, respectively, as participant. Since the invariants $\mathbb{X}_{(o,r)}$ are known to hold in s_0 by assumption, we thus know that the invariants $\mathbb{X}_{o,r} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\}$ hold in s_{n+1}. Since $BaseType(o') \neq BaseType(o)$ and since the invariants $\mathbb{X}_{o,r} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\}$ hold in s_{n+1}, we conclude by Lemma 5.5 that the invariants $\mathbb{X}_{o',r'}$ hold in s_{n+1}.

(o' = instance of base type B, but not mold instance): Identical to previous case.

(o' = instance of mold type): See proof of Case 1.

(n + 1 is even): See proof of Case 1.

(2): In accordance with the different relationship kinds (see Table 5.4), we consider the following cases:

(R = free): As shown previously in part (1) (a), we know that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{final}. According to Proposition 5.15, these are the invariant of the relationship element receiver instance o as well as the invariant of the relationship extent instance in which o resides. The proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.9) ensures that the invariant of the relationship element receiver instance o is established in s_{final} and that the invariant of the relationship extent instance in which o resides is preserved between s_0 and s_{final}. Thus, the invariants of all instances

for which o is an invalidator are guaranteed to be preserved between s_0 and s_{final}. According to Proposition 5.27, the relationship element receiver instance o is only a guarantor for itself. Since the invariant of the relationship element receiver instance o is guaranteed to be established in s_{final} by the proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.9), the invariants of all instances for which o is a guarantor are guaranteed to be established in s_{final}. According to Table 5.9, the following invariants $\mathbb{X}_{(o,r)}$ are expected to hold in the visible states of r's execution: the invariant of the current relationship element receiver instance o as well as the invariants of all extent and element instances of entities or relationships T (with $\mathsf{T} \neq \mathsf{R}$) of which R is not a direct or transitive participant. By assumption, we know that the invariants $\mathbb{X}_{(o,r)}$ hold in s_0. Since the invariant of the relationship element receiver instance o as well as the invariant of the relationship extent instance in which o resides are the only invariants that can be lost between the semi-visible states s_0 and s_{final} and since those invariants are guaranteed to be preserved between s_0 and s_{final} and the invariant of o is guaranteed to be established in s_{final}, the invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_{final}.

(R = A-benign): As shown previously in part (1) (a), we know that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{final}. According to Proposition 5.15, these are the invariant of the relationship element receiver instance o as well as the invariant of the relationship extent instance in which o resides. The proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.9) ensures that the invariant of the relationship element receiver instance o is established in s_{final} and that the invariant of the relationship extent instance in which o resides is preserved between s_0 and s_{final}. Thus, the invariants of all instances for which o is an invalidator are guaranteed to be preserved between s_0 and s_{final}. According to Proposition 5.27, the relationship element receiver instance o is only a guarantor for itself. Since the invariant of the relationship element receiver instance o is guaranteed to be established in s_{final} by the proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.9), the invariants of all instances for which o is a guarantor are guaranteed to be established in s_{final}. According to Table 5.9 and Table 5.10, the following invariants and uniqueness assertions $\mathbb{X}_{(o,r)}$ are expected to hold in the visible states of r's execution: the invariant of the current relationship element receiver instance o, the invariants of all extent and element instances of entities or relationships T (with $\mathsf{T} \neq \mathsf{R}$) of which R is not a direct or transitive participant, as well as the B-uniqueness assertion of the invariant of the extent instance in which o resides. By assumption, we know that the invariants and uniqueness assertions $\mathbb{X}_{(o,r)}$ hold in s_0. Since the invariant of the relationship element receiver instance o as well as the invariant of the relationship extent instance in which o resides are the only invariants that can be lost between the semi-visible states s_0 and s_{final} and since those invariants are guaranteed to be preserved between s_0 and s_{final} and the invariant of o is guaranteed to be established in s_{final}, the invariants and uniqueness assertions $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_{final}.

(R = B-benign): Analogous to Case (R = A-benign) but with inverted sides.

(R = A-benign–B-benign): Combination of Case (R = A-benign) and Case (R = B-benign).

(R = A-malign): As shown previously in part (1) (a), we know that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{final}. According to Proposition 5.15, these are the following invariants: the invariant of the relationship element receiver instance o, the invariants of all relationship element instances of R that reside in the same relationship extent instance as o and that have the role element instance $o.a$ as participant, and the invariant of the relationship extent instance in which o resides. The proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.9 and Table 5.10) ensures that the invariant of the relationship element receiver instance o is established in s_{final} and that the invariants of all relationship element instances of R that reside in the same relationship extent instance as o and that have the role element instance $o.a$ as participant as well as the invariant of the relationship extent instance in which o resides are preserved between s_0 and s_{final}. Thus, the invariants of all instances for which o is an invalidator are guaranteed to be preserved between s_0 and s_{final}. According to Proposition 5.27, the relationship element receiver instance o is only a guarantor for itself. Since the invariant of the relationship element receiver instance o is guaranteed to be established in s_{final} by the proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.9), the invariants of all instances for which o is a guarantor are guaranteed to be established in s_{final}. According to Table 5.9, the following invariants $\mathbb{X}_{(o,r)}$ are expected to hold in the visible states of r's execution: the invariant of the current relationship element receiver instance o as well as the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant. By assumption, we know that the invariants $\mathbb{X}_{(o,r)}$ hold in s_0. From all the invariants that can be lost between the semi-visible states s_0 and s_{final} only the invariant of the relationship element receiver instance o is expected to hold in the visible states of r's execution. Since this invariant is guaranteed to be established in s_{final}, the invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_{final}.

(R = B-malign): Analogous to Case (R = A-malign) but with inverted sides.

(R = A-malign–B-malign): Combination of Case (R = A-malign) and Case (R = B-malign).

(R = A-benign–B-malign): Combination of Case (R = A-benign) and Case (R = B-malign).

(R = A-malign–B-benign): Combination of Case (R = A-malign) and Case (R = B-benign).

Case 10 (Entity element method): For the execution of an entity element method r on an entity element instance o, we assume that the invariants $\mathbb{X}_{(o,r)}$ hold in r's initial state and show that consequences (1) and (2) hold for k transitive routine invocations made during r's execution. The proof is in parts identical to the proof of Case 1. Furthermore,

the proof is in parts analogous to the one of a free non-interposed relationship element method (Case 9) since the two methods only differ in the framework parameter $\mathbb{C}$, but have analogous instantiations for the remaining framework parameters. Therefore, we only prove the parts that are specific to the execution of an entity element method and refer to the proof of Case 1 for the generic parts of the proof and to the proof of Case 9 for the parts of the proof that are analogous to the one of a free non-interposed relationship element method. We consider an entity element method of an entity E:

(1): To prove (1), we proceed by secondary weak induction on the index i of semi-visible states of r's execution:

(i = 0): See proof of Case 1.

(i = n + 1): Let n be an arbitrary natural number. By inner induction, we can assume that (1) holds in the n-th semi-visible state s_n and we must show that it holds in state s_{n+1}. There are two cases to be considered:

(n + 1 is odd): Control remains within r between s_n and s_{n+1}.

(a) See proof of Case 1.

(b) If s_{n+1} is a pre-state of a direct routine invocation r' on an instance o', then, according to parameter $\mathbb{C}$ (see Table 4.5 on page 85), we know that the callee o' must be any of the below mentioned instances. For some of the cases, the proofs are analogous to the corresponding ones of a free non-interposed relationship element method (Case 9) since they only consider the framework parameters $\mathbb{X}$ and $\mathbb{B}$ as well as the concept of an invalidator (parameters $\mathbb{U}$ and $\mathbb{D}$). As pointed out previously, the two methods differ in the framework parameter $\mathbb{C}$, but have analogous instantiations for the remaining framework parameters.

($o' = o$): Analogous to proof of Case 9, Subcase (R = free).

(o' = instance of mold type): See proof of Case 1.

(n + 1 is even): See proof of Case 1.

(2): This proof is analogous to the one of a free non-interposed relationship element method (Case 9, Subcase (R = free)) since the proof only considers the framework parameters $\mathbb{E}$ and $\mathbb{X}$ as well as the concepts of an invalidator (parameters $\mathbb{U}$ and $\mathbb{D}$) and guarantor (parameter $\mathbb{E}$). As pointed out previously, the two methods differ in the framework parameter $\mathbb{C}$, but have analogous instantiations for the remaining framework parameters.

Case 11 (Interposed relationship element method): For the execution of an interposed relationship element method r on a role element instance o, we assume that the invariants $\mathbb{X}_{(o,r)}$ hold in r's initial state and show that consequences (1) and (2) hold for k transitive routine invocations made during r's execution. The proof is in parts identical to the proof of Case 1. Therefore, we only prove the parts that are specific to the execution of an interposed relationship element method and refer to the proof of Case 1 otherwise. We

consider an interposed relationship element method of a relationship R with the participant base types A and B and the role identifiers a and b, respectively, where the method is interposed into the role R.a:

(1): To prove (1), we proceed by secondary weak induction on the index *i* of semi-visible states of *r*'s execution:

(i = 0): See proof of Case 1.

(i = n + 1): Let *n* be an arbitrary natural number. By inner induction, we can assume that (1) holds in the *n*-th semi-visible state s_n and we must show that it holds in state s_{n+1}. There are two cases to be considered:

(n + 1 is odd): Control remains within *r* between s_n and s_{n+1}.

(a) See proof of Case 1.

(b) If s_{n+1} is a pre-state of a direct routine invocation r' on an instance o', then, according to parameter $\mathbb{C}$ (see Table 4.5 on page 85), we know that the callee o' must be any of the following instances:

($o' = o$): The invoked routine r' must be an interposed relationship element method. According to Table 5.12, this routine expects the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant. Since $o' = o$ and since both the called routine and the calling routine are of the same kind, the expected invariants $\mathbb{X}_{(o',r')}$ are exactly the invariants $\mathbb{X}_{(o,r)}$. The invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_0 by assumption. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the following invariants: the invariants of all relationship element instances O'' of R that have the role element receiver instance o as a participant as well as the invariant of the relationship extent instance in which O'' reside. Thus, we are guaranteed that the invariants $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

(o' = instance of base type A, but not mold instance): The base type of the callee o' must be a direct participant of the base type of the caller o. By Proposition 5.3 and Definition 5.2 it thus holds that $BaseType(o') \neq BaseType(o)$. As shown previously in part (1) (a), we know further that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{n+1}. According to Proposition 5.15, these are the following invariants: the invariants of all relationship element instances O'' of R that have the role element receiver instance o as a participant as well as the invariant of the relationship extent instance in which O'' reside. Since the invariants $\mathbb{X}_{(o,r)}$ are known to hold in s_0 by assumption, we thus know that the invariants $\mathbb{X}_{o,r} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\}$ hold

in s_{n+1}. Since $BaseType(o') \neq BaseType(o)$ and since the invariants $\mathbb{X}_{o,r} \setminus \{inv_{o''} \mid BaseType(o'') = BaseType(o)\}$ hold in s_{n+1}, we conclude by Lemma 5.5 that the invariants $\mathbb{X}_{o',r'}$ hold in s_{n+1}.

(o' = instance of base type B, but not mold instance): Identical to previous case.

(o' = instance of mold type): See proof of Case 1.

(n + 1 is even): See proof of Case 1.

(2): As shown previously in part (1) (a), we know that at most the invariants of those instances for which o is an invalidator are lost between the semi-visible states s_0 and s_{final}. According to Proposition 5.15, these are the following invariants: the invariants of all relationship element instances O'' of R that have the role element receiver instance o as a participant as well as the invariant of the relationship extent instance in which O'' reside. The proof obligation $\mathbb{E}_{(o,r)}$ (see Table 5.12) ensures that the invariants of all relationship element instances O'' as well as the invariant of the relationship extent instance in which O'' resides are preserved between s_0 and s_{final}. Thus, the invariants of all instances for which o is an invalidator are guaranteed to be preserved between s_0 and s_{final}. According to Proposition 5.27, the role element receiver instance o is not a guarantor for any instance, and, thus, the invariants of all instances for which o is a guarantor are trivially guaranteed to be established in s_{final}. According to Table 5.12, the invariants of all extent and element instances of entities or relationships T (with T $\neq$ R) of which R is not a direct or transitive participant are expected to hold in the visible states of r's execution. By assumption, we know that the invariants $\mathbb{X}_{(o,r)}$ hold in s_0. Since none of these expected invariants are lost by r, the invariants $\mathbb{X}_{(o,r)}$ are guaranteed to hold in s_{final}.

□

6 Ownership in the Rumer context

To enlarge the range of verifiable invariants, we extend our verification technique with *selective ownership*. In this chapter, we first introduce selective ownership (Section 6.2) and then present an extended version of our verification technique that also accommodates invariants that depend on locations of owned instances (Section 6.3). We conclude the chapter with an overview of related work on ownership and object-oriented verification techniques (Section 6.4).

6.1 Rationale

The Composite program in Figure 3.11 on page 57 gives rise to an interesting, but challenging invariant: the element invariant of relationship `Composite` (line 30) constrains not only `Composite` relationship element instances but also the `Parent` extent instances related by those relationship element instances. In line with our invariant categorization introduced in [71], we refer to this kind of invariant as an *inter-relationship* invariant as it relates instances of different relationship types. As pointed out in Section 4.3.2 on page 81, these kinds of invariants do not comply with the admissibility criterion specified by the Matryoshka Principle (see Table 4.4 on page 82) and can thus neither be specified nor verified using the verification technique introduced in Chapter 5.

To facilitate the verification of inter-relationship invariants, we extend our verification technique with *selective ownership*. The idea to utilize object ownership to mitigate the adverse affects of aliasing for program understanding [106, 107] has a long-standing tradition. Ownership Types [93, 94, 95, 97] and Universe Types [12, 96, 98, 99, 100] have incorporated those ideas into a programming language's type system to grant a designated owner object unique access to other objects. Object ownership has not only proved viable for easing program understanding but also for thread synchronization [108], memory management [109], and program verification [12, 14, 16, 110]. The latter has been shown for the verification of multi-object invariants in the context of the Universe Type system [12, 14, 16] as well as in the context of a variant of an ownership type system [110].

Verification techniques that are based on Universe Types [12, 14, 16] leverage the fact that Universe Types both impose a tree topology on a program's heap and enforce encapsula-

tion of owned objects [98]. The ownership technique [12, 16], in particular, exploits the following guarantees made by the Universe Type system for program verification: that *(i)* invariants only depend on locations of the invariant-declaring object or of the owned object, that *(ii)* modifications of an owned object's fields are initiated by the object's owner ("owner-as-modifier discipline" [96]), and that *(iii)* owned objects can only invoke impure methods on objects with the same owner or on objects they own themselves. The first and the second property give an owner whose invariant relies on an owned object's fields the chance to re-establish the invariant upon modifications of owned objects. The third property prevents call-backs into owners from within their owned objects or any objects reachable from them.

The guarantees made by the Matryoshka Principle are similar to the ones made by the Universe Type system: that *(i)* invariants only depend on locations of the invariant-declaring type and that *(ii)* there are no transitive call-backs. A comparison with the guarantees made by the Universe Type system listed above reveals that the Matryoshka Principle does not support the notion of a "unique owner" and, hence, does not facilitate the declaration of invariants that depend on an owned instance's locations. As detailed in Section 4.4.2 on page 87, the Matryoshka Principle is less stringent than the Universe Type system and neither imposes a tree topology on a program's heap nor forbids entity or relationship instances to change their relationship participation.

In this chapter, we show how we can leverage the guarantees made by the Matryoshka Principle and superimpose the notion of a "unique owner" on top of the Matryoshka Principle. *Selective ownership* allows a programmer to selectively inject ownership into a program's heap. More specifically, it allows a programmer to make a designated relationship instance the unique owner of any of its participating instances. Making a relationship instance the owner of any of its participating instances allows the invariant of the relationship instance to depend on any of the locations of the owned participating instances. Since modifications of the owned participating instances' locations are guaranteed to be initiated by the unique owner, the owning relationship instance is given a chance to re-establish its invariant upon modifications of the owned participant instances. Moreover, the owning relationship instance is guaranteed not to be encountered in inconsistent states since transitive call-backs are prevented by the underlying Matryoshka Principle. We call this form of ownership *selective* because it is more specific in scope than the ownership established by ownership type systems. Selective ownership only imposes unique ownership on the respective participant of the owning relationship, but not on any transitive participants of the relationship, nor on any other existing instances in the program heap. As a result, selective ownership facilitates a scheme in which *owned* and *shared* instances coexist. In this scheme, the parts of the heap that are unaffected by ownership are not subject to the single ownership restriction, and entity and relationship instances in those parts enjoy the possibility to participate in several relationship instances and to change their relationship participation.

6.2 Selective ownership

In this section, we introduce *selective ownership*. We first illustrate the general idea underlying selective ownership based on an example and then discuss the extensions necessary to the Rumer language introduced earlier to incorporate selective ownership as well as the effects of selective ownership on program executions. The current Rumer prototype compiler does not support selective ownership.

6.2.1 General idea

We illustrate the general idea underlying selective ownership on the Rumer Composite program. Figure 6.1 shows the Rumer Composite program introduced in Figure 3.5 on page 45 extended with selective ownership declarations. As depicted by Figure 3.7 on page 48, we represent a composite in Rumer by an instance of relationship `Composite`, which has the root component of the composite as its `root` participant and a `Parent` extent instance denoting the hierarchical structure of the composite as its `tree` participant. The extent method `createComposite()` on line 5 instantiates a new `Composite` element instance and adds it to its current extent receiver instance. The element method `appendComposite()` on line 10 allows a programmer to append a composite to the current receiver composite at any point in the composite tree. The to-be-inserted `Composite` element instance `c` as well as the point `p` in the tree are passed as arguments to the method. Method `appendComposite()` relies on the element methods `appendComponent()` and `appendSubComposite()` (see line 13 and line 16, resp.) for its implementation, which rely in turn on `Parent`'s extent method `append()` (see line 26). We discuss those methods in more detail in Section 7.1.2 on page 193. The `Composite`'s element invariant declared on line 18 restricts a `Composite` element instance's tree `Parent` extent instance from a forest of trees to a tree (see Section 3.2.5 on page 55 for further details). Since this invariant depends on locations of a `Composite` element instance's tree `Parent` extent instance, it is not admissible according to Table 4.4 on page 82.

To facilitate the verification of a `Composite`'s element invariant, we make use of selective ownership and make a `Composite` element instance become the unique owner of its tree `Parent` extent instance. Selective ownership uses type *modifiers* to express ownership of an instance relative to a current receiver instance. The use of type modifiers (rather than type parameters) results in an ownership type system that is similarly lightweight as the Universe Type system [12, 96, 98, 99, 100]. The `participants` clause of the `Composite` relationship uses the `owned` modifier to indicate that a `Composite` element instance owns the `Parent` extent instance that the element instance relates. In Figure 6.1, for example, a new `Composite` element instance with an owned `Parent` extent instance is instantiated on line 8. A `Composite` element instance's ownership of its tree `Parent` extent instance guarantees that only the `Composite` element instance itself can invoke methods on its tree `Parent` extent instance. For example, the invo-

```
1  // A Composite element instance owns its tree Parent extent instance.
2  relationship Composite
3    participants (Component root, owned Extent<Parent> tree) {
4
5    extent void createComposite(Component c)
6      // New Composite element instance becomes owner of
7      // new Parent extent instance.
8    { these.add(new Composite(c, new owned Extent<Parent>())); }
9
10   void appendComposite(Composite c, Component p) {
11   { this.appendComponent(c.root, p); this.appendSubComposite(c.tree, c.root); }
12
13   void appendComponent(Component c, Component p)
14   { this.tree.append(c, p); } // Invocation admissible: this is owner of tree.
15
16   void appendSubComposite(query Set<Parent> c, Component p) { ... }
17
18   invariant // Invariant admissible: this is owner of tree.
19     !(this.root isElementOf this.tree.child) &
20     (!this.tree.isEmpty() => this.root isElementOf this.tree.parent) &
21     this.tree.transitiveClosure().select(cp: cp.parent == this.root).child ==
22       this.tree.child;
23 }
24
25 relationship Parent participants (Component child, Component parent) {
26   extent void append(Component c, Component p) { ... }
27   ...
28 }
```

Figure 6.1: Rumer Composite program extending the versions shown in Figure 3.5 on page 45 and Figure 3.11 on page 57 with selective ownership. The element invariant declared on line 18 is admissible since a `Composite` element instance is the owner of the `Parent` extent instance related by the `Composite` element instance. See complete Composite pattern specification in Section 7.1.2 on page 193.

cation of the extent method `append()` on the `Parent` extent instance referred to by `this.tree` on line 14 is ownership-admissible since `this` is the owner of the extent instance. Thanks to a `Composite` element instance's ownership of its `Parent` extent instance, the element invariant of relationship `Composite` becomes admissible. Since modifications of the `Parent` extent instance's locations are guaranteed to be initiated by the owning `Composite` element instance, the `Composite` element instance is given a chance to re-establish its invariant upon any such modifications.

To reduce the annotation burden put on the programmer, we use default modifiers if no modifiers are given by the programmer. Since selective ownership permits a hybrid ownership scheme in which owned and shared instances coexist, the default modifier is `shared`. For the Composite program shown in Figure 6.1 this practice consequently means that both the `Parent` element instances residing in a `Composite` element instance's tree `Parent` extent instance and the `Component` element instances related by those `Parent` element instances are shared. As a result, those instances can freely participate in several relationship instances (of possibly different relationship types) and change their relationship participation. Furthermore, modifications of those instances can

be initiated by any of the relationship instances in which they participate.

To permit a hybrid ownership scheme, a type declaration has to be formulated so as to leave the decision of whether a particular instance of this type is shared or owned to the caller-site of the type. For example, relationship `Parent` in Figure 6.1 uses default modifiers for its type declarations, and only the relationship `Composite` indicates that its element instances will become the owners of their tree `Parent` extent instances. Since ownership is only fixed at the caller-site, but left open at the callee-site, this scheme gives rise to situations in which owned instances are perceived as "supposedly shared" instances. In Section 6.2.3 we discuss what precautions are taken to prevent accidental leaking of "supposedly shared" instances.

The `Composite` relationship declared in Figure 6.1 only imposes ownership on its tree `Parent` extent instance. Alternatively, it is possible to impose ownership not only on an extent instance but also on all the element instances that reside in the extent instance. For the example of the Composite program shown in Figure 6.1, extent and element ownership can be achieved by the following declaration:

```
relationship Composite participants (Component root, owned Extent<owned Parent>)
```

This declaration indicates that a `Composite` element instance owns its `Parent` extent instance as well as all the `Parent` element instances contained in the `Parent` extent instance. We refer to the first form of ownership that targets only an extent instance, but excludes the element instances contained in the extent instance, as *exclusive* ownership, and we refer to the second form of ownership that targets an extent as well as all the element instances contained in the extent instance as *inclusive* ownership. Whereas it is possible to impose ownership on an extent instance without imposing ownership on the element instances residing in the extent instance, it is not possible to impose ownership on element instances without imposing ownership on the extent instance in which the element instances reside. This restriction is due to the fact that fields of element instances can be written to by any extent method that executes on the extent instance in which the element instances reside (see parameter $\mathbb{U}$ in Table 4.3 on page 80).

6.2.2 Language extensions

In this section, we discuss the necessary extensions of the Rumer language to support selective ownership. Selective ownership gives rise to a number of system invariants upon which the extended verification technique relies. We first introduce those system invariants and then sketch how to extend the Rumer type system introduced earlier.

System invariants

Selective ownership superimposes the notion of a "unique owner" on top of the Matryoshka Principle. By leveraging the guarantees made by the principle, selective ownership allows for a more localized expression of ownership, permitting a scheme in which

shared and owned instances coexist. Whether an instance is owned or shared is determined at instantiation time. If an instance has an owner, then this owner remains the same for the instance's entire lifetime. We can capture this aspect of selective ownership by the following system invariant:

System Invariant 6.1 (Selective ownership): Consider a type whose instances can have an owner and an instance t of this type. Selective ownership allows t either to have an owner or not to have an owner at all. If t has an owner, this owner is unique, determined at t's instantiation time, and remains unchanged for t's entire lifetime.

To allow selective ownership to benefit from the guarantees made by the Matryoshka Principle and from the absence of transitive call-backs, in particular, the ownership relation must be a subset of a program's participants relation. This requirement gives rise to the following system invariant:

System Invariant 6.2 (Ownership relation is subset of participants relation): Consider an instance t. The instance t can have an instance u as its owner if and only if t is not a mold instance, u is neither a mold instance nor a role element instance, and $BaseType(t)$ is a direct participant of $BaseType(u)$.

System Invariant 6.2 also indicates that owned instances can be entity instances as well as relationship instances and that owners can be relationship instances as well as the application singleton instance. A programmer can selectively impose ownership on an instance by using *selective ownership modifiers*. These modifiers indicate ownership of an instance relative to the current receiver instance. We distinguish the following selective ownership modifiers:

- **owned:** referred-to instance has the current receiver instance as its owner;
- **shared (default modifier):** referred-to instance does not have an owner;
- **readonly:** referred-to instance may or may not have an owner.

Selective ownership modifiers can appear as part of element type declarations and extent type declarations, permitting ownership to be imposed on element instances (including role element instances in the case of a relationship element instance) and extent instances. Owners, on the other hand, can be relationship element instances (excluding role element instances), relationship extent instances, and the application singleton instance. Ownership for role element instances is not supported since role element instances cannot declare any invariants. If a type declaration omits the selective ownership modifier, then the default modifier shared is assumed.

The following system invariant guarantees that element ownership is always combined with extent ownership and consequently prevents shared extent instances from modifying the locations of owned element instances that reside in those extent instances.

System Invariant 6.3 (Element ownership implies extent ownership): Consider an instance o that has an owner. If o is an entity element instance or a relationship element

instance, then the extent instance in which o resides is owned too and has the same owner as o.

Since role element instances are implicitly instantiated along with relationship element instantiation (see Section 3.4.1 on page 65), System Invariant 6.3 also entails the following system invariant:

System Invariant 6.4 (Role element ownership iff relationship element ownership): Consider an instance o that has an owner. If o is a relationship element instance, then the role element instances related by o are owned too and have the same owner as o. If o is a role element instance, then the relationship element instances that relate o are owned too and have the same owner as o.

Selective ownership modifiers constrain routine invocations and thus override the restrictions imposed on routine invocations by the Matryoshka Principle (i.e., parameter $\mathbb{C}$, see Table 4.5 on page 85[1]). In particular, selective ownership only allows owners to invoke routines on owned instances. This regime guarantees that modifications of an owned instance's fields are initiated by the instance's owner and gives rise to a discipline which is referred to in the context of Universe Type systems as the "owner-as-modifier discipline" [96]. In Section 6.2.3, we revisit this property and introduce the idiom "owned-called-by-owner" to contrast the discipline arising from selective ownership with the "owner-as-modifier" discipline. Routine invocations on shared instances, on the other hand, are not constrained by selective ownership. Furthermore, selective ownership permits the reading of fields and query members of owned instances and the application of built-in query operators to owned instances. This aspect of selective ownership can be captured by the following system invariant:

System Invariant 6.5 (Effects of selective ownership modifiers): Consider a current receiver instance o that keeps a reference to an instance o'. If the selective ownership modifier of the reference is shared, then o can only invoke a routine on o' if o' is an admissible callee according to parameter $\mathbb{C}$. If the selective ownership modifier of the reference is owned, then o can only invoke a routine on o' if o' is an admissible callee according to parameter $\mathbb{C}$ and if o is the owner of o'. If the selective ownership modifier of the reference is readonly, then o cannot invoke any routines on o'.

Type system

We next sketch how we extend the Rumer type system introduced in Section 3.3 on page 59 to incorporate selective ownership. We only discuss the most important aspects of this extension since a complete and formal description thereof is beyond the scope of this thesis. Figure 6.2 shows the resulting extension of Rumer type system introduced in Figure 3.12 on page 60 that accommodates selective ownership modifiers. As detailed by

[1] In Section 6.2.3, Table 6.7, we provide an updated account of parameter $\mathbb{C}$ which takes precautions against accidental leaking of "supposedly shared" instances.

SingletonType	≙	ApplicationName
NominalType	≙	EntityName | RelationshipName | RelationshipName.RoleName
ReferenceType	≙	ElementType | ExtentType
ElementType	≙	**modifier** Element⟨NominalType⟩ | **modifier** Any
ExtentType	≙	**modifier** Extent⟨**modifier** Element⟨EntityName⟩⟩ | **modifier** Extent⟨**modifier** Element⟨RelationshipName⟩⟩
modifier	≙	**shared** | **owned**$_{\textbf{owner}}$ | **readonly**
owner	≙	**Elm** | **Ext**
ValueType	≙	SetType | PairType | BagType | ItemType
SetType	≙	Set⟨CoordinateType⟩
PairType	≙	Pair⟨CoordinateType, CoordinateType⟩
CoordinateType	≙	ReferenceType | SetType | PairType
BagType	≙	Bag⟨ItemType⟩ | Bag⟨BagType⟩
ItemType	≙	MoldType | PrimitiveType
MoldType	≙	Mold⟨NominalType⟩
PrimitiveType	≙	boolean | string | int | float

Figure 6.2: Rumer internal types extended with selective ownership modifiers (new constructs in comparison to Figure 3.12 are highlighted in **bold**). The shared modifier is the default modifier.

Figure 6.2, the selective ownership modifiers of the type system basically correspond to the selective ownership modifiers that can appear in the source text. To keep track of the respective kind of owner of an owned instance, however, the owned modifier additionally uses a subscript. An owner that is an element instance is denoted by the subscript Elm and an owner that is an extent instance is denoted by the subscript Ext. The internal type system thus distinguishes the following selective ownership modifiers: owned$_{\text{Elm}}$, owned$_{\text{Ext}}$, shared, and readonly.

To incorporate selective ownership into the subtyping relation introduced in Figure 3.13 on page 64, we define an ordering relation on the selective ownership modifiers. Figure 6.3 shows the resulting ordering relation. The relation indicates that the ordering on selective ownership modifiers is reflexive (ORD-REFLX) and that both the owned and shared modifiers are smaller (i.e., more precise) than the readonly modifier (ORD-OWNED-READONLY and ORD-SHARED-READONLY, resp.), leaving the readonly modifier as the least specific selective ownership modifier. Figure 6.4 details the selective-ownership-aware subtyping relation for Rumer types. As indicated by rule SUBSELOWNR-GEN, the selective-ownership-aware subtyping relation is generally determined by the subtyping relation defined in Figure 3.13 on page 64 and by the ordering relation on the selective ownership modifiers defined in Figure 6.3. For example, the rule SUB-ROLE1 in Figure 3.13 and the rule SUBSELOWNR-GEN in Figure 6.4 indicate for a relationship R with the owned participant base type A a that the owned role Element⟨R.a⟩ conforms to the owned participant supertype Element⟨A⟩. However, for role types and extent types the rules SUBSELOWNR-ROLE and SUBSELOWNR-EXTENT additionally apply. For example, if a `Composite` relationship element instance declared inclusive ownership of its `Composite` element instance's tree `Parent` extent instance (see Figure 6.1), then the rule SUBSELOWNR-ROLE would guarantee that the owned roles Element⟨Parent.child⟩ and Element⟨Parent.parent⟩ conform to the shared participant

$$\mathsf{m} \leqslant \mathsf{m} \qquad \text{(ORD-REFLX)}$$

$$\mathsf{owned}_{\mathsf{owner}} \leqslant \mathsf{readonly} \qquad \text{(ORD-OWNED-READONLY)}$$

$$\mathsf{shared} \leqslant \mathsf{readonly} \qquad \text{(ORD-SHARED-READONLY)}$$

Figure 6.3: Ordering relation on selective ownership modifiers. The meta-variables m and owner range over the modifier names and owner names, resp., defined in Figure 6.2.

$$\frac{\mathsf{m} \leqslant \mathsf{n} \qquad \mathsf{S} <: \mathsf{T}}{\mathsf{m\ S} \overset{selOwnr}{<:} \mathsf{n\ T}} \qquad \text{(SUBSELOWNR-GEN)}$$

$$\frac{\mathsf{owned}_{\mathsf{owner}}\ \mathsf{Element}\langle \mathsf{R.\_}\rangle \qquad \mathsf{shared\ T} \qquad \mathsf{Element}\langle \mathsf{R.\_}\rangle <: \mathsf{T}}{\mathsf{owned}_{\mathsf{owner}}\ \mathsf{Element}\langle \mathsf{R.\_}\rangle \overset{selOwnr}{<:} \mathsf{shared\ T}} \qquad \text{(SUBSELOWNR-ROLE)}$$

$$\mathsf{m\ Extent}\langle \mathsf{n\ S}\rangle \overset{selOwnr}{<:} \mathsf{Set}\langle \mathsf{n\ S}\rangle \qquad \text{(SUBSELOWNR-EXTENT)}$$

Figure 6.4: Selective-ownership-aware subtyping relation. The meta-variables m and n as well as owner range over the modifier names and owner names, resp., defined in Figure 6.2. The meta-variable R ranges over relationship names, and the meta-variable S and T range over types. See Figure 6.3 for relation "$\leqslant$" and Figure 3.13 on page 64 for relation "$<:$".

$\mathbf{m}_1 \triangleright \mathbf{m}_2$

	$\mathbf{m}_2$		
$\mathbf{m}_1$	$\mathsf{owned}_{\mathsf{ownr}}$	shared	readonly
$\mathsf{owned}_{\mathsf{ownr}}$	readonly	shared	readonly
shared	readonly	shared	readonly
readonly	readonly	shared	readonly

Table 6.5: Viewpoint adaptation for selective ownership modifiers.

supertype Element⟨Component⟩. The rule SUBSELOWNR-EXTENT, on the other hand, guarantees that an exclusively owned extent instance conforms to any of its shared subsets.

Since selective ownership modifiers are stated relative to a current receiver instance, the resulting modifier of a reference chain is determined by the modifiers of the individual references. Similarly to the Universe Type system, we define a *viewpoint adaptation* operation[99] that indicates the resulting selective ownership modifier for a compound access of two instances. Table 6.5 shows the resulting viewpoint adaptation for the selective ownership modifiers introduced in this section. It indicates, for example, that the chained access of a shared reference through an owned reference remains shared.

6.2.3 Semantics

In this section, we describe what the effects of selective ownership are on program executions. We first discuss necessary precautions to prevent accidental leaking of "supposedly shared" instances and then capture the "owned-called-by-owner" discipline.

Precautions

Selective ownership requires type declarations to be formulated so as to allow possible callers to obtain either owned or shared instances of the type. As a result, only the caller-site knows about the ownership of an instance and can thus guard the instance against prohibited modifications or against accidental leaking by establishing appropriate selective ownership modifiers. The callee-site, on the other hand, does not know whether a particular instance is owned or shared and thus may perceive an owned instance as "supposedly shared". To guarantee soundness of our hybrid ownership system, we take precautions to prevent the accidental leaking of such "supposedly shared" instances. Before discussing the necessary precautions, we introduce the notion of an *inaccurate reference* to denote a reference that points to a "supposedly shared" instance:

Definition 6.6 (Inaccurate reference): Consider a reference to an instance o with the selective ownership modifier m. The reference is inaccurate, if o has an owner, but the modifier m is shared. Otherwise, the reference is accurate.

Similarly, we define the notions of an *inaccurate field* and *inaccurate query*:

Definition 6.7 (Inaccurate field): Consider an instance's field with the selective ownership modifier m that stores a reference to an instance o. The field is inaccurate, if o has an owner, but the field's modifier m is shared. Otherwise, the field is accurate.

Definition 6.8 (Inaccurate query): Consider an instance's query with the selective ownership modifier m that returns a reference to an instance o. The query is inaccurate, if o has an owner, but the query's modifier m is shared. Otherwise, the query is accurate.

Given the definition of an inaccurate reference, we can restate our goal more precisely as: to prevent an instance from leaking an inaccurate reference to an instance o to an instance o' that can invoke routines on o according to System Invariant 6.5 and that is not the owner of o. Due to the caller-site-controlled ownership scheme, an inaccurate reference to the owned instance o can only be generated from within any (possibly transitive) callee o'' of the owner of o. The only way how such a callee o'' can leak any inaccurate reference to o is either by passing the reference on to a (possibly transitive) callee of itself or by storing the reference in an instance's inaccurate field. Furthermore, if the callee $o'' = o$, then o can leak an inaccurate reference to o to o' by declaring an inaccurate query that returns o. According to the restrictions imposed on routine invocations by the Matryoshka Principle (i.e., parameter $\mathbb{C}$, see Table 4.5 on page 85) and by acyclicity of the participants relation (see System Invariant 4.1 on page 77), it is guaranteed that any (possibly transitive) callee of o'' is not allowed to invoke a routine on o. Thus, in order to guarantee soundness of our hybrid ownership system, the only precautions left to be taken are: to prevent the callee o'' from storing a reference to o in an instance's inaccurate field and to prevent the owned instance o from declaring an inaccurate query that returns o.

To prevent any (possibly transitive) callee o'' of an owner of an instance o from storing a

Instance	Illegal field type
Relationship R with participant base types A a and B b	
Extent⟨Element⟨R⟩⟩ *rx*	For any relationship T with participant base types C c and D d such that T ≠ R and R is a direct or transitive participant of T: Extent⟨Element⟨T⟩⟩, Element⟨T⟩, Element⟨T.c⟩, and Element⟨T.d⟩.
Element⟨R⟩ *r*	Extent⟨Element⟨R⟩⟩, Element⟨R⟩, Element⟨R.a⟩, and Element⟨R.b⟩.
	For any relationship T with participant base types C c and D d such that T ≠ R and R is a direct or transitive participant of T: Extent⟨Element⟨T⟩⟩, Element⟨T⟩, Element⟨T.c⟩, and Element⟨T.d⟩.
Mold⟨R⟩ *mr*	*Defined by Element⟨R⟩*
Element⟨R._⟩ *rab*	Extent⟨Element⟨R⟩⟩, Element⟨R⟩, Element⟨R.a⟩, and Element⟨R.b⟩.
	For any relationship T with participant base types C c and D d such that T ≠ R and R is a direct or transitive participant of T: Extent⟨Element⟨T⟩⟩, Element⟨T⟩, Element⟨T.c⟩, and Element⟨T.d⟩.
Entity E	
Extent⟨Element⟨E⟩⟩ *ex*	For any relationship T with participant base types C c and D d such that E is a direct or transitive participant of T: Extent⟨Element⟨T⟩⟩, Element⟨T⟩, Element⟨T.c⟩, and Element⟨T.d⟩.
Element⟨E⟩ *e*	Extent⟨Element⟨E⟩⟩ and Element⟨E⟩.
	For any relationship T with participant base types C c and D d such that E is a direct or transitive participant of T: Extent⟨Element⟨T⟩⟩, Element⟨T⟩, Element⟨T.c⟩, and Element⟨T.d⟩.
Mold⟨E⟩ *me*	*Defined by Element⟨E⟩*

Table 6.6: Types of which an instance cannot declare fields.

reference to o in an instance's inaccurate field, we impose restrictions on *(i)* the admissible types for field declarations and on *(ii)* the admissible receivers of mold constructor and mold initializer invocations. The first restriction *(i)* removes some of the liberalness discussed in Section 4.4 on page 86 and requires that field declarations do not "point to" instances of a base type of which the declaring base type is a direct or transitive participant. This restriction is listed in Table 6.6. Table 6.6 indicates for an instance of a particular type of which types the instance is not allowed to declare any fields. The second restriction *(ii)* removes some of the liberalness with regard to invocations of mold constructors and mold initializers and require those invocations to propagate — like any other routine invocations — in the direction of a program's participants relation. Without the second restriction *(ii)*, the callee o'' could invoke a routine on a mold instance of a base type of which $\mathit{BaseType}(o)$ is direct or transitive participant and thus store an inaccurate reference to the owned instance o in the mold instance's field. To realize the second restriction *(ii)*, we update the existing restrictions on routine invocations imposed by the Matryoshka Principle (i.e., parameter $\mathbb{C}$, see Table 4.5 on page 85). Table 6.7 shows the new instantiation of parameter $\mathbb{C}$, on which our extended verification technique discussed in Section 6.3 relies. As compared to its previous instantiation shown in Table 4.5, the new instantiation of parameter $\mathbb{C}$ omits the rows that accept an arbitrary mold instance as

Executing routine	Allowed callee
Application App	
action	Current receiver instance of type App.
	For any relationship or entity T, any instance o such that $BaseType(o) = T$.
Relationship R with participant base types A a and B b	
default extent constructor	–
extent constructor	Current receiver instance of type Extent⟨Element⟨R⟩⟩.
	Any instance of type Element⟨R⟩, **Mold⟨R⟩**, Element⟨R.a⟩, Element⟨R.b⟩, or any instance o such that $BaseType(o) = A$ or $BaseType(o) = B$.
extent method	Current receiver instance of type Extent⟨Element⟨R⟩⟩.
	Any instance of type Element⟨R⟩, **Mold⟨R⟩**, Element⟨R.a⟩, Element⟨R.b⟩, or any instance o such that $BaseType(o) = A$ or $BaseType(o) = B$.
non-interposed mold constructor	–
non-interposed mold initializer	–
non-interposed element method	Current receiver instance of type Element⟨R⟩.
	Any instance of type Element⟨R.a⟩ or Element⟨R.b⟩, or any instance o such that $BaseType(o) = A$ or $BaseType(o) = B$.
interposed element method	Current receiver instance of type Element⟨R._⟩.
	Any instance o such that $BaseType(o) = A$ or $BaseType(o) = B$.
Entity E	
default extent constructor	–
extent constructor	Current receiver instance of type Extent⟨Element⟨E⟩⟩.
	Any instance of type Element⟨E⟩ or **Mold⟨E⟩**.
extent method	Current receiver instance of type Extent⟨Element⟨E⟩⟩.
	Any instance of type Element⟨E⟩ or **Mold⟨E⟩**.
mold constructor	–
mold initializer	–
element method	Current receiver instance of type Element⟨E⟩.

Table 6.7: Receivers on which executing routine is allowed to invoke routines (parameter ℂ). This table extends Table 4.5 on page 85 with **selective ownership**, differences are highlighted in **bold**.

an admissible callee.

To prevent the owned instance o from declaring an inaccurate query that returns o, we restrict the admissible types for query declarations. The resulting restriction is listed in Table 6.8. Table 6.8 indicates for an instance of a particular type of which types the instance is not allowed to declare any queries. Except for extent instances, this restriction prevents an instance from declaring a query that redirects to the declaring instance.

Instance	Illegal query type
Relationship R with participant base types A a and B b	
Element⟨R⟩ *r*	Extent⟨Element⟨R⟩⟩, Element⟨R⟩, Element⟨R.a⟩, and Element⟨R.b⟩.
Mold⟨R⟩ *mr*	*Defined by Element⟨R⟩*
Element⟨R._⟩ *rab*	Extent⟨Element⟨R⟩⟩, Element⟨R⟩, Element⟨R.a⟩, and Element⟨R.b⟩.
Entity E	
Element⟨E⟩ *e*	Extent⟨Element⟨E⟩⟩ and Element⟨E⟩.
Mold⟨E⟩ *me*	*Defined by Element⟨E⟩*

Table 6.8: Types of which an instance cannot declare queries.

Given these precautions, we can state an important lemma that precisely proves for any well-formed Rumer call stack and for any routine execution on any callee of a possibly owned instance that the callee does not store any references to the owned instance in inaccurate fields. To state the lemma, we first need to provide an updated definition of the well-formedness condition on a call stack introduced in Definition 5.2 on page 104 to account for the new instantiation of parameter $\mathbb{C}$ (see Table 6.7):

Definition 6.9 (Restatement of "well-formed call stack"): A call stack σ is well-formed if and only if for any consecutive stack frames $(o', r')_l$ and $(o, r)_{l-1}$ contained in σ, where $(o', r')_l$ denotes the callee stack frame and $(o, r)_{l-1}$ denotes the caller stack frame, it either holds that $BaseType(o') = BaseType(o)$ or that $BaseType(o')$ is a direct participant of $BaseType(o)$, according to the participants relation of the program. In the latter case, it holds further that there exists no preceding stack frame $(o'', r'')_k$ such that $0 \leqslant k < l - 1$ and that $BaseType(o'') = BaseType(o')$.

The following corollary is a restatement of Proposition 5.3 on page 104 and guarantees that the execution of a Rumer program only produces well-formed call-stacks:

Corollary 6.10 (Restatement of "Rumer call stacks are well-formed"): Any call stack σ generated by the execution of a Rumer program that is successfully type checked and verified using the extended verification technique is well-formed.

Proof 6.11: Corollary 6.10 is a restatement of Proposition 5.3 on page 104 based on the new definition of a well-formed call stack (see Definition 6.9) and the new instantiation of parameter $\mathbb{C}$ (see Table 6.7). The new instantiation of parameter $\mathbb{C}$ restricts invocations of mold constructors and mold initializers to propagate — like any other routine invocations — in the direction of a program's participants clause. □

We are finally in the position to state the lemma. The lemma relies on the notion of a semi-visible state (see Definition 5.7 on page 105). To state the lemma more concisely, we introduce the notion of "being rooted in an extent instance", which we define as:

Definition 6.12 (Rooted in extent instance): For an extent instance *ox*, the instances

rooted in the extent instance *ox* are:

- if *o* is a relationship extent instance: *ox*, all relationship element instances that reside in *ox*, and all role element instances that are related by relationship element instances residing in *ox*;
- if *o* is an entity extent instance: *ox* and all entity element instances that reside in *ox*.

Lemma 6.13 (Prevention of storing (possibly transitive) callers in callee's fields): For any well-formed call stack σ and for any stack frame (o, r) contained in σ, if *o* is not the application singleton instance, then, for all semi-visible states s_i of *r*'s execution:

(1) for any (possibly transitive) caller o' of *o* such that $o' \neq o$: *r* does not store any references to instances that are rooted in the same extent instance as o' in fields to which *r* can write between the semi-visible states s_0 and s_i;
(2) for the current receiver instance *o*:
   (a) if *o* is an extent instance: *r* stores references to any instances that are rooted in *o* at most in fields of *o* (but in no other fields to which *r* can write) between the semi-visible states s_0 and s_i;
   (b) otherwise: *r* does not store any references to instances that are rooted in the same extent instance as *o* in fields to which *r* can write between the semi-visible states s_0 and s_i.

Proof 6.14: We prove Lemma 6.13 by considering, for the execution of a routine and for the types whose instances must not be stored during the routine's execution, whether the instances to whose fields the routine can write are at all allowed to declare fields of such types. The proof is determined by parameters $\mathbb{U}$ and $\mathbb{C}$ (see Table 4.3 on page 80 and Table 6.7, resp.) as well as by the restrictions on field declarations defined in Table 6.6. We proceed by strong induction on the number *k* of transitive routine invocations made during the execution of a routine *r* on a current receiver instance *o*. Let *k* be an arbitrary natural number of transitive routine invocations. By induction, we can assume that the lemma holds for any natural number *j* of transitive routine invocations such that $j < k$ and we must show that it also holds for *k* transitive routine invocations. We assume that the current receiver instance *o* is not the application singleton instance, giving rise to the following cases:

Case 1 (Default relationship extent constructor): For the execution of a default relationship extent constructor *r* on an instance *o*, we show that Lemma 6.13 holds for *k* transitive routine invocations made during *r*'s execution. We consider a relationship R with the participant base types A and B and the role identifiers a and b, respectively, and proceed by investigating the only two semi-visible states of *r*'s execution:

(Initial state): Immediate since no fields can yet have been written to.

(Final state): According to parameter $\mathbb{U}$ (see Table 4.3 on page 80), *r* can only write to the fields of the current receiver instance *o* between s_0 and s_{final}.

(1) We know that o is a relationship extent instance and that $o' \neq o$, by assumption. According to parameter $\mathbb{C}$ (see Table 6.7), a (possibly transitive) caller o' must then be of a base type T of which R is a direct or transitive participant. In line with Lemma 6.13, we only consider callers that are extent instances, relationship element instances, or role element instances. Thus, T must be a relationship with the participant base types C c and D d, let's say. According to Table 6.6 and since the participants relation is acyclic (see System Invariant 4.1 on page 77), the instances to whose fields r can write between s_0 and s_{final} (see above) are not allowed to declare any fields of the types Extent⟨Element⟨T⟩⟩, Element⟨T⟩, Element⟨T.c⟩, and Element⟨T.d⟩. As a result, r is guaranteed not to store any references to instances that are rooted in the same extent instance as o' in fields to which r can write between the semi-visible states s_0 and s_{final}.

(2) According to Table 6.6, the only instance of those instances to whose fields r can write between s_0 and s_{final} (see above) and that is allowed to declare any fields of the types Extent⟨Element⟨R⟩⟩, Element⟨R⟩, Element⟨R.a⟩, and Element⟨R.b⟩ is the current extent receiver instance o. Since o is empty, r is guaranteed to store at most a reference to o in fields of o (but in no other fields) between s_0 and s_{final}.

Case 2 (Default entity extent constructor): The proof is analogous to the proof of a default relationship extent constructor (see Case 1) since the two constructors have analogous instantiations for parameters $\mathbb{U}$ (see Table 4.3 on page 80) and $\mathbb{C}$ (see Table 6.7) as well as analogous restrictions on field declarations (see Table 6.6) for the writable instances determined by parameter $\mathbb{U}$.

Case 3 (Relationship extent constructor or method): We collate the proofs of relationship extent constructors and relationship extent methods since they share the same set of writable instances (parameter $\mathbb{U}$, see Table 4.3 on page 80), to which the same restrictions on field declarations (see Table 6.6) consequently apply, and since they share the same set of (possibly) transitive callers (parameter $\mathbb{C}$, see Table 6.7). For the execution of such a routine r on an instance o, we show that Lemma 6.13 holds for k transitive routine invocations made during r's execution. We consider a relationship R with the participant base types A and B and the role identifiers a and b, respectively, and proceed by secondary weak induction on the index i of semi-visible states of r's execution:

(i = 0): Base case: immediate since no fields can yet have been written to.

(i = n + 1): Let n be an arbitrary natural number. By inner induction, we can assume that (1) and (2) hold in s_n and we must show that they hold in s_{n+1}. There are two cases to be considered:

(n + 1 is odd): Control remains within r between s_n and s_{n+1}. According to parameter $\mathbb{U}$ (see Table 4.3 on page 80), r can write to the fields of the following instances between s_n and s_{n+1}: o, any relationship element instance residing in o, and any role element instance related by such relationship element instances.

(1) We know that o is a relationship extent instance and that $o' \neq o$, by assumption. According to parameter $\mathbb{C}$ (see Table 6.7), a (possibly transitive) caller o' must then be of a base type T of which R is a direct or transitive participant. In line with Lemma 6.13, we only consider callers that are extent instances, relationship element instances, or role element instances. Thus, T must be a relationship with the participant base types C c and D d, let's say. According to Table 6.6 and since the participants relation is acyclic (see System Invariant 4.1 on page 77), the instances to whose fields r can write between s_n and s_{n+1} (see above) are not allowed to declare any fields of the types Extent⟨Element⟨T⟩⟩, Element⟨T⟩, Element⟨T.c⟩, and Element⟨T.d⟩. As a result, r is guaranteed not to store any references to instances that are rooted in the same extent instance as o' in fields to which r can write between s_n and s_{n+1} and, thus, by inner induction, between s_0 and s_{n+1}.

(2) According to Table 6.6, the only instance of those instances to whose fields r can write between s_n and s_{n+1} (see above) and that is allowed to declare any fields of the types Extent⟨Element⟨R⟩⟩, Element⟨R⟩, Element⟨R.a⟩, and Element⟨R.b⟩ is the current extent receiver instance o. As a result, r is guaranteed to store references to o or to any element instances comprised in o (including role element instances related by such element instances) at most in fields of o (but in no other fields to which r can write) between s_n and s_{n+1} and, thus, by inner induction, between s_0 and s_{n+1}.

($n+1$ is even): A direct routine invocation r'' on an instance o'' is made between s_n and s_{n+1}. By inner induction, we can assume that (1) and (2) hold in s_n and we must show that they hold in s_{n+1}. Since there must be strictly fewer than k routine invocations made during the execution of the routine r'', we know, by outer induction, that Lemma 6.13 holds for the execution of r''. Furthermore, by parameter $\mathbb{C}$ (see Table 6.7) and by acyclicity of the participants relation (see System Invariant 4.1 on page 77), we know that the callee o'' cannot be the singleton application instance and, thus, that (1) and (2) hold for the execution of r''. We consider the following cases:

($o'' = o$): Then, we know by parameters $\mathbb{U}$ (see Table 4.3 on page 80) and $\mathbb{C}$ (see Table 6.7) that r and r'' can write to fields of the same set of instances and that o and o'' share the same set of (possibly) transitive callers, respectively, as well as that o and o'' share the same set of "self-references". As a result, (1) and (2) are also guaranteed to hold for r between s_n and s_{n+1} and, thus, by inner induction, between s_0 and s_{n+1}.

($o'' \neq o \wedge \mathit{BaseType}(o'') = \mathit{BaseType}(o) \wedge o''$ is not mold instance): Then, we know by parameters $\mathbb{C}$ and $\mathbb{U}$ (see Table 6.7 and Table 4.3 on page 80, resp.) that r'' can write to the fields of a subset of those instances to whose fields r can write. However, we know that the set of (possibly) transitive callers of o is a subset of the set of (possibly) transitive callers of o''. As a result, (1) is also guaranteed to hold for r between s_n and s_{n+1} and, thus, by inner induction,

between s_0 and s_{n+1}. Furthermore, we know that o is a caller of o'', and that (1) holds for the execution of r''. As a result, (2) is also guaranteed to hold for r between s_n and s_{n+1} and, thus, by inner induction, between s_0 and s_{n+1}.

(otherwise): Then, we know by parameters $\mathbb{C}$ and $\mathbb{U}$ (see Table 6.7 and Table 4.3 on page 80, resp.) as well as by acyclicity of the participants relation (see System Invariant 4.1 on page 77) that r'' cannot write to the fields of any of those instances to whose fields r can write. As a result, (1) and (2) are also guaranteed to hold for r between s_n and s_{n+1} and, thus, by inner induction, between s_0 and s_{n+1}.

Case 4 (Entity extent constructor or method): The proof is analogous to the proof of a relationship extent constructor or relationship extent method (see Case 3) since entity extent constructors or methods and relationship extent constructors or methods have analogous instantiations for parameters $\mathbb{U}$ (see Table 4.3 on page 80) and $\mathbb{C}$ (see Table 6.7) as well as analogous restrictions on field declarations (see Table 6.6) for the writable instances determined by parameter $\mathbb{U}$.

Case 5 (Non-interposed relationship mold constructor or initializer): We collate the proofs of non-interposed relationship mold constructors and non-interposed relationship mold initializers since they share the same set of writable instances (parameter $\mathbb{U}$, see Table 4.3 on page 80), to which the same restrictions on field declarations (see Table 6.6) consequently apply, and since they share the same set of (possibly) transitive callers (parameter $\mathbb{C}$, see Table 6.7). For the execution of such a routine r on an instance o, we show that Lemma 6.13 holds for k transitive routine invocations made during r's execution. We consider a relationship R with the participant base types A and B and the role identifiers a and b, respectively, and proceed by investigating the only two semi-visible states of r's execution:

(Initial state): Immediate since no fields can yet have been written to.

(Final state): According to parameter $\mathbb{U}$ (see Table 4.3 on page 80), r can write to the fields of the following instances between s_0 and s_{final}: o, $o.a$ and $o.b$.

(1) We know that o is a relationship mold instance and that $o' \neq o$, by assumption (and since mold constructors and initializers cannot invoke other routines). According to parameter $\mathbb{C}$ (see Table 6.7), a (possibly transitive) caller o' must then be any of the below mentioned instances. In line with Lemma 6.13, we only consider callers that are extent instances, relationship element instances, or role element instances and ignore the singleton instance of the application App:

(o' = instance of type Extent⟨Element⟨R⟩⟩): According to Table 6.6, the instances to whose fields r can write between s_0 and s_{final} (see above) are not allowed to declare any fields of the types Extent⟨Element⟨R⟩⟩, Element⟨R⟩, Element⟨R.a⟩, and Element⟨R.b⟩. As a result, r is guaranteed not to store any references to o' nor references to any element instances comprised in o' (including role element instances related by such element instances) in fields to which r can write between s_0 and s_{final}.

(*BaseType*(o') = T ∧ R is direct or transitive participant of T ∧ T ≠ App): T must be a relationship with the participant base types C c and D d, let's say. According to Table 6.6 and since the participants relation is acyclic (see System Invariant 4.1 on page 77), the instances to whose fields r can write between s_0 and s_{final} (see above) are not allowed to declare any fields of the types Extent⟨Element⟨T⟩⟩, Element⟨T⟩, Element⟨T.c⟩, and Element⟨T.d⟩. As a result, r is guaranteed not to store any references to instances that are rooted in the same extent instance as o' in fields to which r can write between the semi-visible states s_0 and s_{final}.

(2) Vacuous since o is a relationship mold instance.

Case 6 (Entity mold constructor or initializer): The proof is analogous to the proof of a non-interposed relationship mold constructor or non-interposed relationship mold initializer (see Case 5) since entity mold constructors or initializers and relationship mold constructors or initializers have analogous instantiations for parameters $\mathbb{U}$ (see Table 4.3 on page 80) and $\mathbb{C}$ (see Table 6.7) as well as analogous restrictions on field declarations (see Table 6.6) for the writable instances determined by parameter $\mathbb{U}$.

Case 7 (Non-interposed relationship element method): For the execution of a non-interposed relationship element method r on an instance o, we show that Lemma 6.13 holds for k transitive routine invocations made during r's execution. The proof is in parts analogous to the proof of Case 3. Therefore, we only prove the parts that are specific to the execution of a non-interposed relationship element method and refer to the proof of Case 3 otherwise. We consider a relationship R with the participant base types A and B and the role identifiers a and b, respectively, and proceed by secondary weak induction on the index i of semi-visible states of r's execution:

(i = 0): Base case: immediate since no fields can yet have been written to.

(i = n + 1): Let n be an arbitrary natural number. By inner induction, we can assume that (1) and (2) hold in s_n and we must show that they hold in s_{n+1}. There are two cases to be considered:

(n + 1 is odd): Control remains within r between s_n and s_{n+1}. According to parameter $\mathbb{U}$ (see Table 4.3 on page 80), r can write to the fields of the following instances between s_n and s_{n+1}: o, $o.a$, and $o.b$.

(1) We know that o is a relationship element instance and that $o' \neq o$, by assumption. According to parameter $\mathbb{C}$ (see Table 6.7), a (possibly transitive) caller o' must then be any of the below mentioned instances. In line with Lemma 6.13, we only consider callers that are extent instances, relationship element instances, or role element instances and ignore the singleton instance of the application App:

(o' = instance of type Extent⟨Element⟨R⟩⟩): According to Table 6.6, the instances to whose fields r can write between s_n and s_{n+1} (see above) are not

allowed to declare any fields of the types Extent⟨Element⟨R⟩⟩, Element⟨R⟩, Element⟨R.a⟩, and Element⟨R.b⟩. As a result, r is guaranteed not to store any references to o' nor references to any element instances comprised in o' (including role element instances related by such element instances) in fields to which r can write between s_n and s_{n+1} and, thus, by inner induction, between s_0 and s_{n+1}.

(*BaseType*(o') = T ∧ R is direct or transitive participant of T ∧ T ≠ App): T must be a relationship with the participant base types C c and D d, let's say. According to Table 6.6 and since the participants relation is acyclic (see System Invariant 4.1 on page 77), the instances to whose fields r can write between s_n and s_{n+1} (see above) are not allowed to declare any fields of the types Extent⟨Element⟨T⟩⟩, Element⟨T⟩, Element⟨T.c⟩, and Element⟨T.d⟩. As a result, r is guaranteed not to store any references to instances that are rooted in the same extent instance as o' in fields to which r can write between s_n and s_{n+1} and, thus, by inner induction, between s_0 and s_{n+1}.

(2) According to Table 6.6, the instances to whose fields r can write between s_n and s_{n+1} (see above) are not allowed to declare any fields of the types Extent⟨Element⟨R⟩⟩, Element⟨R⟩, Element⟨R.a⟩, and Element⟨R.b⟩. As a result, r is guaranteed not to store references to the extent instance ox in which o resides nor references to any element instances comprised in ox (including role element instances related by such element instances) in fields to which r can write between s_n and s_{n+1} and, thus, by inner induction, between s_0 and s_{n+1}.

(n + 1 is even): A direct routine invocation r'' on an instance o'' is made between s_n and s_{n+1}. Since the callee o'' cannot be the singleton application instance, this proof is analogous to the proof of Case 3.

Case 8 (Entity element method): The proof is analogous to the proof of a non-interposed relationship element method (see Case 7) since entity element methods and relationship element methods have analogous instantiations for parameters $\mathbb{U}$ (see Table 4.3 on page 80) and $\mathbb{C}$ (see Table 6.7) as well as analogous restrictions on field declarations (see Table 6.6) for the writable instances determined by parameter $\mathbb{U}$.

Case 9 (Interposed relationship element method): For the execution of an interposed relationship element method r on an instance o, we show that Lemma 6.13 holds for k transitive routine invocations made during r's execution. The proof is in parts analogous to the proof of Case 3. Therefore, we only prove the parts that are specific to the execution of an interposed relationship element method and refer to the proof of Case 3 otherwise. We consider a relationship R with the participant base types A and B and the role identifiers a and b, respectively, and proceed by secondary weak induction on the index i of semi-visible states of r's execution:

(i = 0): Base case: immediate since no fields can yet have been written to.

(i = n + 1): Let n be an arbitrary natural number. By inner induction, we can assume that (1) and (2) hold in s_n and we must show that they hold in s_{n+1}. There are two cases to be considered:

(n + 1 is odd): Control remains within r between s_n and s_{n+1}. According to parameter $\mathbb{U}$ (see Table 4.3 on page 80), r can only write to the fields of o between s_n and s_{n+1}.

(1) We know that o is a role element instance and that $o' \neq o$, by assumption. According to parameter $\mathbb{C}$ (see Table 6.7), a (possibly transitive) caller o' must then be any of the below mentioned instances. In line with Lemma 6.13, we only consider callers that are extent instances, relationship element instances, or role element instances and ignore the singleton instance of the application App:

(o' = instance of type Extent⟨Element⟨R⟩⟩): According to Table 6.6, the instances to whose fields r can write between s_n and s_{n+1} (see above) are not allowed to declare any fields of the types Extent⟨Element⟨R⟩⟩, Element⟨R⟩, Element⟨R.a⟩, and Element⟨R.b⟩. As a result, r is guaranteed not to store any references to o' nor references to any element instances comprised in o' (including role element instances related by such element instances) in fields to which r can write between s_n and s_{n+1} and, thus, by inner induction, between s_0 and s_{n+1}.

(o' = instance of type Element⟨R⟩): According to Table 6.6, the instances to whose fields r can write between s_n and s_{n+1} (see above) are not allowed to declare any fields of the types Extent⟨Element⟨R⟩⟩, Element⟨R⟩, Element⟨R.a⟩, and Element⟨R.b⟩. As a result, r is guaranteed not to store any references to the extent instance ox' in which o' resides nor references to any element instances comprised in ox' (including role element instances related by such element instances) in fields to which r can write between s_n and s_{n+1} and, thus, by inner induction, between s_0 and s_{n+1}.

(*BaseType*(o') = T ∧ R is direct or transitive participant of T ∧ T ≠ App): T must be a relationship with the participant base types C c and D d, let's say. According to Table 6.6 and since the participants relation is acyclic (see System Invariant 4.1 on page 77), the instances to whose fields r can write between s_n and s_{n+1} (see above) are not allowed to declare any fields of the types Extent⟨Element⟨T⟩⟩, Element⟨T⟩, Element⟨T.c⟩, and Element⟨T.d⟩. As a result, r is guaranteed not to store any references to instances that are rooted in the same extent instance as o' in fields to which r can write between s_n and s_{n+1} and, thus, by inner induction, between s_0 and s_{n+1}.

(2) According to Table 6.6, the instances to whose fields r can write between s_n and s_{n+1} (see above) are not allowed to declare any fields of the types Extent⟨Element⟨R⟩⟩, Element⟨R⟩, Element⟨R.a⟩, and Element⟨R.b⟩. As a result, r is guaranteed not to store references to the extent instance ox in which

the relationship element instances that relate o reside nor references to any element instances comprised in ox (including role element instances related by such element instances) in fields to which r can write between s_n and s_{n+1} and, thus, by inner induction, between s_0 and s_{n+1}.

(n + 1 is even): A direct routine invocation r'' on an instance o'' is made between s_n and s_{n+1}. Since the callee o'' cannot be the singleton application instance, this proof is analogous to the proof of Case 3.

□

Owned-called-by-owner

In this section, we introduce Lemma 6.22 that captures the effects of selective ownership on a program's call stack. Lemma 6.22 is stated from the perspective of an owned instance and guarantees that any owned instance residing on the call stack is preceded by its owner. As a result, Lemma 6.22 guarantees that any modifications of an owned instance's field are initiated by the instance's owner. However, since our ownership system is hybrid, this property does not hold for all the instances in a program's heap, but only for those instances in a program's heap that are owned. To contrast the discipline emerging from selective ownership with the "owner-as-modifier" discipline of the Universe Type system [96], we refer to our discipline as the "owned-called-by-owner" discipline.

In the proof of Lemma 6.22, we rely on Proposition 6.16, Proposition 6.18, and Proposition 6.20. Proposition 6.16 provides an upper bound on the regions in a program's heap to which inaccurate references can possibly be leaked. Proposition 6.16 relies on the notion of "being rooted in an extent instance" introduced in Definition 6.12 as well as on the notion of an *implicit reference*, which we define as:

Definition 6.15 (Implicit reference): For an element instance or extent instance o, its implicit references are:

- if o is an extent instance: o and the implicit references of any element instances residing in o;
- if o is a relationship element instance of a relationship R with the participant base types A a and B b: o, $o.a$, and $o.b$;
- if o is an entity element instance: o.

Proposition 6.16 (Escape of owned instance): For any well-formed call stack σ, for any owned instance o', and for any stack frame (o'', r'') contained in σ such that o'' is rooted in the same extent instance as o', if o' escapes r'' to an instance o, then o is either the owner of o' or it holds that $BaseType(o) = BaseType(o')$ or that $BaseType(o)$ is a direct or transitive participant of $BaseType(o')$. In the former case it holds that the reference to the escaped instance o' kept by o is accurate. In the latter case it holds that the reference to the escaped instance o' kept by o is inaccurate (unless the modifier is readonly) and that the instance o only stores a reference to o' in a field f or redirects a query q to o' if o

is the extent instance in which o' is rooted and the field f and the query q are members of o. Any such field f and query q is inaccurate (unless the modifier is readonly).

Proof 6.17: We consider a current receiver instance o'' that is rooted in the same extent instance as o' because such an instance may have an implicit reference (see Definition 6.15) to o'. Since o'' is rooted in the same extent instance as o', it holds that $BaseType(o'') = BaseType(o')$. A temporary reference to o' can escape the routine r'' if r'' returns such a reference to its caller. By the well-formedness of a call stack (see Corollary 6.10 and Definition 6.9), by parameter $\mathbb{C}$ (see Table 6.7), and by System Invariant 6.2 and since $BaseType(o'') = BaseType(o')$, we know that the caller of o'' is either of $BaseType(o')$ or of the same base type as the owner of o'. In the latter case, we know, by System Invariant 6.5 and since the caller-site of an owned instance uses selective ownership modifiers to control access to the owned instance, that the caller of o'' must be the owner of o', which keeps an accurate reference to o'. Furthermore, a temporary reference to o' can escape the routine r'' if r'' passes such a reference on to its (possibly transitive) callees. By the well-formedness of a call stack (see Corollary 6.10 and Definition 6.9) and by parameter $\mathbb{C}$ (see Table 6.7) and since $BaseType(o'') = BaseType(o')$, we know that any (possibly transitive) callee of o'' is of $BaseType(o')$ or of a base type that is a direct or transitive participant of $BaseType(o')$. For any instance p that obtains an escaped, temporary reference to o' and that is of $BaseType(o')$ or of a base type that is a direct or transitive participant of $BaseType(o')$, we know that its reference to o' is inaccurate (unless the modifier is readonly) since p resides at the callee-site of the owner of o'. For such an instance p, we know further, by Lemma 6.13, that p only stores a reference to o' in a field f if p is the extent instance in which o' is rooted and if f is a field of that extent instance. The field f is inaccurate as well, unless the field's modifier is readonly. Furthermore, by Table 6.8, we know, for such an instance p, that p can only declare a query q that redirects to o' if p is the extent instance in which o' is rooted and if q is a query of that extent instance. The query q is inaccurate as well, unless the query's modifier is readonly. □

Selective ownership may permit routine invocations on owned instances through inaccurate references. Proposition 6.18 delimits those cases for extent instances and states, for the execution of a method on an extent instance, under which circumstances the method may get hold of an inaccurate reference to an instance that is of the same base type as the extent instance.

Proposition 6.18 (Possibly inaccurate references for extent instance): For any well-formed call stack σ and for any stack frame (o, r) contained in σ such that o is an extent instance, if r keeps a reference to an owned instance o' such that $BaseType(o') = BaseType(o)$, then the reference can only be inaccurate if o' is rooted in o.

Proof 6.19: By assumption, we know that $BaseType(o) = BaseType(o')$ and that the reference to o' is inaccurate. By Proposition 6.16 we know that an inaccurate reference to o' can at most be stored in a field or redirected to a query of the extent instance in which o' is

rooted. According to System Invariant 6.3 and System Invariant 6.4, this extent instance must have the same owner as o'. Thus, in order for r to get hold of an inaccurate reference to o' or of an inaccurate reference to the extent instance in which o' is rooted, such a reference must escape from a (possibly transitive) caller of o. According to Proposition 6.16, the reference must escape from a routine execution on an instance o'' such that o'' is rooted in the same extent instance as o'. Since o'' is rooted in the same extent instance as o', it follows that $BaseType(o'') = BaseType(o')$, and since $BaseType(o') = BaseType(o)$, that $BaseType(o'') = BaseType(o)$. Since o is an extent instance, it follows from parameter $\mathbb{C}$ (see Table 6.7) that o'' can only be a caller of o if $o'' = o$ and, thus, that o' is rooted in o. □

Similarly, Proposition 6.20 states, for the execution of a relationship element method, how the method may get hold of an inaccurate reference to a role element instance.

Proposition 6.20 (Possibly inaccurate references for relationship element instance): For any well-formed call stack σ and for any stack frame $(o, r)_l$ contained in σ such that o is a relationship element instance, if r keeps a reference to an owned instance o' such that o' is a role element instance and $BaseType(o') = BaseType(o)$, then the reference can only be inaccurate if o is rooted in the same extent instance as o' or if o is not rooted in the same extent instance as o', but there exists a stack frame $(o'', r'')_k$ such that $k < l$ and such that o'' is rooted in the same extent instance as o'.

Proof 6.21: By assumption, we know that $BaseType(o) = BaseType(o')$ and that the reference to o' is inaccurate. By Proposition 6.16 we know that an inaccurate reference to o' can at most be stored in a field or redirected to a query of the extent instance in which o' is rooted. According to System Invariant 6.3 and System Invariant 6.4, this extent instance must have the same owner as o'. Thus, in order for r to get hold of an inaccurate reference to o' or of an inaccurate reference to the extent instance in which o' is rooted, such a reference must escape from a (possibly transitive) caller of o. According to Proposition 6.16, the reference must escape from a routine execution on an instance p such that p is rooted in the same extent instance as o'. Since p is rooted in the same extent instance as o', it follows that $BaseType(p) = BaseType(o')$, and since $BaseType(o') = BaseType(o)$, that $BaseType(p) = BaseType(o)$. According to parameter $\mathbb{C}$ (see Table 6.7), p can be any of the following instances:

- **($p = o$):** Since p is rooted in the same extent instance as o', it follows that o is rooted in the same extent instance as o'.
- **(p = extent instance in which o is rooted):** Since p is the extent instance in which o' is rooted, it follows that o is rooted in the same extent instance as o'.
- **(p = extent instance of $BaseType(o) \wedge o$ not rooted in p):** Since p is the extent instance in which o' is rooted and since o is not rooted in p, it follows that o is not rooted in the same extent instance as o', but it holds for $o'' = p$ that o'' is rooted in the same extent instance as o'. □

We can finally state Lemma 6.22, which captures the "owned-called-by-owner" discipline that selective ownership establishes:

Lemma 6.22 (Call stack and ownership): For any well-formed call stack σ and for any stack frame $(o', r')_n$ contained in σ, if the instance o' is owned, then there exists an index k (where $k < n$) and a stack frame $(o, r)_k$ such that:

(1) o' is owned by o;
(2) for all stack frame $(p', s')_m$ (where $k < m < n$):
  (a) if o' is an extent instance, then p' is also owned by o;
  (b) if o' is an element instance, then it holds that:
    - p' is also owned by o, or,
    - p' is not owned by o, but there exists an index l and stack frame $(p, s)_l$, such that $k < l < m$ and $p \neq p'$, where p is owned by o.

Proof 6.23: We assume that o' is owned and choose k such that, for the stack frames $(o, r)_k$ and $(q, t)_{k+1}$, it holds that $BaseType(o')$ is a direct participant of $BaseType(o)$ and that $BaseType(q) = BaseType(o')$:

(1): By assumption, we know that o' is owned. According to System Invariant 6.2, the base type of the owner of o' is a base type of which $BaseType(o')$ is a direct participant. Since we chose k and the stack frames $(o, r)_k$ and $(q, t)_{k+1}$ such that $BaseType(o')$ is a direct participant of $BaseType(o)$ and such that $BaseType(q) = BaseType(o')$, it follows that the owner of o' is an instance of $BaseType(o)$. We show first *(i)* that q must be an instance that is rooted in the same extent instance as o' and then show *(ii)* that, in this case, o is also the owner of o':

(i) We show that q must be an instance that is rooted in the same extent instance as o' by assuming that q is not rooted in the same extent instance as o' and by reaching a contradiction. If q is rooted in a different extent instance than o', then, it follows from System Invariant 3.6 on page 71 that $q \neq o'$. According to parameter $\mathbb{C}$ (see Table 6.7) and since $BaseType(q) = BaseType(o')$, q can be any of the following instances:

(q = extent instance): Then the callee o' can be any of the following instances:

(o' = relationship or entity element instance): Then, in order for q to invoke a routine on o' the routine executing on q needs to keep an inaccurate reference to o'. Since $BaseType(q) = BaseType(o')$, it follows from Proposition 6.18 that o' is rooted in q, which leads to a contradiction.

(o' = role element instance): Then, o' is either invoked by q or o' is invoked by a relationship element instance x such that $BaseType(x) = BaseType(o')$ and such that x is a (possibly transitive) callee of q. In the former case, we reach a contradiction through Proposition 6.18 as shown in Case (o' is relationship or entity element instance). In the latter case, the routine executing on x needs to keep an inaccurate reference to o' in order for x to invoke a routine on o'. Then, since $BaseType(x) = BaseType(o')$, it follows from Proposition 6.20

that either *(i)* x is rooted in the same extent instance as o' or that *(ii)* x is not rooted in the same extent instance as o', but (possibly transitively) invoked by an instance y which is rooted in the same extent instance as o'. In case *(i)*, we know by System Invariant 6.3 and System Invariant 6.4 that x is owned as well, leading to a contradiction through Proposition 6.18 as shown in Case (o' is relationship or entity element instance). In case *(ii)*, it follows from System Invariant 3.6 on page 71 that $y \neq x$ since the instance x is not rooted in the same extent instance as o'. According to parameter $\mathbb{C}$ (see Table 6.7), y must be the relationship extent instance q. However, this leads to a contradiction since y is rooted in the same extent instance as o'.

(q = relationship element instance): In order for q to invoke a routine on o', the routine executing on q needs to keep an inaccurate reference to o'. Then, since $BaseType(q) = BaseType(o')$, it follows from Proposition 6.20 that either *(i)* q is rooted in the same extent instance as o' or that *(ii)* q is not rooted in the same extent instance as o', but (possibly transitively) invoked by an instance y which is rooted in the same extent instance as o'. In case *(i)*, we know by System Invariant 6.3 and System Invariant 6.4 that q is owned as well, leading to a contradiction through Proposition 6.18 as shown in Case (o' is relationship or entity element instance). In case *(ii)*, it follows from System Invariant 3.6 on page 71 that $y \neq q$ since the instance q is not rooted in the same extent instance as o'. According to parameter $\mathbb{C}$ (see Table 6.7), y must be a relationship extent instance such that $BaseType(y) = BaseType(o')$. However, this leads to a contradictions since q is the most recent instance on the call stack of $BaseType(o')$.

(ii) If q is an instance that is rooted in the same extent instance as o', then, it follows from System Invariant 6.3 and System Invariant 6.4 that q has the same owner as o'. Since the caller-site of an owned instance uses selective ownership modifiers to control access to the owned instance, it follows from System Invariant 6.5 that the caller o of q is the owner of q and, thus, that o is also the owner of o'.

(2): As shown previously in part (1), we know that o' is owned by o. Furthermore, by the well-formedness of a call stack (see Corollary 6.10 and Definition 6.9), we know for any stack frame $(p', s')_m$ (where $k < m < n$) that $BaseType(p') = BaseType(o')$.

(a) If o' is an extent instance, then, according to parameter $\mathbb{C}$ (see Table 6.7), its (possibly transitive) caller p' can only be of $BaseType(o')$ if $p' = o'$. Thus, it holds that p' is also owned by o.

(b) By assumption we know that o' is an element instance and we proceed by weak induction on the index m:

(m = n - 1): Base case where p' is the direct caller of o'. By assumption we know that o' is an element instance, giving rise to the following cases:

(o' = element instance of entity or relationship T): According to parameter $\mathbb{C}$ (see Table 6.7) and since $BaseType(p') = BaseType(o')$, p' can be any of the following instances:

($p' = o'$): Then, since $p' = o'$, p' is also owned by o.

(p' = instance of type Extent⟨Element⟨T⟩⟩): In order for p' to invoke a method on the owned instance o', the routine executing on p' needs to keep an inaccurate reference to the instance o'. Since $BaseType(p') = BaseType(o')$, it follows from Proposition 6.18 that o' is rooted in p'. Then, it follows from System Invariant 6.3 that p' has the same owner as o', and, thus, that p' is also owned by o.

(o' = role element instance of type Element⟨R.a⟩): According to parameter $\mathbb{C}$ (see Table 6.7) and since $BaseType(p') = BaseType(o')$, p' can be any of the following instances:

($p' = o'$): Then, since $p' = o'$, p' is also owned by o.

(p' = instance of type Element⟨R⟩): In order for p' to invoke a routine on o', the routine executing on p' needs to keep an inaccurate reference to o'. Then, since $BaseType(p') = BaseType(o')$, it follows from Proposition 6.20 that either *(i)* p' is rooted in the same extent instance as o' or that *(ii)* p' is not rooted in the same extent instance as o', but (possibly transitively) invoked by an instance y which is rooted in the same extent instance as o'. Since y is rooted in the same extent instance as o', it holds that $BaseType(y) = BaseType(o')$. In case *(i)*, we know by System Invariant 6.3 and System Invariant 6.4 that p' has the same owner as o' and, thus, that that p' is also owned by o. In case *(ii)*, we know by System Invariant 6.3 and System Invariant 6.4 that y has the same owner as o' and, thus, that y is also owned by o.

(p' = instance of type Extent⟨Element⟨R⟩⟩): In order for p' to invoke a method on the owned instance o', the routine executing on p' needs to keep an inaccurate reference to the instance o'. Since $BaseType(p') = BaseType(o')$, it follows from Proposition 6.18 that o' is rooted in p'. Then, it follows from System Invariant 6.3 and System Invariant 6.4 that p' has the same owner as o', and, thus, that p' is also owned by o.

(m = n - x): Let x be an arbitrary natural number such that $1 < x < n - k$. In this case, p' is a transitive caller of o' and the direct caller of the receiver instance p'' of the stack frame $(p'', s'')_{m+1}$. By assumption we know that o' is an element instance, giving rise to the following cases:

(o' = element instance of entity or relationship T): If $p'' = o'$, then the proof is analogous to the one of the base case. Otherwise, it follows from parameter $\mathbb{C}$ (see Table 6.7) and from $BaseType(p'') = BaseType(o')$ that p'' must be an instance of type Extent⟨Element⟨T⟩⟩. Then, it follows similarly from parameter $\mathbb{C}$ and from $BaseType(p') = BaseType(o')$ that $p' = p''$. By induction, we know that (2) (b) holds for the stack frame $(p'', s'')_{m+1}$. Since $p' = p''$, we can thus conclude that p' is also owned by o or that there exists an index l and stack frame $(p, s)_l$, such that $k < l < m$ and $p \neq p'$, where p is owned by o.

(o' = role element instance of type Element⟨R.a⟩): If $p'' = o'$, then the proof is analogous to the one of the base case. Otherwise, it follows from parameter $\mathbb{C}$ (see Table 6.7) and from $BaseType(p'') = BaseType(o')$ that p'' must be an instance of type Element⟨R⟩ or of type Extent⟨Element⟨R⟩⟩. We consider each case separately:

(p'' = instance of type Element⟨R⟩): Then, it follows similarly from parameter $\mathbb{C}$ and from $BaseType(p') = BaseType(o')$ that either $p' = p''$ or that p' is an instance of type Extent⟨Element⟨R⟩⟩. Again, we consider each case separately:

($p' = p''$): Then, we know by induction that (2) (b) holds for the stack frame $(p'', s'')_{m+1}$. Since $p' = p''$, we can thus conclude that p' is also owned by o or that there exists an index l and stack frame $(p, s)_l$, such that $k < l < m$ and $p \neq p'$, where p is owned by o.

(p' = instance of type Extent⟨Element⟨R⟩⟩): Then, we know by induction that (2) (b) holds for the stack frame $(p'', s'')_{m+1}$. Thus, we know that p'' is also owned by o or that p'' is not owned by o, but that there exists an index l and stack frame $(p, s)_l$, such that $k < l < m + 1$ and $p \neq p''$, where p is owned by o. If p'' is owned by o, then the routine executing on p' needs to keep an inaccurate reference to the instance p'' to invoke a method on the owned instance p''. Since $BaseType(p') = BaseType(p'')$, it follows from Proposition 6.18 that p'' is rooted in p'. Then, it follows from System Invariant 6.3 that p' has the same owner as p'', and, thus, that p' is also owned by o. If p'' is not owned by o, then, according to parameter $\mathbb{C}$ and since $k < l$ and $p \neq p''$, the current receiver instance p of the stack frame $(p, s)_l$ must be an instance of type Extent⟨Element⟨R⟩⟩. It follows from the well-formedness of a call stack (see Corollary 6.10 and Definition 6.9) that $p' = p$ and, thus, since p is owned by o, that p' is also owned by o.

(p'' = instance of type Extent⟨Element⟨R⟩⟩): Then, it follows similarly from parameter $\mathbb{C}$ and from $BaseType(p') = BaseType(o')$ that $p' = p''$. By induction, we know that (2) (b) holds for the stack frame $(p'', s'')_{m+1}$. Since $p' = p''$, we can thus conclude that p' is also owned by o or that there exists an index l and stack frame $(p, s)_l$, such that $k < l < m$ and $p \neq p'$, where p is owned by o. □

6.3 Extended verification technique

In this section, we extend the verification technique presented in Chapter 5 with ownership-based invariants. Henceforth, we refer to the verification technique presented in Chapter 5 as *basic verification technique (BVT)* and to new verification technique presented in this section as *extended verification technique (EVT)*.

Invariant	Instance of invariant	Referable location
Application App		
invariant	App *app*	Global variables of *app*.
		For any instance *o* that is owned by *app*, locations of *o* as defined by Table 6.10.
Relationship R with participant base types A a and B b		
extent invariant	Extent⟨Element⟨R⟩⟩ *rx*	Content of *rx*.
		Fields of *rx*.
		For any instance Element⟨R⟩ *r* in *rx*: fields of *r*, fields of *r.a*, and fields of *r.b*.
		For any instance *o* that is owned by *rx*, locations of *o* as defined by Table 6.10.
element invariant	Element⟨R⟩ *r*	Fields of *r*, fields of *r.a*, and fields of *r.b*.
		For any instance *o* that is owned by *r*, locations of *o* as defined by Table 6.10.
Entity E		
extent invariant	Extent⟨Element⟨E⟩⟩ *ex*	Content of *ex*.
		Fields of *ex*.
		Fields of any instance Element⟨E⟩ e in *ex*.
element invariant	Element⟨E⟩ e	Fields of *e*.

Table 6.9: Locations on which an invariant is allowed to depend (parameter $\mathbb{D}$, indirectly). This table extends Table 4.4 on page 82 with **selective ownership**, differences are highlighted in **bold**. See Table 6.10 for overview of referable locations.

6.3.1 Admissibility criteria

In this section, we revisit the admissibility criteria introduced in Section 4.3 on page 78. To facilitate the declaration of invariants that depend on locations of owned instances, we update the criterion "admissible invariants" as well as the criterion "admissible invocations".

Admissible invariants

The extended verification technique allows invariants not only to depend on locations that are encapsulated by the invariant-declaring type but also on locations that are owned by the invariant-declaring type. Table 6.9 indirectly instantiates the framework parameter $\mathbb{D}$ for the extended verification technique. Differences to the corresponding instantiation of parameter $\mathbb{D}$ for the basic verification technique (see Table 4.4 on page 82) are highlighted in bold. As indicated by Table 6.9, the new instantiation of parameter $\mathbb{D}$ includes the one for the basic verification technique and complements it with ownership-based invariants.

Table 6.9 indicates, for a particular invariant of an instance, the locations on which the

Owned instance	Location referable by owner
Relationship R with participant base types A a and B b	
Extent⟨Element⟨R⟩⟩ *rx*	Content of *rx*.
	Fields of *rx*.
Element⟨R⟩ *r*	Fields of *r*.
Element⟨R._⟩ *rab*	Fields of *rab*.
Entity E	
Extent⟨Element⟨E⟩⟩ *ex*	Content of *ex*.
	Fields of *ex*.
Element⟨E⟩ *e*	Fields of *e*.

Table 6.10: Locations of owned instance on which an invariant of the instance's owner is allowed to depend.

invariant is allowed to depend. However, the actual locations of an owned instance are not listed in Table 6.9, but determined by Table 6.10. For example, if an element instance *r* of a relationship R with the participant base types A a and B b owns an entity element instance *e*, then the invariant of *r* is allowed to depend on the fields of *r*, *r.a*, and *r.b* as well as on the fields of *e*. Table 6.10 also indicates, for an owned extent instance, that the owner's invariant is only allowed to depend on the extent instance's content and fields, but not on any fields of the element instances that reside in the extent instance. This treatment is necessary to decouple exclusive extent ownership from inclusive extent ownership.

According to Table 6.9 and Table 6.10, the `Composite` element invariant declared on line 18 in Figure 6.1 is admissible since a `Composite` element instance is the owner of its related `Parent` extent instance and since the `Composite` element instance's invariant only depends on its instance's locations and on its instance owned `Parent` extent instance's locations.

Admissible invocations

As discussed in Section 6.2.3, selective ownership removes some of the liberalness with regard to invocations of mold constructors and mold initializers and requires those invocations to propagate in the direction of a program's participants relation. Table 6.7 shows the new instantiation of parameter $\mathbb{C}$.

6.3.2 Proof obligations and assumptions

To account for ownership-based invariant declarations, we re-instantiate the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for the extended verification technique. Since the application resides at the top of a program's participants relation, no ownership can be imposed on the application singleton instance. Consequently, we only re-instantiate the parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$

Parameters for: default extent constructor of an entity or relationship T **Current receiver:** Extent⟨Element⟨T⟩⟩ *tx*	
𝕏:	For any instance Extent⟨Element⟨T⟩⟩ *tx′* such that *tx* ≠ *tx′*: invariant of *tx′* and invariants of all instances Element⟨T⟩ *t′* in *tx′*.
	For any entity or relationship S such that S ≠ T and T is not a direct or transitive participant of S: for any instance Extent⟨Element⟨S⟩⟩ *sx*, invariant of *sx* and invariants of all instances Element⟨S⟩ *s* in *sx*.
𝕍:	Invariant of *tx*.
	Invariant of *tx*'s owner.
𝔹:	*n/a*
𝔼:	Invariant of *tx*.

Table 6.11: Parameters 𝕏, 𝕍, 𝔹, and 𝔼 for default entity and relationship extent constructors. This table extends Table 5.6 with **selective ownership**, differences are highlighted in **bold**.

Parameters for: extent constructor of an entity or relationship T **Current receiver:** Extent⟨Element⟨T⟩⟩ *tx*	
𝕏:	For any instance Extent⟨Element⟨T⟩⟩ *tx′* such that *tx* ≠ *tx′*: invariant of *tx′* and invariants of all instances Element⟨T⟩ *t′* in *tx′*.
	For any entity or relationship S such that S ≠ T and T is not a direct or transitive participant of S: for any instance Extent⟨Element⟨S⟩⟩ *sx*, invariant of *sx* and invariants of all instances Element⟨S⟩ *s* in *sx*.
𝕍:	Invariant of *tx* and invariants of all instances Element⟨T⟩ *t* in *tx*.
	Invariant of *tx*'s owner.
𝔹:	If callee = *tx*, then invariant of *tx* and invariants of all instances Element⟨T⟩ *t* in *tx*.
	If callee = instance Element⟨T⟩ *t* in *tx*, then invariant of *t*.
	If callee = instance Element⟨T⟩ *t* in *tx* and T = relationship with participant base types A and B that is A-benign and/or B-benign, then B-uniqueness and/or A-uniqueness assertion of invariant of *tx*, resp.
𝔼:	Invariant of *tx* and invariants of all instances Element⟨T⟩ *t* in *tx*.

Table 6.12: Parameters 𝕏, 𝕍, 𝔹, and 𝔼 for programmer-defined entity and relationship extent constructors. This table extends Table 5.7 with **selective ownership**, differences are highlighted in **bold**.

for (possibly default) extent constructors, extent methods, element methods, and mold constructors and initializers. Table 6.11, Table 6.12, Table 6.13, Table 6.14, Table 6.15, Table 6.16, Table 6.17, and Table 6.18 show the corresponding instantiations.

Since selective ownership is tied to a program's participants relation (see System Invariant 6.2) and since the parameters 𝕏, 𝕍, 𝔹, and 𝔼 of the basic verification technique are also stated relative to a program's participants relation, the new instantiations of the parameters only slightly differ from their corresponding instantiations for the basic verification technique (see Section 5.2 on page 91). We highlight the differences between the two instantiations in bold. The highlights indicate that the extended verification technique only differs from the basic verification technique in its instantiation of parameter 𝕍. This parameter includes the invariant of a current receiver instance's owner in addition to

Parameters for: extent method of an entity or relationship T **Current receiver:** Extent⟨Element⟨T⟩⟩ *tx*
$\mathbb{X}$: For any instance Extent⟨Element⟨T⟩⟩ *tx*′: invariant of *tx*′ and invariants of all instances Element⟨T⟩ *t*′ in *tx*′.
For any entity or relationship S such that S ≠ T and T is not a direct or transitive participant of S: for any instance Extent⟨Element⟨S⟩⟩ *sx*, invariant of *sx* and invariants of all instances Element⟨S⟩ *s* in *sx*.
$\mathbb{V}$: Invariant of *tx* and invariants of all instances Element⟨T⟩ *t* in *tx*.
Invariant of *tx*'s owner.
$\mathbb{B}$: If callee = *tx*, then invariant of *tx* and invariants of all instances Element⟨T⟩ *t* in *tx*.
If callee = instance Element⟨T⟩ *t* in *tx*, then invariant of *t*.
If callee = instance Element⟨T⟩ *t* in *tx* and T = relationship with participant base types A and B that is A-benign and/or B-benign, then B-uniqueness and/or A-uniqueness assertion of invariant of *tx*, resp.
$\mathbb{E}$: Invariant of *tx* and invariants of all instances Element⟨T⟩ *t* in *tx*.

Table 6.13: Parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for entity and relationship extent methods. This table extends Table 5.8 with **selective ownership**, differences are highlighted in **bold**.

Parameters for: element method of an entity or free relationship T **Current receiver:** Element⟨T⟩ *t*
$\mathbb{X}$: Invariant of *t*.
For any entity or relationship S such that S ≠ T and T is not a direct or transitive participant of S: for any instance Extent⟨Element⟨S⟩⟩ *sx*, invariant of *sx* and invariants of all instances Element⟨S⟩ *s* in *sx*.
$\mathbb{V}$: Invariant of *t*.
Invariant of instance Extent⟨Element⟨T⟩⟩ *tx* in which *t* resides.
Invariant of *t*'s owner.
$\mathbb{B}$: If callee = *t*, then invariant of *t*.
$\mathbb{E}$: Invariant of *t*.
Preservation of invariant of instance Extent⟨Element⟨T⟩⟩ *tx* in which *t* resides.

Table 6.14: Parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for entity and free, non-interposed relationship element methods. This table extends Table 5.9 with **selective ownership**, differences are highlighted in **bold**.

Adjunct parameter for: element method of an A-benign relationship R **Participants:** base types A and B and role identifiers a and b, resp. **Current receiver:** Element⟨R⟩ *r*
$\mathbb{X}^+$: B-uniqueness assertion of invariant of instance Extent⟨Element⟨R⟩⟩ *rx* in which *r* resides.

Table 6.15: Adjunct for parameter $\mathbb{X}$ for unilaterally benign, non-interposed relationship element methods. This parameter conjoins with the parameters listed in Table 6.14 and, if the relationship is inversely benign or malign with the parameters listed in Table 6.15 or Table 6.16, resp. This table extends Table 5.10 with **selective ownership**, differences are highlighted in **bold**.

Adjunct parameter for: element method of a B-malign relationship R **Participants:** base types A and B and role identifiers a and b, resp. **Current receiver:** Element⟨R⟩ *r*
$\mathbb{V}^+$: Invariants of all instances Element⟨R⟩ $r' \neq r$ such that $r'.b = r.b$.
$\mathbb{E}^+$: Preservation of invariants of all instances Element⟨R⟩ $r' \neq r$ such that $r'.b = r.b$.

Table 6.16: Adjuncts for parameters $\mathbb{V}$ and $\mathbb{E}$ for unilaterally malign, non-interposed relationship element methods. These parameters conjoin with the parameters listed in Table 6.14 and, if the relationship is inversely benign or malign with the parameters listed in Table 6.15 or Table 6.16, resp. This table extends Table 5.11 with **selective ownership**, differences are highlighted in **bold**.

Parameters for: element method of a role R.a declared by relationship R **Current receiver:** Element⟨R.a⟩ *ra*	
$\mathbb{X}$:	For any entity or relationship S such that S ≠ R and R is not a direct or transitive participant of S: for any instance Extent⟨Element⟨S⟩⟩ *sx*, invariant of *sx* and invariants of all instances Element⟨S⟩ *s* in *sx*.
$\mathbb{V}$:	Invariants of all instances Element⟨R⟩ *r* such that *r.a* = *ra*.
	Invariant of instance Extent⟨Element⟨R⟩⟩ *rx* in which all instances Element⟨R⟩ *r* with *r.a* = *ra* reside.
	Invariant of *ra*'s owner.
$\mathbb{B}$:	–
$\mathbb{E}$:	Preservation of invariants of all instances Element⟨R⟩ *r* such that *r.a* = *ra*.
	Preservation of invariant of instance Extent⟨Element⟨R⟩⟩ *rx* in which all instances Element⟨R⟩ *r* with *r.a* = *ra* reside.

Table 6.17: Parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for interposed relationship element methods. This table extends Table 5.12 with **selective ownership**, differences are highlighted in **bold**.

Parameters for: mold constructor or initializer of an entity or relationship T **Current receiver:** Mold⟨T⟩ *t*	
$\mathbb{X}$:	None.
$\mathbb{V}$:	None.
$\mathbb{B}$:	*n/a*
$\mathbb{E}$:	None.

Table 6.18: Parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for entity and non-interposed relationship mold constructors and programmer-defined initializers. This table extends Table 5.13 with **selective ownership**, differences are highlighted in **bold**.

the invariants of the current receiver instance. The instantiation of parameter $\mathbb{V}$ relies on System Invariant 6.3 and System Invariant 6.4, which guarantee, in the case of inclusive ownership, that element instances have the same owner as the extent instance in which the element instances are rooted.

6.3.3 Soundness

To prove the extended verification technique sound, we rely on the soundness of the basic verification technique (see Section 5.3 on page 103). As the extended verification technique only differs in its instantiations of the parameters $\mathbb{C}$, $\mathbb{D}$, and $\mathbb{V}$ from the basic verification technique, we must reconsider those propositions from Section 5.3 that involve the notion of a call stack and the concept of an invalidator. We have already elaborated on the effects of selective ownership on a program's call stack in Section 6.2.3. Next, we discuss how the concept of an invalidator is affected by selective ownership.

Corollary 6.24 relies on the definition of an invalidator introduced in Definition 5.14 on page 107 and accounts for the fact that invariants may depend on locations of owned instances (see Section 6.3.1). We highlight the differences between Proposition 5.15 on page 107 and its updated version in bold:

Corollary 6.24 (Restatement of "set of invalidators" in EVT): For the possible instances in a Rumer program that is successfully type checked and verified using the **extended verification technique**, the invalidators for such an instance o are:

- For an application instance o: the application instance o; **if o exclusively owns an extent instance tx, then the extent instance tx; if o inclusively owns an extent instance tx, then the extent instance tx as well as all the element instances t residing in tx (including the role element instances related by such an element instance t if tx is a relationship extent instance)**.
- For a relationship extent instance o: the relationship extent instance o, all relationship element instances residing in o, and all role element instances related by the relationship element instances residing in o; **if o exclusively owns an extent instance tx, then the extent instance tx; if o inclusively owns an extent instance tx, then the extent instance tx as well as all the element instances t residing in tx (including the role element instances related by such an element instance t if tx is a relationship extent instance)**.
- For a relationship element instance o that is neither A-malign nor B-malign (first four rows in Table 5.4): the relationship element instance o, o's role element instances, and the relationship extent instance in which o resides; **if o exclusively owns an extent instance tx, then the extent instance tx; if o inclusively owns an extent instance tx, then the extent instance tx as well as all the element instances t residing in tx (including the role element instances related by such an element instance t if tx is a relationship extent instance)**.
- For a relationship element instance o of a relationship R with participant base types A and B and role identifiers a and b, respectively, that is A-malign and/or B-malign (last five rows in Table 5.4): the relationship element instance o, o's role element instances, all relationship element instances that reside in the relationship extent instance of o and that have the role element instance $o.a$ and/or $o.b$, respectively, as participant, and the relationship extent instance in which o resides; **if o exclusively owns an extent**

instance *tx*, then the extent instance *tx*; if *o* inclusively owns an extent instance *tx*, then the extent instance *tx* as well as all the element instances *t* residing in *tx* (including the role element instances related by such an element instance *t* if *tx* is a relationship extent instance).

- For a role element instance o: *n/a*.
- For an entity extent instance o: the entity extent instance o and all entity element instances residing in o.
- For an entity element instance o: the entity element instance o and the entity extent instance in which o resides.

Proof 6.25: Corollary 6.24 is a restatement of Proposition 5.15 on page 107 based on the new instantiation of parameter $\mathbb{D}$ (see Table 6.9) and on System Invariant 6.3 and System Invariant 6.4. System Invariant 6.3 guarantees that owned entity and relationship element instances have the same owner as the extent instance in which the entity and relationship element instances reside and, thus, that modifications of any element instances residing in the same extent instance may only compromise the ownership-based invariant of the common owner. System Invariant 6.4 guarantees that owned role element instances have the same owner as the relationship element instances that relate the role element instances and, thus, that modifications of any role element instances that are related by the same relationship element instance may only compromise the ownership-based invariant of the common owner. □

To capture the distinction between the basic verification technique and the extended verification technique, we consider whether an invalidator takes care to preserve (or even re-establish) the invariants it breaks. Depending on the proof obligation $\mathbb{E}$ imposed on the routines executing on an invalidator, we divide invalidators into *responsible* and *irresponsible* invalidators:

Definition 6.26 (Responsible invalidator): An instance o is a responsible invalidator for an instance o' if and only if o is an invalidator for o' and every routine executing on o preserves or establishes the invariant of o'.

Definition 6.27 (Irresponsible invalidator): An instance o is an irresponsible invalidator for an instance o' if and only if o is an invalidator for o' and every routine executing on o neither preserves nor establishes the invariant of o'.

Lemma 6.32 delimits the base types of responsible invalidators. Lemma 6.34 relates irresponsible invalidators to ownership. The proofs of those lemmas rely on the following two propositions:

Proposition 6.28 (Base types of owning and owned instances): For any instance o, if o is the owner of an instance o', then it holds that $\mathit{BaseType}(o) \neq \mathit{BaseType}(o')$.

Proof 6.29: Proposition 6.28 follows directly from System Invariant 6.2 and from System Invariant 4.1 on page 77. □

Proposition 6.30 (Base types of depending instances): For any instance o, if the invariant of o is allowed to depend on locations of an instance o', then it either holds that $BaseType(o) = BaseType(o')$ or that o is owned and that o is the owner of o'.

Proof 6.31: Proposition 6.30 follows directly from the routine-specific instantiations of parameter $\mathbb{D}$ (see Table 6.9 and Table 6.10) as well as from Proposition 6.28. □

The lemma that delimits the base type of a responsible invalidator is as follows:

Lemma 6.32 (Responsible invalidator and base types): An instance o is a responsible invalidator for an instance o' if and only if o is an invalidator for o' and $BaseType(o) = BaseType(o')$.

Proof 6.33: Lemma 6.32 follows from Corollary 6.24, Definition 6.26, Proposition 6.28, Corollary 5.17 on page 108, and the routine-specific instantiations of parameter $\mathbb{E}$ (see Table 5.5 on page 97, Table 6.11, Table 6.12, Table 6.13, Table 6.14, Table 6.15, Table 6.16, Table 6.17, and Table 6.18). □

The lemma that relates irresponsible invalidators to ownership is as follows:

Lemma 6.34 (Irresponsible invalidators and ownership): For any instance o, o is an irresponsible invalidator for an instance o' if and only if o' is the owner of o, if o is owned.

Proof 6.35: We prove Lemma 6.34 by showing both directions of the biconditional:

(⇒): By assumption, we know that o is an irresponsible invalidator for o' and that o is owned. By Definition 6.27, we know further that o is an invalidator for o' and that any routine executing on o neither preserves nor establishes the invariant of o' in the routine's final state. From Definition 6.26 and Lemma 6.32 it thus follows that $BaseType(o) \neq BaseType(o')$. Since o is an invalidator for o', the invariant of o' must depend on locations to which o can write. According to parameter $\mathbb{U}$ (see Table 4.3 on page 80), the instances of these locations must all be of base type $BaseType(o)$. Since $BaseType(o) \neq BaseType(o')$, it follows from Proposition 6.30 that o' is the owner of o.

(⇐): By assumption, we know that o is owned and that o' is the owner of o. By System Invariant 6.2 and Corollary 6.24 we know that o is an invalidator for o'. By Proposition 6.28 we know that $BaseType(o) \neq BaseType(o')$. By considering the routine-specific instantiations of parameter $\mathbb{E}$ (see Table 5.5 on page 97, Table 6.11, Table 6.12, Table 6.13, Table 6.14, Table 6.15, Table 6.16, Table 6.17, and Table 6.18), we know further that, for a routine execution r on an instance o, r neither preserves nor establishes invariants of any instances of a different base type than $BaseType(o)$. Since $BaseType(o) \neq BaseType(o')$, no routine execution on o preserves or establishes the invariant of o', and, thus, o is an irresponsible invalidator for o'. □

The next two propositions guarantee that invalidators are either responsible or irrespon-

sible, but not both:

Proposition 6.36 (Invalidators are responsible or irresponsible): For any instance o, if o is an invalidator for an instance o', then o is a responsible invalidator or irresponsible invalidator for o'.

Proof 6.37: We prove Proposition 6.36 by showing that the listing of invalidators provided in Corollary 6.24 only contains responsible or irresponsible invalidators. According to Lemma 6.34, all the invalidators highlighted in bold in Corollary 6.24 are irresponsible invalidators. The remaining invalidators listed in Corollary 6.24, on the other hand, are all of the same base type as the instances for which they are invalidators, and, thus, by Lemma 6.32, those invalidators are responsible invalidators. □

Proposition 6.38 (Responsible and irresponsible invalidators are disjoint): For any instance o, the set of instances O' for which o is a responsible invalidator and the set of instances O'' for which o is an irresponsible invalidator are disjoint.

Proof 6.39: By Lemma 6.32, we know that the set of instances O' for which o is a responsible invalidator must be of the same base type as o. By Lemma 6.34 and Proposition 6.28, we know further that the set of instances O'' for which o is an irresponsible invalidator must be of a different base type than o. Since the set of instances O' and O'' contain instances of pairwise different base types, those sets are disjoint. □

Given the differentiation between responsible invalidators and irresponsible invalidators, we can relate the invalidators of the basic verification technique (see Proposition 5.15 on page 107) to the ones of the extended verification technique (see Corollary 6.24) as follows:

Proposition 6.40 (Invalidators of BVT versus invalidators of EVT): The invalidators of the basic verification technique relate to the invalidators of the extended verification technique as follows:

$$Invalidators_{BVT} = Invalidators_{responsible}$$
$$Invalidators_{EVT} = Invalidators_{responsible} \cup Invalidators_{irresponsible}$$

Proof 6.41: Proposition 6.40 follows directly from the definition of an invalidator in the extended verification technique (see Corollary 6.24), from the fact that invalidators are either responsible invalidators or irresponsible invalidators, but not both (see Proposition 6.36 and Proposition 6.38), from the fact that an invalidator is at most an irresponsible invalidator for its owner (see Lemma 6.34), and from the fact that only the extended verification technique allows invariants to depend on locations of owned instances (see Table 4.4 on page 82 versus Table 6.9). □

Relying on Proposition 6.40, we can restate the soundness theorem for the basic verification technique (see Theorem 5.35 on page 113) as follows (we highlight the differences between Theorem 5.35 and the new Corollary 6.42 in bold):

Corollary 6.42 (Restatement of "Soundness of BVT"): For any routine execution r on an instance o in a successfully type checked and verified Rumer program, if the invariants $\mathbb{X}_{(o,r)}$ hold in the initial state of r's execution, then:

(1) in all semi-visible states s_i of r's execution:
   (a) at most the invariants of those instances for which o is a **responsible invalidator** are lost between the semi-visible states s_0 and s_i;
   (b) if s_i is a pre-state of a direct routine invocation r' on an instance o', the invariants $\mathbb{X}_{(o',r')}$ hold;

(2) in the final state s_{final} of r's execution: the invariants of those instances for which o is a **responsible invalidator** are preserved (since s_0) and also established for any of these instances for which o is a guarantor, and the invariants $\mathbb{X}_{(o,r)}$ hold.

Proof 6.43: Corollary 6.42 is a restatement of Theorem 5.35 on page 113 based on Proposition 6.40 and on the updated account of a well-formed call stack (see Corollary 6.10 and Definition 6.9). □

Before stating the main soundness Theorem 6.53, we introduce an auxiliary corollary as well as auxiliary lemmas that are used in the proof of the main soundness theorem.

The following corollary is a restatement of Proposition 5.22 on page 109, which gives an upper bound on the invariants lost between two consecutive semi-visible states between which control remains within a routine. We highlight the differences between Proposition 5.22 and its updated version below in bold:

Corollary 6.44 (Restatement of "Invalidators and lost invariants" in EVT): For any routine execution r on an instance o, consider any consecutive semi-visible states s_i and s_{i+1} of r's execution between which no routine is invoked (i.e., s_i is even and control remains within r). Then, at most the invariants of those instances for which o is a **responsible invalidator or irresponsible invalidator** are lost between the semi-visible states s_i and s_{i+1}.

Proof 6.45: Corollary 6.44 is a restatement of Proposition 5.22 on page 109 based on the updated concept of an invalidator (see Corollary 6.24) as well as on Lemma 6.32. □

The succeeding auxiliary lemmas are all based on the concept of a *quasi-irresponsible* invalidator. Such an invalidator is *not* an invalidator for an instance, but guaranteed to be *preceded* by one on the call stack. The instances p' in Lemma 6.22 that are not owned by o are exactly quasi-irresponsible invalidators. Our choice of terminology may be surprising since a quasi-irresponsible invalidator is not an invalidator at all. However, as we will see, the term quasi-irresponsible invalidator allows for a succinct formulation of the main soundness Theorem 6.53.

Definition 6.46 (Quasi-irresponsible invalidator): An instance o is a quasi-irresponsible invalidator for an instance o' if and only if o is not an invalidator for o' but (possibly transitively) invoked by a caller p such that $p \neq o$ and such that p is an irresponsible

invalidator for o'.

The following lemma indicates that a callee does not expect the invariants of those instances for which its (possibly transitive) callers are irresponsible or quasi-irresponsible invalidators:

Lemma 6.47 (**Expectations towards caller-(quasi)-irresponsible invariants**)**:** For any routine execution r on an instance o, the routine r does not expect any invariants of instances for which the routine's (possibly transitive) callers are irresponsible or quasi-irresponsible invalidators.

Proof 6.48: From Lemma 6.34 and System Invariant 6.2, it follows that the base type of an irresponsible invalidator for an instance is a direct participant of the base type of that instance. Furthermore, from Definition 6.46 and parameter $\mathbb{C}$ (see Table 6.7), it follows that the base type of a quasi-irresponsible invalidator for an instance is a direct or transitive participant of the base type of that instance. Thus, it follows from the well-formedness of a call stack (see Corollary 6.10 and Definition 6.9), from the acyclicity of the participants relation (see System Invariant 4.1 on page 77), and from parameter $\mathbb{C}$ (see Table 6.7), that the base type of the callee o is a direct or transitive participant of the base type of any instance for which a (possibly transitive) caller of o is an irresponsible or quasi-irresponsible invalidator. According to the routine-specific instantiations of parameter $\mathbb{X}$ (see Table 5.5 on page 97, Table 6.11, Table 6.12, Table 6.13, Table 6.14, Table 6.15, Table 6.16, Table 6.17, and Table 6.18), an instance whose invariant is expected in the visible states of the routine execution r on the callee o is of $\mathit{BaseType}(o)$ or of any base type of which $\mathit{BaseType}(o)$ is not a direct or transitive participant. Therefore, the invariant of any instance for which a (possibly transitive) caller of o is an irresponsible or quasi-irresponsible invalidator is not expected in the visible states of routine r. □

Inversely, the following two lemmas indicate what kind of invalidator the caller becomes for those instances for which the callee is a irresponsible or quasi-irresponsible invalidator:

Lemma 6.49 (**Responsibility for callee-irresponsible invariants**)**:** For any invocation of a routine r' on an instance o' made during the execution of a routine r on an instance o, for the instances for which the callee o' is an irresponsible invalidator, the caller o is either a responsible, irresponsible or, quasi-irresponsible invalidator.

Proof 6.50: By assumption, we know that the callee o' is an irresponsible invalidator and, thus, by Lemma 6.34, we know that o' is owned and that it is an irresponsible invalidator for its owner. According to Lemma 6.22, the caller o must either be the owner of o' or an instance that is a direct or transitive callee of the owner of o'. We consider each case separately:

(o = owner of o'): By assumption, we know that o' is an irresponsible invalidator for its owner o, which also is its caller, by case distinction. According to Corollary 6.24

and Lemma 6.32, an owner is its own responsible invalidator, and, thus, the caller o is a responsible invalidator for the instance for which the callee o' is an irresponsible invalidator.

(o = callee of owner of o'): By assumption, we know that o' is an irresponsible invalidator for its owner. Furthermore, by case distinction, we know that the caller o is (possibly transitively) called by the owner of o'. According to Lemma 6.22, o is either owned by the owner of o' or is not owned by the owner of o' but is (possibly transitively) invoked by a caller that is owned by the owner of o'. If o is owned by the owner of o', then it follows from Lemma 6.34 that o is an irresponsible invalidator for the instance for which the callee o' is an irresponsible invalidator. Otherwise, o is not a responsible invalidator for the owner of o' either (see Lemma 6.32) and, hence, not an invalidator for the owner of o' at all (see Proposition 6.36). Thus, it follows from Definition 6.46 that the caller o is a quasi-irresponsible invalidator for the instance for which the callee o' is an irresponsible invalidator. □

Lemma 6.51 (Responsibility for callee-quasi-irresponsible invariants): For any invocation of a routine r' on an instance o' made during the execution of a routine r on an instance o, for the instances for which the callee o' is a quasi-irresponsible invalidator, the caller o is either an irresponsible or quasi-irresponsible invalidator.

Proof 6.52: Lemma 6.51 follows immediately from Definition 6.46, Proposition 6.36, and Lemma 6.22. □

We are finally in the position to state the main soundness theorem for the extended verification technique. We highlight the differences between Corollary 6.42 and the new Theorem 6.53 in bold:

Theorem 6.53 (Soundness of EVT): For any routine execution r on an instance o in a successfully type checked and verified Rumer program, if the invariants $\mathbb{X}_{(o,r)}$ hold in the initial state of r's execution, then:

(1) in all semi-visible states s_i of r's execution:
   (a) at most the invariants of those instances for which o is a responsible, **irresponsible**, or **quasi-irresponsible** invalidator are lost between the semi-visible states s_0 and s_i;
   (b) if s_i is a pre-state of a direct routine invocation r' on an instance o', the invariants $\mathbb{X}_{(o',r')}$ hold;

(2) in the final state s_{final} of r's execution: the invariants of those instances for which o is a responsible invalidator are preserved (since s_0) and also established for any of these instances for which o is a guarantor, and the invariants $\mathbb{X}_{(o,r)}$ hold.

Proof 6.54: We prove Theorem 6.53 by strong induction on the number k of transitive routine invocations made during the execution of a routine r. Let k be an arbitrary natural number of transitive routine invocations. By induction, we can assume that the theorem

holds for any natural number j of transitive routine invocations such that $j < k$ and we must show that it also holds for k transitive routine invocations. Since the proof relies on the soundness of the basic verification technique (Corollary 6.42) and remains generic with regard to the aspects specific to the extended verification technique, we collate several routine kinds (see Table 6.7) and consider the following two cases:

Case 1 (Action, method, and extent constructor): This case subsumes all routine kinds that are allowed to invoke other routines. According to Table 6.7, these routines are: application actions, relationship extent constructors and entity extent constructors, relationship extent methods and entity extent methods, non-interposed and interposed relationship element methods and entity element methods. For the execution of such a routine r on the instance o, we assume that the invariants $\mathbb{X}_{(o,r)}$ hold in r's initial state and show that consequences (1) and (2) hold for k transitive routine invocations made during r's execution:

(1): To prove (1), we proceed by secondary weak induction on the index i of semi-visible states of r's execution:

(i = 0): Base case:

(a) Immediate since no invariants can yet have been lost.

(b) Vacuous since the initial state is not a pre-state of a routine invocation.

(i = n + 1): Let n be an arbitrary natural number. By inner induction, we can assume that (1) holds in the n-th semi-visible state s_n and we must show that it holds in state s_{n+1}. There are two cases to be considered:

(n + 1 is odd): Control remains within r between s_n and s_{n+1}.

(a) By inner induction, we know that (1) (a) holds in s_n and, thus, that at most the invariants of those instances for which o is a responsible, irresponsible, or quasi-irresponsible invalidator are lost between the semi-visible states s_0 and s_n. By Corollary 6.44, we know further that at most the invariants of those instances for which o is a responsible or irresponsible invalidator are lost between s_n and s_{n+1}. Consequently, we can conclude that at most the invariants of those instances for which o is a responsible, irresponsible, or quasi-irresponsible invalidator are lost between the semi-visible states s_0 and s_{n+1}.

(b) If s_{n+1} is a pre-state of a direct routine invocation r' on an instance o', then we must show that the invariants $\mathbb{X}_{(o',r')}$ hold in s_{n+1}. As shown previously in part (1) (a), we know that at most the invariants of those instances for which the caller o is a responsible, irresponsible, or quasi-irresponsible invalidator are lost between the semi-visible states s_0 and s_{n+1}. In the proof of the soundness theorem for the basic verification technique (Corollary 6.42) we have already shown that the invariants of those instances for which the caller o is a responsible invalidator are either not expected by the callee o' or re-established before the routine

invocation. Furthermore, we have indirectly shown, by relating the invariants $\mathbb{X}_{(o',r')}$ expected by the callee to the invariants $\mathbb{X}_{(o,r)}$ expected by the caller, that the invariants of instances for which transitive callers of o' are responsible invalidators are either not expected by the callee o' or have been re-established before the invocation of r. Given the disjointness of responsible and irresponsible invalidators (see Proposition 6.38) and the fact that quasi-irresponsible invalidators are no invalidators (see Definition 6.46), it remains to be shown that the invariants of those instances for which the caller o or any of its (possibly transitive) callers is an irresponsible or quasi-irresponsible invalidator are either not expected by the callee o' or re-established before the routine invocation. From Lemma 6.47 it follows immediately that the invariants of those instances for which the caller o or any of its (possibly transitive) callers is an irresponsible or quasi-irresponsible invalidator are not expected by the callee o', and, thus, we are guaranteed that the invariants $\mathbb{X}_{(o',r')}$ hold in s_{n+1}.

(n + 1 is even): A direct routine invocation r' on an instance o' is made between s_n and s_{n+1}.

(a) By inner induction, we know that (1) holds in s_n and, thus, that at most the invariants of those instances for which the caller o is a responsible, irresponsible, or quasi-irresponsible invalidator are lost between the semi-visible states s_0 and s_n and that the invariants $\mathbb{X}_{o',r'}$ hold in s_n. Since there must be strictly fewer than k routine invocations made during the execution of r', we know, by outer induction, that Theorem 6.53 holds for the execution of r'. Since the invariants $\mathbb{X}_{o',r'}$ hold in s_n, we know further that (1) and (2) hold for r'. In particular, we know that at most the invariants of those instances for which the callee o' is a responsible, irresponsible, or quasi-irresponsible invalidator are lost between s_n and s_{n+1} and that the invariants of those instances for which o' is a responsible invalidator are preserved between s_n and s_{n+1} and also established in s_{n+1} for any of these instances for which o' is a guarantor. In the proof of the soundness theorem for the basic verification technique (Corollary 6.42) we have already shown that none of the invariants for which the callee o' is a responsible invalidator are lost between the semi-visible states s_n and s_{n+1}. Given the disjointness of responsible and irresponsible invalidators (see Proposition 6.38) and the fact that quasi-irresponsible invalidators are no invalidators (see Definition 6.46), it remains to be shown that either none of the invariants for which the callee o' is an irresponsible or quasi-irresponsible invalidator are lost between the semi-visible states s_n and s_{n+1} or that the caller o is a responsible, irresponsible, or quasi-irresponsible invalidator for the instances for which o' is an irresponsible or quasi-irresponsible invalidator. From Lemma 6.49 and Lemma 6.51 it follows immediately that the caller o is a responsi-

ble, irresponsible, or quasi-irresponsible invalidator for the instances for which o' is an irresponsible invalidator and that the caller o is an irresponsible or quasi-irresponsible invalidator for the instances for which o' is a quasi-irresponsible invalidator, respectively. Thus, we can conclude that at most the invariants of those instances for which the caller o is a responsible, irresponsible, or quasi-irresponsible invalidator are lost between the semi-visible states s_0 and s_{n+1}.

(b) Vacuous since state s_{n+1} is not a pre-state of a routine invocation.

(2): As shown previously in part (1), we know that at most the invariants of those instances for which o is a responsible, irresponsible, or quasi-irresponsible invalidator are lost between the semi-visible states s_0 and s_{final}. In the proof of the soundness theorem for the basic verification technique (Corollary 6.42) we have already shown that the invariants of those instances for which o is a responsible invalidator are preserved and also established for any of these instances for which o is a guarantor. Furthermore, we have indirectly shown, by considering the invariants $\mathbb{X}_{(o,r)}$ expected by o, that the invariants of instances for which a (possibly transitive) caller of o is a responsible invalidator are either not expected by o or established. Given the disjointness of responsible and irresponsible invalidators (see Proposition 6.38) and the fact that quasi-irresponsible invalidators are no invalidators (see Definition 6.46), it remains to be shown that the invariants of those instances for which o or any of its (possibly transitive) callers is an irresponsible or quasi-irresponsible invalidator are either not included in $\mathbb{X}_{(o,r)}$ or are established. Since o may be a caller of itself (see parameter $\mathbb{C}$, Table 6.7), it follows immediately from Lemma 6.47 that the invariants of those instances for which o or any of its (possibly transitive) callers is an irresponsible or quasi-irresponsible invalidator are not included in $\mathbb{X}_{(o,r)}$, and, thus, we can conclude that the invariants of those instances for which o is a responsible invalidator are preserved and also established for any of these instances for which o is a guarantor, and that the invariants $\mathbb{X}_{(o,r)}$ hold in s_{final}.

Case 2 (Default extent constructor, mold constructor, and mold initializer): This case subsumes all routine kinds that are not allowed to invoke other routines. According to Table 6.7, these routines are: default relationship extent constructors and default entity extent constructors, non-interposed relationship mold constructors and entity mold constructors, and non-interposed relationship mold initializers and entity mold initializers. For the execution of such a routine r on the instance o, we assume that the invariants $\mathbb{X}_{(o,r)}$ hold in r's initial state and show that consequences (1) and (2) hold for k transitive routine invocations made during r's execution:

(1): Since the routine r cannot invoke any other routines (see Table 6.7), we prove (1) by investigating the only two semi-visible states of r's execution:

(Initial state):

(a) Immediate since no invariants can yet have been lost.

(b) Vacuous since the initial state is not a pre-state of a routine invocation.

(Final state):

(a) Since s_{final} is the immediate successor state of s_0 and, thus, control remains within r between s_0 and s_{final}, we know by Corollary 6.44 that at most the invariants of those instances for which o is a responsible or irresponsible invalidator are lost between the semi-visible states s_0 and s_{final}.

(b) Vacuous since the final state is not a pre-state of a routine invocation.

(2): As shown previously in part (1), we know that at most the invariants of those instances for which o is a responsible or irresponsible invalidator are lost between the semi-visible states s_0 and s_{final}. In the proof of the soundness theorem for the basic verification technique (Corollary 6.42) we have already shown that the invariants of those instances for which o is a responsible invalidator are preserved and also established for any of these instances for which o is a guarantor. Furthermore, we have indirectly shown, by considering the invariants $\mathbb{X}_{(o,r)}$ expected by o, that the invariants of instances for which a (possibly transitive) caller of o is a responsible invalidator are either not expected by o or established. Given the disjointness of responsible and irresponsible invalidators (see Proposition 6.38) and the fact that quasi-irresponsible invalidators are no invalidators (see Definition 6.46), it remains to be shown that the invariants of those instances for which o is an irresponsible invalidator or for which any of its (possibly transitive) callers is an irresponsible or quasi-irresponsible invalidator are either not included in $\mathbb{X}_{(o,r)}$ or are established. Since o may be a caller of itself (see parameter $\mathbb{C}$, Table 6.7), it follows immediately from Lemma 6.47 that the invariants of those instances for which o or any of its (possibly transitive) callers is an irresponsible or quasi-irresponsible invalidator are not not included in $\mathbb{X}_{(o,r)}$, and, thus, we can conclude that the invariants of those instances for which o is a responsible invalidator are preserved and also established for any of these instances for which o is a guarantor, and that the invariants $\mathbb{X}_{(o,r)}$ hold in s_{final}.

□

6.4 Related work

In this section, we discuss the related work on object ownership as well as the one on object-oriented verification. We first briefly relate selective ownership to "traditional" forms of ownership and then provide an overview of object-oriented, invariant-based verification techniques.

6.4.1 Ownership

Work on "traditional" forms of ownership can broadly be categorized into work on ownership types [93, 94, 95, 97] and work on Universe Types [12, 96, 98, 99, 100]. In both ownership schemes, all objects are owned and have exactly one owning object.

The two schemes, however, differ in their applied encapsulation discipline. Whereas ownership types typically enforce the *owner-as-dominator* discipline, Universe Types typically enforce the *owner-as-modifier* discipline [96]. The owner-as-dominator discipline requires all reference chains to an object to pass through the object's owner. The owner-as-modifier discipline, on the other hand, enforces a less stringent alias restriction and only requires modifications of an object to be initiated by the object's owner. To compensate for their restrictiveness, ownership types allow owned objects to establish back-references to their owners. Universe Types, in contrast, forbid modifying reference chains to leave a context. This restriction makes the Universe Type system attractive for program verification since it forbids call-backs into owners. The amenability of Universe Types for program verification has been shown in [12, 14, 16]. The benefits of ownership for program verification have also been demonstrated in the context of Oval [110], a variant of an ownership-type-based language. As opposed to other ownership type systems [93, 94, 95, 97], Oval's types system enforces an owner-as-modifier discipline and employs effect annotations to deal with call-backs.

Selective ownership is most closely related to Universe-type-based ownership since it is similarly lightweight thanks to the use of ownership modifiers. However, as detailed in Section 6.1, selective ownership allows for the sharing of instances that are transitively reachable through owners. This relaxation is thanks to Rumer's separation between "heap topology" (achieved by the Matryoshka Principle, see also discussion in Section 4.4.2 on page 87) and unique ownership restriction and facilitates a hybrid ownership scheme in which owned and shared instances coexist.

6.4.2 Invariant-based verification techniques

A wide palette of techniques for the verification of object-oriented programs has been elaborated up to date. Most of these techniques rely on invariants [20] for program verification. More recently, techniques have been suggested that are not based on invariants but leverage forms of heap partitioning to verify programs in the presence of shared mutable state. Parkinson and Bierman [23, 24, 25] introduce the ideas of separation logic [22] to Java. An alternative expression of separation is used in works on dynamic frames [27, 28, 29, 30] 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 [25, 111].

In the remainder of this section we concentrate on object-oriented verification techniques that are based on invariants since our verification technique is also based on invariants. Presently, no other verification techniques for relationship-based programming languages have been devised. We first review invariant-based verification techniques that establish

a visible-state semantics for invariants and then review invariant-based verification techniques that are not based on a visible-state semantics for invariants. We begin the discussion of visible-state verification techniques with the ownership technique since this is the visible-state verification technique that is, despite the difference in underlying programming model, most closely related to our verification technique.

Visible-state verification techniques

Ownership technique. The ownership technique [12, 16] is a visible-state verification technique that employs Universe Types [12, 96, 98, 99, 100] to enforce a tree topology on program's heap. Based on this heap topology, the technique can accommodate not only single-object invariants but also certain forms of multi-object invariants (invariants at the one-side of one-to-many dependencies) at a small number of proof obligations. The proof obligations imposed by the ownership technique are relative to the declaring class, and the technique is thus entirely modular.

Our verification technique resembles the ownership technique in two aspects: it leverages heap stratification to prevent transitive call-backs and it facilitates modular reasoning about multi-object invariants. However, our verification technique differs from the ownership technique in the underlying heap stratification and in the separation between heap stratification and unique ownership. As discussed in Section 4.4.2 on page 87, the stratification induced by the Matryoshka Principle allows entity and relationship instances to participate in several relationship instances and to change their relationship participation at run-time. The separation between heap stratification and unique ownership, on the other hand, allows a programmer to treat selective ownership as an orthogonal mechanism that can be superimposed, if desired, on top of the heap stratification induced by the Matryoshka Principle. This possibility for customization is beneficial as it allows a programmer to choose the most appropriate combination of mechanisms, depending on the situation. As a result, our verification technique can accommodate a larger set of verification problems than the ownership technique (see examples in Section 7.1), but still at a small number of proof obligations. These proof obligations are modular, however, rely on the non-modular premise that a program's participants relation is acyclic (see System Invariant 4.1 on page 77) and that entity and relationship fields do not point to instances of a base type of which the declaring base type is a direct or transitive participant (see Section 6.2.3). In Section 7.2 on page 204, we discuss the elaboration of a module system as part of future work to improve the scalability of both the Rumer programming language and our verification technique. Such a module system would certainly also make those semantic checks modular.

In addition, our verification technique supports member interposition. Member interposition is orthogonal to selective ownership and can be applied, if desired, with or without selective ownership. For example, in Section 7.1.2 on page 193, we show two version of the Composite pattern where the first version combines both member interposition and selective ownership and the second version only uses member interposition. 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 further a slightly different semantics in terms of proof obligations: whereas ownership allows the declaration of an invariant-compromising method in an owned object in principle, member interposition prevents the declaration of such a method in the declaring relationship.

Remaining visible-state verification techniques. Lu et al. [110] present a verification technique based on their language Oval to facilitate the verification of multi-object invariants. Like the ownership technique [12, 16], Lu et al.'s technique also uses ownership to support modular reasoning about multi-object invariants. Unlike the ownership technique, Oval is based on a variant of ownership types [93, 94, 95, 97] that enforces an owner-as-modifier discipline. To cope with call-backs that may potentially arise from owned objects, Oval employs the notion of a "validity contract". Such a contract is declared by each method of a class and indicates the expected and vulnerable invariants for that a method. Method invocations are then constrained to adhere to the "validity subcontract rule" [110]. This rule ensures that the caller guarantees the invariants expected by the callee and that the invariants vulnerable to the callee are also vulnerable to the caller. Validity contracts provide some flexibility in terms of allowed method invocations, but at the cost of increased specification burden for the programmer. Drossopoulou et al. provide a comparison of the ownership technique and Lu et al.'s technique in [104].

Müller et al. [16] introduce the visibility technique, a visible-state verification technique that complements the ownership technique [12, 16] with visibility-based invariants. A visibility-based invariant is allowed to depend on fields of peer objects provided that those fields are visible to the declaring class and vice versa. The visibility technique also relaxes object encapsulation by allowing peer objects to assign to each others' fields. As a result, the visibility technique is capable of accommodating multi-object invariants over non-hierarchical object structures. Like the ownership technique, the visibility technique imposes modular (with regard to a class' visibility scope) proof obligations. However, the gain in expressiveness comes at the price of an increased number of proof obligations. Furthermore, the requirement for mutual visibility rules out invariants that are based on arrays or library classes.

Summers et al. [105] extend the visibility technique [16] with static invariants (i.e., per-class invariants). To support static invariants, the authors extend the Universe Type heap topology with multiple trees. Each tree is rooted in class that becomes the owner of its "representation objects". Static invariants are allowed to depend not only on the fields of their owned objects but also to quantify over all instances of the class. The authors present a basic version of their technique (visibility technique with statics (VTS)) that relies on an effect system to prevent dangerous call-backs between multiple trees through static method invocations. The technique introduces the proof obligation of invariant

preservation for static invariants, which requires a method to show that it does not break the invariant and is similar to our proof obligation (see Definition 5.11 on page 107). As discussed in Section 5.2, we adopt this proof obligation for extent invariants ands element invariants of malign relationships. Summers et al. further introduce two more permissive versions of VTS, strong VTS and layered VTS, which allow for certain call-backs between trees and for a reduction of the number of effect annotations, respectively.

In addition to the previously discussed, modular visible-state verification techniques, a number of non-modular techniques have been suggested:

Poetzsch-Heffter [10] introduces a verification technique for object invariants that soundly deals with multi-object invariants and call-backs. His technique neither restricts invariants nor field assignments and method invocations, yet comes at the price of extremely strong proof obligations. In particular, all invariants must be shown to hold in pre-states of method invocations and in final states of method executions. Those proof obligations in turn guarantee that all invariants hold in visible states.

Huizing and Kuiper [11] introduce a slightly less permissive but less costly verification technique that restricts field assignments to the current receiver instance and reduces the number of proof obligations. The latter is achieved by a syntactic analysis that determines the set of invariants that are invalidated by field updates.

Techniques with a relaxed visible-state semantics. More recently, two verification techniques have been suggested that employ a relaxed visible-state semantics to accommodate multi-object invariants over non-hierarchical object structures:

Middelkoop et al. [18] present a verification technique that relies on a refined form of data type induction. This refined form supports update patterns that span nested method executions by allowing certain methods to be executed in states where certain invariants are broken. To delimit the methods and invariants for which no visible-state semantics should be assumed, the authors introduce two special specification constructs “inc” and “coop”. The inc and coop constructs denote, for a method and field, respectively, the set of invariants that the method preserves but does not expect and the set of invariants that may be broken due to assignments to the field, respectively.

Summers and Drossopoulou [19] extend Middelkoop et al.’s work [18] and introduce Considerate Reasoning, a verification technique that 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. Summers and Drossopoulou refine Middelkoop et al.’s coop set to concerns descriptions that are bounded by supporting invariants. Furthermore, the authors provide the encoding of their technique in the Boogie intermediate verification language [112], facilitating the automatic verification of their running example.

Boogie methodology

The Boogie methodology describes a verification technique [13] for the Spec♯ programming system [85] that allows the modular verification of Spec♯ programs. As opposed to the verification techniques discussed in the previous sections, the Boogie methodology does not employ a visible-state semantics. Instead, programmers are required to delimit the code blocks during which an object supposedly falsifies its invariant. Spec♯ provides the block statement `expose` for that purpose, and programmers can only modify an object by exposing the object beforehand. Using an explicit specification of when an object is expected to satisfy its invariant avoids the possible unsoundness with regard to callbacks that visible-state verification techniques typically face.

Various extensions to the Boogie methodology have been introduced that allow the expression of multi-object invariants. Leino and Müller [14] incorporate Universe-type-based ownership [12, 96] into the Boogie methodology and contribute a dynamic ownership model that allows for ownership transfer. Furthermore, the authors describe visibility-based invariants to accommodate multi-object invariants over non-hierarchical object structures. The idea of enlarging an invariant's scope of visibility has been taken further by Barnett and Naumann [15] and Leino and Schulte [17]. Both techniques extend the Boogie methodology to facilitate the verification of Observer-like patterns (see discussion in Section 7.1.3).

7 Evaluation

In this chapter, we illustrate the concepts introduced in the previous chapters on a number of examples. All examples employ Design-by-Contract-style assertions, including invariants. We show that those specifications are admissible by our verification technique and discuss the proof obligations that our verification technique generates for the verification of those specifications. We first provide the evaluation of this thesis work based on the examples (Section 7.1) and then conclude with a discussion of open challenges and possible directions for future work (Section 7.2).

7.1 Addressed challenges

In this section, we demonstrate the Rumer programming language and our verification technique on a number of examples.

7.1.1 University example

The university example introduced earlier (see Section 2.1.1 on page 8 and Figure 3.3 on page 43) provides ample opportunities for the specification of invariants and routine preconditions and postconditions. We consider, in turn, element and extent invariants both for entities and relationships as well as an ownership-based application invariant.

Figure 7.1 displays the declaration of entity `Student` augmented with an element invariant (line 22) and an extent invariant (line 23). The element invariant requires a student's `id` to be a positive integer and the extent invariant requires a student's `id` to uniquely identify a `Student` element instance in its extent instance. Both the element invariant and the extent invariant are admissible according to the basic verification technique (see Table 4.4 on page 82) since they only depend on locations of the invariants' instances. Entity `Student` declares the mold initializer `initialize()` (see Section 3.4.1 on page 65), which returns a `Student` mold instance whose fields are initialized with the arguments passed to the initializer. Even though mold constructors and mold initializers do not obey a visible-state semantics, `initialize()` declares preconditions and postconditions. These preconditions and postconditions are devised so that `initialize()`

```
1  entity Student {
2    string name;
3    int id;
4    string language;
5
6    init initialize(string name, int id, string language) // mold initializer
7      requires
8        id > 0;
9      ensures
10       this.name == name & this.id == id & this.language == language;
11   { this.name = name; this.id = id; this.language = language; }
12
13   extent Student createStudent(string name, int id, string language)
14     requires
15       id > 0 &
16       !(id isItemOf these.id);
17     ensures
18       result.name == name & result.id == id & result.language == language &
19       these == old(these) union Set(result);
20   { return these.add(new Student().initialize(name, id, language)); }
21
22   invariant this.id > 0; // element invariant
23   extent invariant forAll(s isElementOf these: // extent invariant
24     these.select(other: other.id == s.id).count() == 1); // id is unique
25 }
26
27 application University {
28   Extent<Student> students = new Extent<Student>();
29
30   main() {
31     if (!(1 isItemOf students.id)) {
32       students.createStudent("Susan", 1, "French");
33     }
34   }
35 }
```

Figure 7.1: Entity `Student` of university program shown in Figure 3.3 on page 43 augmented with an element invariant, extent invariant, and routine preconditions and postconditions.

can meet the `add()` operator's precondition. As detailed by Table 3.17 on page 68, an `add()` operator requires that the fields of the mold instance passed as argument satisfy the restrictions imposed by an element invariant on the corresponding fields of an element instance.

Entity `Student` declares the extent method `createStudent()`. This method must obey a visible-state semantics, and, according to Table 6.13 on page 167, the method can expect the following invariants to hold in its visible states (parameter $\mathbb{X}$): the invariants of all instances *sx* of type Extent⟨Element⟨Student⟩⟩ (including the current receiver instance `these`) as well as the invariants of all instances *s* of type Element⟨Student⟩ comprised in those instances *sx*. In addition, method `createStudent()` can expect the extent and element invariants of all entity and relationship types of which `Student` is not a direct or transitive participant. These types encompass the entities `Course` and `Faculty` as well as the relationship `Teaches`. The invariants vulnerable to the execution of the extent method `createStudent()` (parameter $\mathbb{V}$) are, according to Table 6.13, the invariant

```
1  relationship Teaches participants (Faculty prof, Course course) {
2
3    extent void makeFacultyTeachCourse(Faculty f, Course c)
4      requires
5        f != null & c != null &
6        !(c isElementOf these.course);
7      ensures
8        these == old(these) union Set(Pair(f, c));
9    { these.add(new Teaches(f, c)); }
10
11   extent invariant these.inverse().isPartialFunction(); // extent invariant
12 }
```

Figure 7.2: Relationship `Teaches` of university program shown in Figure 3.3 on page 43 augmented with an extent invariant and method preconditions and postconditions.

of the current receiver instance `these`, the invariants of all element instances comprised in `these`, and the invariant of the owner of `these`. Should the ownership be inclusive, then System Invariant 6.3 on page 142 guarantees that both the extent instance and its element instances have the same owner. The proof obligation $\mathbb{B}$ is empty for extent method `createStudent()` since the method does not invoke any routines[1]. In the final state of its execution, the extent method `createStudent()` has to prove that the invariant of its current receiver instance `these` as well as the invariants of all element instances comprised in `these` hold. Since the method is asserted that those invariants hold in the method's initial state and, given the preconditions of `createStudent()` and the postcondition of the built-in `add()` operator (see Table 3.17 on page 68), the method should be in the position to show that those invariants also hold in the method's final state. Furthermore, by relying on the postcondition of the mold initializer `initialize()` and the built-in `add()` operator, the extent method should be in the position to guarantee its postcondition.

Figure 7.1 also displays an excerpt of the application declaration `University`. In its `main()` action, the application invokes the extent method `createStudent()` on the `Student` extent instance created upon start-up (line 28), provided that no `Student` element instance with the `id` 1 exists in that extent instance. According to Table 5.5 on page 97, the `main()` action can expect the invariant of the `University` application singleton instance as well as all extent and element invariants of all types declared by the university program[2]. These types encompass the entities `Student`, `Course` and `Faculty` as well as the relationships `Attends`, `Assists`, and `Teaches`. As the `University` application does not declare an invariant in Figure 7.1, the proof obligations $\mathbb{B}$ and $\mathbb{E}$ are empty for the `main()` action.

Figure 7.2 displays the declaration of relationship `Teaches` augmented with an extent invariant (line 11). The invariant requires that the inverse of the relation described by the invariant's `Teaches` extent instance forms a partial function. The invariant thus

[1] Our verification technique treats invocations of the `add()` and `remove()` operator as statements (see Section 4.3 on page 78).

[2] The parameters $\mathbb{X}$, $\mathbb{V}$, $\mathbb{B}$, and $\mathbb{E}$ for application actions are the same for BVT and EVT.

guarantees that a course can be taught by at most one faculty member, satisfying the UML multiplicity constraint imposed on association teaches in Figure 2.1 on page 8. This invariant is admissible according to the basic verification technique (see Table 4.4 on page 82) since it only depends on locations of the invariant's extent instance.

`Teaches` further declares the extent method `makeFacultyTeachCourse()`. This method creates a new `Teaches` element instance that relates the `Faculty` and `Course` element instances provided as argument to the extent method. According to Table 6.13 on page 167, the extent method `makeFacultyTeachCourse()` can expect the following invariants to hold in its visible states (parameter $\mathbb{X}$): the invariants of all instances *tx* of type Extent⟨Element⟨Teaches⟩⟩ (including the current receiver instance `these`) as well as the invariants of all instances *t* of type Element⟨Teaches⟩ comprised in those instances *tx*. In addition, method `makeFacultyTeachCourse()` can expect the extent and element invariants of all entity and relationship types of which `Teaches` is not a direct or transitive participant. These types encompass the entities `Student`, `Course`, and `Faculty` as well as the relationship `Attends` and `Assists`. The invariants vulnerable to the execution of the extent method `createStudent()` (parameter $\mathbb{V}$) are, according to Table 6.13, the invariant of the current receiver instance `these`, the invariants of all element instances comprised in `these`, and the invariant of the owner of `these`. The proof obligation $\mathbb{B}$ is empty for extent method `makeFacultyTeachCourse()` as the method does not invoke any routines. In the final state of its execution, the extent method `makeFacultyTeachCourse()` has to prove that the invariant of its current receiver instance `these` as well as the invariants of all element instances comprised in `these` hold. Given the assertion that those invariants hold in the initial state of method `makeFacultyTeachCourse()` and given the precondition that the `Course` element instance referred to by the argument `c` is not yet related by a `Teaches` element instance residing in `these` (line 6), method `makeFacultyTeachCourse()` should be able to show that those invariants also hold in its final state.

Figure 7.3 shows the declaration of relationship `Attends`. The relationship declares the non-interposed element field `grade`, which the element invariant on line 19 restricts to range between 1 and 6. As the invariant only depends on locations of the invariant's element instance, it is admissible according to the basic verification technique (see Table 4.4 on page 82).

The element method `setGrade()` updates the non-interposed element field `grade`. Relationship `Attends` is an instance of a free relationship (see Table 5.4 on page 96) since it neither declares an interposed element field nor imposes an element invariant on such a field. The framework parameters for method `setGrade()` are thus defined by Table 6.14 on page 167. According to Table 6.14, the element method `setGrade()` can expect the invariant of the current receiver instance `this` in its visible states (parameter $\mathbb{X}$) as well as the extent and element invariants of all entity and relationship types of which `Attends` is not a direct or transitive participant. These types encompass the entities `Student`, `Course`, and `Faculty` as well as the relationships `Assists` and

```
1  relationship Attends participants (Student learner, Course course) {
2    int grade;
3
4    void setGrade(int grade)
5      requires
6        1 <= grade & grade <= 6;
7      ensures
8        this.grade == grade;
9    { this.grade = grade; }
10
11   extent void registerStudentForCourse(
12       query Set<Student> sts, query Set<Course> cs) // query method arguments
13     requires
14       ((sts cartesianProduct cs) intersection these).isEmpty();
15     ensures
16       sts cartesianProduct cs isSubsetOf these;
17   { these.add(new Attends(sts, cs)); }
18
19   invariant 1 <= this.grade & this.grade <= 6; // element invariant
20 }
21
22 application University {
23   Extent<Student> students = new Extent<Student>();
24   Extent<Course> courses = new Extent<Course>();
25   Extent<Attends> attends = new Extent<Attends>();
26
27   main() {
28     ...
29     let csStudents be students.select(s: s.id < 1000);
30     let csCourses be courses difference courses.select(c: c.title == "Art");
31     attends.registerStudentForCourse(csStudents, csCourses);
32   }
33 }
```

Figure 7.3: Relationship `Attends` of university program shown in Figure 3.3 on page 43 augmented with an element invariant and method preconditions and postconditions.

`Teaches`. The invariants that are vulnerable to the execution of the element method `setGrade()` (parameter $\mathbb{V}$) are the invariant of the current receiver instance `this`, the invariant of the extent instance in which the current receiver instance resides, and the invariant of the owner of the current receiver instance. The proof obligation $\mathbb{B}$ is empty for the element method `setGrade()` as the method does not invoke any routines. In the final state of its execution, the element method `setGrade()` has to prove that the invariant of its current receiver instance `this` holds and that it preserves the invariant of the extent instance in which its current receiver instance resides. The latter is empty and, given the assertion that the invariant of `setGrade()`'s current receiver instance holds in the initial state and given `setGrade()`'s precondition, `setGrade()` should be able to show that the invariant of its current receiver instance holds also in the final state.

Relationship `Attends` declares the extent method `registerStudentForCourse()` to populate the current extent receiver instance with `Attends` relationship instances. The framework parameters are analogous to the framework parameters of the extent method `makeFacultyTeachCourse()` of relationship `Teaches` (see Figure 7.2). The arguments of the extent method `registerStudentForCourse()` are notewor-

```
1  application University {
2    owned Extent<Student> students = new owned Extent<Student>();
3    owned Extent<Attends> attends = new owned Extent<Attends>();
4    owned Extent<Assists> assists = new owned Extent<Assists>();
5    Extent<Course> courses = new Extent<Course>();
6
7    main() {
8      Course prog = courses.createCourse("Programming");
9      Course art = courses.createCourse("Art");
10     Student susan = enroll("Susan", 1, "French", "Programming");
11     assignTa(susan, art);
12   }
13
14   Student enroll(string name, int id, string lang, string title)
15     requires
16       id > 0 & !(id isItemOf students.id) & title isItemOf courses.title;
17     ensures
18       result.name == name & result.id == id & result.language == lang &
19       students == old(students) union Set(result) &
20       students.select(s: s == result) cartesianProduct
21         courses.select(c: c.title == title) isSubsetOf attends;
22   {
23     Student newStudent = students.createStudent(name, id, lang);
24     attends.registerStudentForCourse(students.select(s: s == newStudent),
25       courses.select(c: c.title == title));
26     return newStudent;
27   }
28
29   void assignTa(Student st, Course c)
30     requires
31       st != null & c != null &
32       !(Pair(st,c) isElementOf attends) & !(Pair(st,c) isElementOf assists);
33     ensures
34       assists == old(assists) union Set(Pair(st,c));
35   { assists.assignTaToCourse(st, c); }
36
37   // ownership-based application invariant
38   invariant attends.learner == students; // attends is total on student
39   invariant (attends intersection assists).isEmpty();
40 }
```

Figure 7.4: Application `University` of university program shown in Figure 3.3 on page 43 augmented with ownership-based invariants and action preconditions and postconditions.

thy. The method takes query expressions (see Section 3.2.3 on page 49) as arguments, whose result sets are stored upon relationship mold instance creation. As indicated by Table 3.16 on page 67, an `Attends` mold instance is created for every element in the cross product of the two argument sets. The resulting bag of `Attends` mold instances is passed to the `add()` operator, which creates an `Attends` relationship element instance for each mold instance in the argument bag (see Table 3.18 on page 69). Figure 7.3 shows also an excerpt of the application declaration `University`, which invokes `registerStudentForCourse()` on line 31 and passes as arguments the predicate references declared on line 29 and line 30.

As a last illustration, we extend the `University` application with ownership-based invariants to accommodate inter-relationship invariants. Figure 7.4 shows the resulting

application declaration. The invariants on line 38 and 39 guarantee, for the university's extent instances, that students are enrolled in at least one course and that students cannot attend the same courses that they assist, respectively. Both invariants are admissible according to the extended verification technique (see Table 6.9 on page 164). As the `University` application owns the extent instances `students`, `attends`, and `assists`, the invariants are allowed to refer to the content of those extent instances (see Table 6.10 on page 165).

In addition to the mandatory `main()` action, the `University` application declares the actions `enroll()` and `assignTa()`. According to Table 5.5 on page 97, those actions can expect the invariants of the `University` application singleton instance as well as all extent and element invariants of all types declared by the university program. These types encompass the entities `Student`, `Course` and `Faculty` as well as the relationships `Attends`, `Assists`, and `Teaches`. The application invariants declared on line 38 and 39 are vulnerable (parameter $\mathbb{V}$) to the execution of all those actions. These are also the invariants that have to be proven to hold before another action is invoked on the `University` application singleton instance (parameter $\mathbb{B}$) and in the final state of an action execution (parameter $\mathbb{E}$). We review those proof obligations for each action declaration in Figure 7.4.

The action `assignTa()` creates a new `Assists` element instance that relates the `Student` and `Course` element instances provided as arguments. For its implementation, it relies on `Assists`'s extent method `assignTaToCourse()` (see specification of `assignTaToCourse()` in Figure 7.5). The invocation of `assignTaToCourse()` on line 35 is ownership-admissible since the singleton application instance is the owner of the extent instance `assists`. Since the action `assignTa()` does not invoke any actions during its execution, the proof obligation $\mathbb{B}$ is empty. In the final state of its execution, action `assignTa()` has to prove that the `University` application invariants declared on line 38 and 39 hold. To establish those invariants, `assignTa()` requires as a precondition that the argument student-course pair is not yet related by any `Attends` element instance contained in `attends`. Given this precondition and the postcondition of method `assignTaToCourse()`, the action `assignTa()` is guaranteed not to break the invariant declared on line 39. By relying on the assertion that the application invariants hold in the action's initial state, action `assignTa()` should be able to prove that those invariants hold also in its final state.

The action `enroll()` creates a new `Student` element instance as well as a new `Attends` element instance that relates the newly created `Student` element instance to the `Course` element instance with the title passed as argument[3]. For its implementation, action `enroll()` relies on `Student`'s extent method `createStudent()` (see Figure 7.1) as well as on `Attends`'s extent method `registerStudentForCourse()` (see Figure 7.3). Both invocations of these methods are ownership-admissible since

[3]Should the course title not be unique, then corresponding `Attends` relationship element instances are created for each `Course` element instance with a matching title.

```
1  entity Course {
2    string title;
3
4    extent Course createCourse(string title)
5      ensures
6        result.title == title & these == old(these) union Set(result);
7  }
8
9  relationship Assists participants (Student ta, Course course) {
10   extent void assignTaToCourse(Student s, Course c)
11     requires
12       s != null & c != null & !(Pair(s, c) isElementOf these);
13     ensures
14       these == old(these) union Set(Pair(s, c));
15 }
```

Figure 7.5: Method specifications for entity `Course` and relationship `Assists` of university program shown in Figure 3.3 on page 43 .

the singleton application instance is the owner of the extent instances `students` and `assists`. The proof obligation $\mathbb{B}$ is empty for action `enroll()` as the action does not invoke any actions during its execution. During its execution, the action `enroll()` may break `University`'s invariants. The action does in fact break the invariant declared on line 38 by invoking the extent method `createStudent()` on line 23. After the invocation of that method, there exists a `Student` element instance in the `Student` extent instance `students` that is not related to a `Course` element instance by the `Attends` extent instance `attends`. To re-establish the broken invariant, `enroll()` invokes the extent method `registerStudentForCourse()` on the extent instance `attends` to enroll the newly created student to the `Course` element instance with the title passed as argument. Given the assertion that the application invariants hold in the action's initial state, the action `enroll()` should be in the position to prove that the application invariants also hold in its final state.

The proof obligation $\mathbb{B}$ for the `main()` action of application `University` is more interesting. Before the invocations of actions `enroll()` and `assignTa()` on line 10 and 11, respectively, the `main()` action has to prove that the application invariants hold. Since no ownership is declared for the `Course` extent instance `courses`, the invocations of extent method `createCourse()` on line 8 and 9 do not break the application invariants (see specification of `createCourse()` in Figure 7.5). As a result, those invariants still hold in the pre-state of the action invocation `enroll()` (on line 10). Furthermore, since `enroll()` guarantees that the application invariants hold in the final state of its execution, the invariants also hold in the pre-state of the action invocation `assignTa()` (on line 11). And finally, since `assignTa()` guarantees that the application invariants hold in the final state of its execution, the `main()` action should be able to prove that the application invariants hold in the final state of `main()`'s execution as well (parameter $\mathbb{E}$).

The reviewed examples also illustrate that our verification technique imposes modular

```
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 }
```

Figure 7.6: Object-oriented implementation (w/o contracts) of the Composite pattern in a Java-like language as suggested in [87].

proof obligations. As the proof obligations $\mathbb{B}$ and $\mathbb{E}$ are relative to an instance of the declaring type, each type declaration of the university program can be verified independently from the remaining type declarations.

7.1.2 Composite pattern

In this section, we provide the full specification of the Composite pattern program introduced earlier (see Section 3.2.2 on page 44 as well as Figure 3.5 on page 45 and Figure 3.11 on page 57). The Composite pattern has been suggested recently as a verification challenge [87, 19] and served as the "Challenge Problem" for the 7th International Workshop on Specification and Verification of Component-Based Systems (SAVCBS).

Figure 7.6 shows an object-oriented implementation of the Composite pattern as suggested in [87]. In accordance with the UML class diagram shown in Figure 3.4 on page 45, 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) which indicates the total number of children components that can be reached from a component. The SAVCBS challenge problem 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 method `addComponent()` is invoked as well as the invariants of all its transitive parent composites are re-established. However, for the duration of the recursive invocations of method `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.

Figure 7.7 and Figure 7.8 give the complete specification of the Composite pattern in Rumer. The specification captures not only the SAVCBS invariant regarding a composite's `total` field (invariant on line 26 in Figure 7.7) but also provides a precise definition of a composite's tree properties. As detailed in Section 3.2.5 on page 55, the first two conjuncts of `Parent`'s extent invariant (line 24 in Figure 7.7) guarantee that the relation described by a `Parent` extent instance forms a forest of trees. `Composite`'s element invariant (line 54 in Figure 7.8) then restricts a `Composite` element instance's tree `Parent` extent instance from a forest of trees to a tree. `Parent`'s extent invariant is admissible according to the basic verification technique (see Table 4.4 on page 82). It only depends on the content of the invariant's `Parent` extent instance as well as on the interposed fields `total` of the element instances residing in that extent instance. `Composite`'s element invariant, on the other hand, is admissible according to the extended verification technique (see Table 6.9 on page 164). As detailed in Section 6.3.1 on page 164, a `Composite` element instance is the owner of its related `Parent` extent instance, and, hence, the `Composite` element invariant is allowed to depend not only on its instance's locations but also on the owned `Parent` extent instance's locations.

The extent method `append()` of relationship `Parent` (line 4 in Figure 7.7) appends the argument component `c` as child of the argument component `p`. The method instantiates a `Parent` element instance that relates the components `c` and `p` by adding a new `Parent` mold instance to the current extent receiver instance (line 15). The loop on line 17 increments the `total` field of all transitive parent components of the child component `c`. According to Table 6.13 on page 167, the method can expect the following invariants to hold in its visible states (parameter $\mathbb{X}$): the invariants of all instances *px* of type Extent⟨Element⟨Parent⟩⟩ (including the current receiver instance `these`) as well as the invariants of all instances *p* of type Element⟨Parent⟩ comprised in those instances *px*. In addition, method `append()` can expect the extent and element invariants of all entity and relationship types of which `Parent` is not a direct or transitive participant. These types encompass the entity `Component`. Since the extent method `append()` expects the invariant of its current receiver instance, the update of the `total` field of all affected parent components must actually happen in a loop, as opposed to a recursive invocation. The invariants vulnerable to the execution of the extent method `append()` (parameter $\mathbb{V}$) are, according to Table 6.13, the invariant of the current receiver instance `these`, the

```
1  relationship Parent participants (Component child, Component parent) {
2    int >parent total;
3
4    extent void append(Component c, Component p)
5      requires
6        c != null & p != null & c != p &
7        !(c isElementOf these.child union these.parent) &
8        (!these.isEmpty() => p isElementOf these.child union these.parent);
9      ensures
10       these == old(these) union Set(Pair(c, p)) &
11       forAll(x isElementOf
12         these.transitiveClosure().select(cp: cp.child == c).parent:
13           x.total == old(x.total) + 1);
14   {
15     these.add(new Parent(c, p));
16     // increments total of all transitive parents of c
17     foreach (x isElementOf
18       these.transitiveClosure().select(cp: cp.child == c).parent)
19     { x.total = x.total + 1; }
20   }
21
22   extent invariant
23     // these is a forest of trees
24     these.isPartialFunction() & these.transitiveClosure().isIrreflexive() &
25     // SAVCBS challenge invariant
26     forAll(p isElementOf these.parent:
27       p.total == these.transitiveClosure().select(cp: cp.parent == p).count());
28 }
```

Figure 7.7: Complete Rumer Composite pattern specification for `Parent` relationship.

invariants of all element instances comprised in `these`, and the invariant of the owner of `these`. Since relationship `Parent` does not declare an element invariant, only the extent invariant declared on line 22 in Figure 7.7 and the invariant of the current receiver instance's owner are vulnerable to the execution of the extent method `append()`. The proof obligation $\mathbb{B}$ is empty for extent method `append()` as the method does not invoke any routines. In the final state of its execution, the extent method `append()` has to prove that the invariant of its current receiver instance `these` as well as the invariants of all element instances comprised in `these` hold. The latter is empty, and given the assertion that the extent invariant holds in the initial state and given the precondition of `append()` as well as the fact that `append()` updates the `total` field of all transitive parent components of the child component `c`, the extent method `append()` should be able to show that its current receiver's invariant also holds in the final state.

The extent method `createComposite()` of relationship `Composite` (line 4 in Figure 7.8) instantiates a `Composite` element instance by adding a new `Composite` mold instance to the current extent receiver instance of the method. The new `Composite` element instance has the component `c` as a `root` participant and an empty `Parent` extent instance as a `tree` participant. As discussed in Section 6.2.2 on page 141, the newly created `Composite` element instance is the owner of its tree `Parent` extent instance. The framework parameters for the extent method `createComposite()` are defined by Table 6.13 on page 167 and, thus, they are analogous to the parameters of

```
1  relationship Composite
2    participants (Component root, owned Extent<Parent> tree) {
3
4    extent Composite createComposite(Component c)
5      requires c != null;
6      ensures
7        result.root == c & result.tree.isEmpty() &
8        these == old(these) union Set(result);
9    { return these.add(new Composite(c, new owned Extent<Parent>())); }
10
11   void appendComposite(Composite c, Component p)
12     requires
13       c != null & p != null & c.root != p &
14       !(c.root isElementOf this.tree.child union this.tree.parent) &
15       (!this.tree.isEmpty() =>
16         p isElementOf this.tree.child union this.tree.parent) &
17       (this.tree.isEmpty() => p == this.root) &
18       ((this.tree.child union this.tree.parent) intersection
19         (c.tree.child union c.tree.parent)).isEmpty();
20     ensures
21       this.tree == old(this.tree) union Set(Pair(c.root, p)) union c.tree;
22   { this.appendComponent(c.root, p); this.appendSubComposite(c.tree, c.root); }
23
24   void appendComponent(Component c, Component p)
25     requires
26       c != null & p != null & c != p &
27       !(c isElementOf this.tree.child union this.tree.parent) &
28       (!this.tree.isEmpty() =>
29         p isElementOf this.tree.child union this.tree.parent) &
30       (this.tree.isEmpty() => this.root == p);
31     ensures
32       this.tree == old(this.tree) union Set(Pair(c, p));
33   { this.tree.append(c, p); }
34
35   void appendSubComposite(query Set<Parent> c, Component p)
36     requires
37       p != null &
38       !(p isElementOf c.child) &
39       (!c.isEmpty() => p isElementOf c.parent) &
40       (c.child == c.transitiveClosure().select(cp: cp.parent == p).child) &
41       (c.child intersection
42         (this.tree.child union this.tree.parent)).isEmpty() &
43       (!this.tree.isEmpty() =>
44         p isElementOf this.tree.child union this.tree.parent) &
45       (this.tree.isEmpty() => this.root == p);
46     ensures
47       this.tree == old(this.tree) union c;
48   { foreach (cp isElementOf c.select(x: x.parent == p)) {
49       this.appendComponent(cp.child, cp.parent);
50       this.appendSubComposite(c.select(x: x.child isElementOf
51        c.transitiveClosure().select(y: y.parent == cp.child).child), cp.child);
52     }}
53
54   invariant // this.tree is a tree with the root this.root
55     !(this.root isElementOf this.tree.child) &
56     (!this.tree.isEmpty() => this.root isElementOf this.tree.parent) &
57     this.tree.transitiveClosure().select(cp: cp.parent == this.root).child ==
58       this.tree.child;
59 }
```

Figure 7.8: Complete Rumer Composite pattern specification for `Composite` relationship.

Parent's extent method append(), detailed previously. Unlike append(), however, createComposite() can expect not only the element and extent invariants of its declaring type Composite but also the element and extent invariants of relationship Parent and entity Component.

The element method appendComposite() (line 11 in Figure 7.8) appends the composite c to the current receiver instance as a child of component p. The method thus allows a programmer to extend a composite at any point in its tree. The method first invokes the element method appendComponent() on its current receiver instance to append the root component of composite c to the current receiver instance's tree extent instance as a child of p. Then, it invokes the element method appendSubComposite() on the current receiver instance to append the sub-composite rooted at c's root component as a child of the previously inserted root component. The framework parameters for the element method appendComposite() are defined by Table 6.14 on page 167. Since relationship Composite neither declares an interposed element field nor imposes an element invariant on such a field, it is an instance of a free relationship (see Section 5.2 on page 91 and Table 5.4 on page 96). Method appendComposite() can hence expect (parameter $\mathbb{X}$) the invariant of its current Composite element receiver instance in its visible states as well as the element and extent invariants of relationship Parent and entity Component. The invariants vulnerable to the execution of method appendComposite() (parameter $\mathbb{V}$) are the invariant of the current receiver instance this, the invariant of the extent instance in which the current receiver instance resides, and the invariant of the owner of the current receiver instance. Since Composite does not declare an extent invariant, the invariant of the extent instance in which the current receiver instance resides is not vulnerable to the execution of appendComposite(). For both method invocations, the invariant of the current receiver instance this has to be shown to hold before the call (parameter $\mathbb{B}$). Those proof obligations should be easily met since the invariant of the current receiver instance is asserted to hold in the visible states of Composite's element methods. Analogously, it should be straightforward to show that the invariant of the current receiver instance holds in the final state of the element method appendComposite() (parameter $\mathbb{E}$).

The element method appendComponent() (line 24 in Figure 7.8) invokes the extent method append() with the argument components c and p on the current receiver instance's tree extent instance. The invocation is ownership-admissible since the current receiver instance is the owner of its tree extent instance. As a result, the component c is appended as a child of the component p in the composite's tree extent instance. The element method appendComponent() has the same set of invariants $\mathbb{X}$ and $\mathbb{V}$ as the element method appendComposite(). The proof obligation $\mathbb{B}$ is empty for appendComponent() since the receiver instance this.tree of the method invocation on line 33 is different from the current receiver instance. Based on the assertion that the current receiver instance's invariant holds in the initial state and given the precondition of appendComponent() as well as the postcondition of append(), method appendComponent() should be able to show that its current receiver instance's in-

variant also holds in the final state.

The element method `appendSubComposite()` (line 35 in Figure 7.8) appends the sub-composite denoted by the query expression `c` to the current receiver instance's tree extent instance as a child of component `p`. The method is implemented recursively to append the sub-composite in a depth-first traversal order. In each recursive invocation, one child component of the sub-composite is appended to its corresponding parent component in the current receiver instance's tree extent instance. 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. Method `appendSubComposite()` has the same set of invariants $\mathbb{X}$ and $\mathbb{V}$ as the methods `appendComposite()` and `appendComponent()`. For both method invocations on line 49 and on line 50, the invariant of the current receiver instance `this` has to be shown to hold before the call (parameter $\mathbb{B}$). For the invocation on line 49, the invariant is asserted to hold for the first loop iteration and guaranteed to be established by `appendSubComposite()` for the remaining loop iterations. For the invocation on line 50, the invariant is guaranteed to be established by the previous statement and thus to hold in the pre-state of the invocation. Given the preconditions on line 38–40 (guarantee that `c` is empty or a tree with the root `p`) as well as the precondition on line 41–42 (prevents cycles with regard to `these`'s tree extent instance), `appendSubComposite()` should be able to show that the invariant of its current receiver instance holds in the final state (parameter $\mathbb{E}$).

The specification of the Composite pattern in Rumer shown in Figure 7.7 and Figure 7.8 demonstrates that the SAVCBS challenge invariant can be encapsulated in a relationship invariant using member interposition and verified employing our basic, visible-state verification technique presented in Chapter 5. The example also demonstrates that the Rumer specification is able to succinctly express the tree properties of the Composite pattern. To specify and verify the invariant of relationship `Composite`, which guarantees that a composite's tree extent instance forms indeed a tree and not a forest, selective ownership is required. Thus, a composite's tree invariant can be verified using our extended, visible-state verification technique presented in Chapter 6.

The ownership restriction necessary to verify the tree invariant of the Composite pattern shown in Figure 7.7 and Figure 7.8 is minimal. It only encompasses a composite's tree `Parent` extent instance, but not the `Parent` element instances comprised in that extent instance, nor the `Component` element instances related by those `Parent` element instances. If this form of ownership is still too restrictive, an alternative specification of the Composite pattern is conceivable. Figure 7.9 shows the alternative specification of the Composite pattern in Rumer. This specification represents a composite by a `Composite` *extent* instance and not, as the previous specification, by a `Composite` *element* instance. Such a `Composite` extent instance keeps a reference to its root component (extent field `root` declared on line 2), and the special case of a composite existing only of a root component is represented by an empty `Composite` extent instance. Since this alternative specification does not introduce an indirection as the specification shown Figure 7.7

```
1  relationship Composite participants (Component child, Component parent) {
2    extent Component root; // denotes root of composite
3    int >parent total;
4
5    Extent<Composite>(Component c)
6      requires
7        c != null;
8      ensures
9        these.root == c & these.isEmpty();
10   { these.root = c; }
11
12   extent void append(Component c, Component p)
13     requires/ensures // analogous to append() in Figure 7.7
14   {
15     these.add(new Composite(c, p));
16     foreach (x isElementOf
17       these.transitiveClosure().select(cp: cp.child == c).parent)
18     { x.total = x.total + 1; }
19   }
20
21   extent void appendSubComposite(query Set<Composite> c, Component p)
22     requires/ensures // analogous to appendSubComposite() in Figure 7.8
23   {
24     foreach (cp isElementOf c.select(x: x.parent == p)) {
25       these.appendComponent(cp.child, cp.parent);
26       these.appendSubComposite(c.select(x: x.child isElementOf
27        c.transitiveClosure().select(y: y.parent == cp.child).child), cp.child);
28     }
29   }
30
31   extent void appendComposite(Extent<Composite> c, Component p)
32     requires/ensures // analogous to appendComposite() in Figure 7.8
33   {
34     these.appendComponent(c.root, p);
35     these.appendSubComposite(c, c.root);
36   }
37
38   extent invariant
39     these.isPartialFunction() & these.transitiveClosure().isIrreflexive() &
40     !(these.root isElementOf these.child) &
41     (!these.isEmpty() => these.root isElementOf these.parent) &
42     these.transitiveClosure().select(cp: cp.parent == these.root).child ==
43       these.child &
44     forAll(p isElementOf these.parent:
45       p.total == these.transitiveClosure().select(cp: cp.parent == p).count());
46 }
```

Figure 7.9: Alternative Rumer Composite pattern specification. As opposed to the specification in Figure 7.7 and Figure 7.8, which represents a composite by a `Composite` element instance, this specification represents a composite by a `Composite` extent instance.

and Figure 7.8, it can fully encapsulate the tree invariant (line 38). As a result, the tree invariant can be specified without ownership. However, representing a composite by an extent instance rather than an element instance has a number of consequences. Unlike element instances, extent instances cannot be retrieved from the program heap using query expressions as there is no "meta extent" for extent instances. Furthermore, it needs to be reconsidered how element type fields, such as the extent field `root`, should be imple-

mented in the compiler. As briefly mentioned in Section 4.4 on page 86, element type variables or fields are implemented as weak references to facilitate the destruction of the referred-to element instances through removal operations. The case at hand, however, suggests that element type fields of extent instances should be implemented as strong references to guarantee that the referred-to element instance exists at least as long as it is referred-to by the extent instance. A detailed consideration of this issue and an evaluation of whether the specification suggested in Figure 7.9 represents a faithful Rumer specification of the Composite pattern remains as future work (see also discussion in Section 7.2).

The Composite pattern specifications shown in Figure 7.7 and Figure 7.8 as well as in Figure 7.9 all use a field to store the number of transitive children of a composite. This choice is in line with the SAVCBS challenge problem and also in line with the prevailing programming practice. However, a more Rumer-way of implementing the number of a composite's transitive children would be to use an interposed element query (see Section 3.2.3) rather than an interposed element field. This choice would avoid the need of incrementing the `total` field of a component's transitive parents in method `append()`, and the SAVCBS challenge problem would not arise in the first place.

A number of proposals have been suggested that address the challenges of verifying the SAVCBS invariant of the Composite pattern in a pure object-oriented setting. Verification techniques based on *ownership* [12, 14, 16] allow the specification and verification of the Composite pattern by leveraging the heap topology enforced by Universe Types. However, an ownership-based specification of the Composite pattern prevents modifications of a composite's components by non-owners. Other proposals typically employ a relaxed visible-state semantics for invariants. Summers and Drossopoulou [19] introduce Considerate Reasoning, a verification technique that 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. Furthermore, techniques have been presented that do not employ a visible-state semantics for invariants. Bierhoff and Aldrich [113] leverage typestate-based permissions to verify a simpler invariant than the SAVCBS challenge invariant for binary Composite tree structures and Jacobs et al. [114] leverage separation logic to verify the SAVCBS invariant for binary Composite tree structures.

7.1.3 Synchronized clocks

In this section, we illustrate the Rumer concepts and verification technique on the synchronized clocks example. The synchronized clocks example has been introduced by Barnett and Naumann [15] as an instance of the Observer pattern. Observer-like patterns give typically rise to multi-object invariants that are challenging for verification.

Figure 7.10 shows an object-oriented implementation of the synchronized clocks example as suggested in [15]. The program implements a very simple scheme to synchronize a clock with its associated server clock. According to this scheme, only a server clock can advance its time (method `tick()` in Figure 7.10(a)), but a client clock can only

```
1  public class ServerClock {
2    private int time;
3
4    public ServerClock() {
5      this.time = 0;
6    }
7
8    public int time() {
9      return this.time;
10   }
11
12   public void tick(int incr) {
13     this.time = this.time + incr;
14   }
15 }
```

```
1  public class Clock {
2    private int time;
3    private ServerClock server;
4
5    public Clock(ServerClock server) {
6      this.time = 0;
7      this.server = server;
8    }
9
10   public int time() {
11     return this.time;
12   }
13
14   public void sync() {
15     this.time = server.time();
16   }
17 }
```

Figure 7.10: Object-oriented implementation (w/o contracts) of synchronized clocks example in a Java-like language, based on [15].

update its time by synchronizing its time with the time of its associated server clock (method `sync()` in Figure 7.10(b)). A client clock thus becomes the "observer" of a server clock's time field, making the server clock become the "subject". This simple clock synchronization scheme establishes the invariant that a client clock cannot overtake its server clock. The verification of this invariant is challenging since it represents a multi-object invariant that is stated at the observer-side.

Figure 7.11 shows the specification of the synchronized clocks example as introduced in previous work [101]. The specification consists of the entity declaration `Clock` and the relationship declaration `SyncClocks`. Relationship `SyncClocks` relates a client clock to a server clock and encapsulates the clock synchronization scheme explained earlier. The relationship declares the two interposed element fields `time` (line 4 and 5), the mold initializer `initialize()` (line 7), the interposed element method `tick()` (line 11), the non-interposed element method `sync()` (line 16), and the extent method `createSyncClocks()` (line 20). Relationship `SyncClocks` interposes a `time` field into each role of the relationship. The mold initializer `initialize()` sets a client clock's time and a server clock's time to zero. The element method `tick()` is interposed into a server clock and increments that server clock's `time` field by the argument value. The non-interposed element method `sync()` sets, for a client clock and its associated server clock, the client clock's `time` field to the value of the server clock's `time` field. Relationship `SyncClocks` establishes an element invariant to ascertain the intended synchronization scheme. The element invariant is declared on line 30 and guarantees that a clock's time is positive and that a client clock's time is less than or equal to its server clock time. Furthermore, relationship `SyncClocks` establishes an extent invariant to guarantee that a client clock can at most synchronize with one server clock and to prevent a clock from being both a client and a server (line 32 and line 33, resp.).

Both invariants declared by relationship `SyncClocks` are admissible according to the

```
1  entity Clock {...}
2
3  relationship SyncClocks participants (Clock client, Clock server) {
4    int >client time;
5    int >server time;
6
7    init initialize()
8      ensures client.time == 0 & server.time == 0;
9    { client.time = 0; server.time = 0; }
10
11   void >server tick(int incr)
12      requires 0 <= incr;
13      ensures  old(time) == time + incr;
14   { time = time + incr; }
15
16   void sync()
17     ensures client.time == server.time;
18   { client.time = server.time; }
19
20 extent SyncClocks createSyncClocks (Clock clCl, Clock svCl)
21   requires
22     clCl != null & svCl != null &
23     !(clCl isElementOf these.client) &
24     !(clCl isElementOf these.server);
25   ensures
26     result.client == clCl & result.server == svCl &
27     these == old(these) union Set(result);
28   { return these.add(new SyncClocks(clCl, svCl).initialize()); }
29
30   invariant 0 <= client.time & client.time <= server.time;
31   extent invariant
32     these.isPartialFunction() &
33     (these.client intersection these.server).isEmpty();
34 }
```

Figure 7.11: Rumer specification of synchronized clocks example.

basic verification technique (see Table 4.4 on page 82). We illustrate our verification technique on the interposed element method `tick()` and the non-interposed element method `sync()`. The verification of the extent method `createSyncClocks()` is analogous to the verification of the extent method `makeFacultyTeachCourse()` of the university example in Figure 7.2. Relationship `SyncClocks` is an instance of an A-benign–B-malign relationship (see Section 5.2 on page 91). As illustrated by Table 5.4 on page 96, the relationship is A-benign–B-malign since it declares an element invariant on element fields that are interposed into both roles of the relationship and since it is B-unique. The part of the element invariant that is responsible for `SyncClocks`'s B-malignity is the reference to a server clock's time. As a server clock's time is shared among several client clocks, the update of the server clock's time through a relationship element instance can compromise the invariants of all other relationship element instances that relate client clocks to the same server clock. The reference to a client clock's time, on the other hand, does not constitute a malign reference. The extent invariant of `SyncClocks` makes the relationship B-unique, guaranteeing that a client clock is related to a server clock by at most one `SyncClocks` element instance.

The framework parameters for the non-interposed element method `sync()` are defined by Table 6.14 on page 167, conjoined with Table 6.15 on page 167 and Table 6.16 on page 168. The method can expect (parameter $\mathbb{X}$) the invariant of its current receiver instance, the element and extent invariants of entity `Clock`, as well as the B-uniqueness assertion of the invariant of the extent instance in which the current receiver instance resides. The invariants vulnerable to the execution of method `sync()` (parameter $\mathbb{V}$) are the invariants of those `SyncClocks` element instances *S* that are related to the same server clock as the current receiver instance (including the current receiver instance), the invariant of the `SyncClocks` extent instance *sx* in which the current receiver instance resides, and the invariant of the owner of the current receiver instance. The proof obligation $\mathbb{B}$ is empty since method `sync()` does not invoke any methods. The proof obligation $\mathbb{E}$ requires method `sync()` to prove that the invariant of its current receiver instance holds and that it preserves the invariants of all the element instances *S* (excluding the current receiver instance) as well as the invariant of the extent instance *sx*. Given the assertion that the invariant of the current receiver instance holds in the initial state of method `sync()`, the method should be able to show that it also holds after the update on line 18. Furthermore, by assuming that the invariants of all the element instances *S* hold in the initial state of method `sync()` and since the update on line 18 does not alter the shared server clock's time, the method should be able to preserve those invariants. Similarly, the method `sync()` should be able to preserve the extent invariant of the extent instance *sx* since that invariant does not rely on the updated interposed field.

The framework parameters for the interposed element method `tick()` are defined by Table 6.17 on page 168. As opposed to the non-interposed element method `sync()`, the interposed element method `tick()` can only expect (parameter $\mathbb{X}$) the element and extent invariants of entity `Clock`. The invariants vulnerable to the execution of method `tick()` (parameter $\mathbb{V}$) are the invariants of those `SyncClocks` element instances *S* that relate the current receiver instance, the invariant of the `SyncClocks` extent instance *sx* in which the element instances *S* reside, and the invariant of the owner of the current receiver instance. The proof obligation $\mathbb{B}$ is vacuous for interposed element methods. The proof obligation $\mathbb{E}$ requires method `tick()` to prove that it preserves the invariants of all the element instances *S* as well as the invariant of the extent instance *sx*. By assuming that the invariants of all the element instances *S* hold in the initial state of method `tick()` and since the update on line 13 does at most advance the shared server clock's time, the method should be able to preserve those invariants. Similarly, the method `tick()` should be able to preserve the extent invariant of the extent instance *sx* since that invariant does not rely on the updated interposed field.

The synchronized clocks example demonstrates that the proof obligation $\mathbb{E}$ established for non-interposed and interposed element methods of unilaterally or bilaterally malign relationships prevent such methods from breaking any element invariants that hold in the initial state of the method. Whether a non-interposed or interposed element method of such a relationship is able to preserve an element invariant depends on the particular formulation of the invariant. For example, if the invariant on line 30 required a client

clock's time to be at most ten ticks behind its server clock's time, then method `tick()` could not be verified. To produce a verifiable program, a programmer would need to encapsulate the update of a server clock's time in an extent method. Since an extent method can invoke methods on any instances of the current receiver instance's element type, it could invoke method `sync()` on all `SyncClocks` element instances that relate the updated server clock and whose client clock's time would fall below the threshold.

The same reasoning also applies to any element method of an entity or relationship (free, benign, or malign) with regard to the preservation of extent invariants. If an element method is not able to preserve the extent invariant of the extent instance in which the current receiver instance resides, the element method's body must be encapsulated in a corresponding extent method. For example, the update of a parent component's `total` field in the Composite pattern program shown in Figure 7.7 and Figure 7.9 is only verifiable if the update occurs in an extent method. Only an extent method can "atomically" write to the `total` fields of all transitive parent components of the newly appended child component and can thus re-establish the broken extent invariant.

The verification of Observer-like patterns in a pure object-oriented setting is challenging. Due to the nature of the pattern, the multi-object invariant is established at the observer-side. Since this side is the many side of the dependency, verification techniques based on *ownership* [12, 14, 16] are not applicable. The only invariant-based techniques that are able to verify Observer-like patterns are the work by Barnett and Naumann [15] on friendship-based invariants and the work by Leino and Schulte [17] on history invariants. The two techniques differ in their applicability and entailed restrictions. Whereas the friendship-based approach requires a granting class (subject) to "know" its friend (observer) classes, the history invariant-based approach drops this restriction. This flexibility comes at the cost of limiting the possible evolution of the subject. 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, on the other hand, the subject is limited to evolve monotonically.

7.2 Future work

As demonstrated in the previous section, the current Rumer programming and specification language as well as its verification technique are capable of accommodating a substantial range of programs and verification challenges. Nonetheless, there remain open challenges, pointing us to exciting opportunities for further investigation. We first present opportunities for language development and then opportunities for program verification.

7.2.1 Language development

Element instance destruction. Due to the non-strict execution semantics of the Rumer built-in removal operator (see Section 3.4.2 on page 71), the actual point in time when a

to-be destructed element instance is actually destructed can generally not be determined modularly. Only in combination with ownership, is a modular, static determination of the time of destruction possible. In future work, we would like to investigate whether this form of non-modularity is undesirable or even limiting. Our investigations may result in extensions to the Rumer specification language to allow the expression of scheduling constraints for element instance destruction. In addition, further thought has to be given to the discrimination between references and relationship participation. System Invariant 3.7 on page 74 only prevents the destruction of an element instance as long as the element instance (possibly transitively) participates in a relationship element instance. The existence of references to an element instance, however, is ignored by System Invariant 3.7. As briefly mentioned during the discussion of the alternative Composite pattern implementation (see Figure 7.9), this treatment may have to be changed for reference fields of extent instances.

Query optimizations. The call-by-name evaluation of predicate references (see Section 3.2.3 on page 49) adds some run-time overhead to the execution of a Rumer program. We foresee the implementation of various strategies to prevent reevaluation of a query expression if the underlying extent instances have not changed since the last evaluation of the query. Both dynamic and static strategies are conceivable. For example, a dynamic strategy could monitor writes to extent instances and invalidate the cached result sets of those query expressions that rely on the changed extent instances. Leveraging the strong encapsulation properties of the Rumer language as well as routine contracts, this idea could also be partially implemented at compile-time. Orthogonally to optimizations targeting the unnecessary reevaluation of query expressions, performance gains for query evaluation can be obtained through parallelization. Since query expressions in Rumer are side-effect free, they can safely be parallelized.

Pure methods. An obvious extension to the current Rumer programming language would be the support of pure methods. Pure methods would complement the currently available built-in query operators and would allow programmers to provide their own customized queries. As prior work on the support of pure methods in object-oriented specifications [115, 116, 117, 118] has shown, there are a number of subtleties to consider. Since pure methods can declare preconditions and postconditions like ordinary methods, they may lead to recursive specifications. Furthermore, pure methods are typically axiomatized in the underlying deductive system used for reasoning about specifications. This axiomatization can lead to the accidental introduction of inconsistencies through user-defined preconditions and postconditions. We expect the solutions devised to address these issues in the context of object-oriented specifications to apply likewise to Rumer specifications. In addition, we expect the concept of an extent instance to prove viable in the formulation of well-founded, recursive pure methods.

Sharing of interposed members. Member interposition gives rise to implicit role declarations. Whilst member interposition is lightweight and adequate for many scenarios, it prohibits the sharing of interposed members across several relationship types. In future work, we would like to address this limitation and investigate the addition of explicit roles in the spirit of [61, 64] to the Rumer language. A preliminary investigation has already been carried out by Sebestyén-Pál [80] in her master's thesis. Those investigations suggest that explicit roles are key to the adaptation and integration of entities and relationships to allow consolidating separately devised Rumer programs.

Program scalability. The scalability of the current Rumer language is limited. The most limiting factor is the application unit of a Rumer program. The current constraint that there exists only one application unit per Rumer program results in the culmination of applied code in the application. To allow Rumer programs to scale in a modular way, we would like to introduce a module system to the Rumer language. Such a module system will allow the encapsulation of a subsystem ("mini-world") of entity and relationship declarations together with their applied code in an instantiable unit. A preliminary investigation has already been carried out by Conconi [81] in his master's thesis. The elaborated Rumer module system leverages explicit roles (based on [80]) in combination with type genericity and virtual members and supports visibility modifiers to facilitate information hiding. Case studies (including the implementation of a raytracer) have been executed to assess the new module system. The results indicate that the elaborated module system allows for modular scalability of Rumer programs and at the same time preserves the guarantees made by the Matryoshka Principle.

7.2.2 Program verification

Verification scalability. In future work, we would like to extend our verification technique to support a module system similar to the one elaborated as part of Conconi's thesis [81]. Our preliminary investigation suggests that a module system can provide program scalability and at the same time preserve those guarantees made by the Matryoshka Principle that are crucial to program verification. Such an effort promises to allow the modular specification and verification of role-based invariants, which constrain the dependencies between various entities and relationships, and thus may constitute an important step towards scaling program verification to real-world software. In addition, we expect a module system to help making those two program well-formedness checks that are currently not modular, modular. These checks guarantee that a program's participants relation is acyclic (see System Invariant 4.1 on page 77) and that entity and relationship fields do not point to instances of a base type of which the declaring base type is a direct or transitive participant (see Section 6.2.3 on page 146).

(Semi-) automatic verification. The implementation of a static verifier that allows the compile-time verification of Rumer programs based on our verification technique is the

logical next step to take as part of future work. To implement a static verifier for Rumer, we have to *(i)* choose a prover that we can use as a back-end to our compiler and we have to *(ii)* axiomatize the semantics of the Rumer language and verification technique in terms of the prover's deductive system. For the choice of the prover, we favor a whole verification toolchain, such as the Boogie intermediate verification language [112] and verification condition generator. Such a toolchain would take over the task of verification condition generation and shield us from direct interactions with the underlying prover (i.e., the Z3 SMT solver [119] in the case of Boogie). However, given the limited support of set theory by current SMT solvers, we may either have to extend an SMT solver's mathematical theory with set theory or to consider a theorem prover as a direct back-end. For the axiomatization of the Rumer language and verification technique, we have to give further thought to the expression of frame properties [120]. We hope to leverage a Rumer program's participants clauses as they implicitly indicate the locations that may potentially change during the execution of a routine. If such an implicit declaration of frame properties is unfeasible, we consider extending the Rumer specification language with explicit modifies clauses in the spirit of [12].

8 Conclusion

This thesis presents a modular, invariant-based verification technique for our relationship-based programming and specification language Rumer. The verification technique relies on a stratified programming model. This model results from the abstractions underlying the Rumer language. Rumer provides two instantiable language abstractions — the entity and the relationship — to represent objects and the various relationships between them. A relationship declaration indicates the type of instances it relates and accepts both entities and relationships as participants. As a result, relationship declarations naturally stratify the program, giving structure to the heap and preventing cyclic chains of references.

The stratified programming model allows the implementation of a modularization discipline that provides strong encapsulation for Rumer types. The Matryoshka Principle captures this discipline in terms of admissibility criteria that stipulate restrictions on writes to locations, the expression of invariants, and method invocations. These admissibility criteria guarantee that locations are encapsulated by their declaring type, that invariants can only rely on locations that are encapsulated by the invariant-declaring type, and that method invocations cannot lead to transitive call-backs.

Our verification technique leverages the new programming model in multiple ways. It is based on the Matryoshka Principle, which provides modular reasoning about state changes and prevents transitive call-backs. Relying on these guarantees, our verification technique adopts a visible-state semantics for invariants. To facilitate the verification of multi-object invariants, our technique relies on two language mechanisms that can be used separately or combined. The first mechanism is member interposition. It allows the declaration of the properties of a relationship participant that are specific to a relationship and encapsulates those properties in the respective relationship. The second mechanism is selective ownership. It represents a hybrid ownership scheme in which owned and shared instances coexist and facilitates the declaration of ownership-based invariants.

We evaluated our verification technique and the suggested programming model based on a number of well-known program verification problems. The results provide evidence that the suggested verification technique and programming model is capable of addressing those challenge problems. Crucial to the success seems to be the coordinated interplay of the various unique features of the Rumer programming language. For example, the notion

of an extent instance to capture a programmer-defined “universe” of instances proved to be beneficial for expressing and verifying sophisticated invariants. We also performed a preliminary investigation of future extensions to the Rumer language. These extensions allow Rumer programs to scale to real-world-sized programs. The investigation indicates that the properties of the current Rumer system that are crucial to modular verification are scalable too. These insights raise hope that the presented verification technique and programming model can serve as a stepping stone towards scaling program verification to real-world software.

Appendix

In this appendix, we provide the complete grammar of the Rumer programming and specification language.

A.1 Rumer grammar

Below, we list the Rumer grammar in EBNF (Extended Backus-Naur Form) form: curly braces { *term* } denote zero, one, or more repetitions of *term* and squared brackets [*term*] denote zero or one repetition of *term*. The grammar conforms to the ANTLR [73] style guidelines and uses lower case initial letters for non-terminals (i.e., parser rules) and upper case initial letters for terminals (i.e., lexer rules).

program	≙	{ typeDecl } applicationDecl { typeDecl }
typeDecl	≙	entityDecl |
		relationshipDecl
entityDecl	≙	"entity" Identifier "{" { memberDecl } "}"
relationshipDecl	≙	"relationship" Identifier participantsDecl "{" { memberDecl } "}"
participantsDecl	≙	"participants" "(" type Identifier "," type Identifier ")"
memberDecl	≙	fieldDecl |
		queryDecl |
		routineDecl |
		invariantDecl
fieldDecl	≙	elementFieldDecl |
		extentFieldDecl
elementFieldDecl	≙	type [interpositionDecl] Identifier { "," Identifier } ";"
interpositionDecl	≙	">" (Identifier | "domain" | "range")
extentFieldDecl	≙	"extent" type Identifier { "," Identifier } ";"
queryDecl	≙	elementQueryDecl |
		extentQueryDecl
elementQueryDecl	≙	"let" [interpositionDecl] Identifier "be" predicate ";" |
		"let" "query" type [interpositionDecl] Identifier "be" predicate ";"
extentQueryDecl	≙	"let" "extent" Identifier "be" predicate ";" |
		"let" "extent" "query" type Identifier "be" predicate ";"
routineDecl	≙	moldInitializerDecl |
		extentConstructorDecl |
		methodDecl
moldInitializerDecl	≙	"init" routineSignatureTail block
extentConstructorDecl	≙	"extent" routineSignatureTail block
methodDecl	≙	elementMethodDecl |

		extentMethodDecl
elementMethodDecl	≙	“void” [interpositionDecl] routineSignatureTail block |
		type [interpositionDecl] routineSignatureTail block
extentMethodDecl	≙	“extent” “void” routineSignatureTail block |
		“extent” type routineSignatureTail block
routineSignatureTail	≙	Identifier “(” [formalParameters] “)” { precondition } { postcondition }
formalParameters	≙	parameterDecl { “,” parameterDecl }
parameterDecl	≙	type Identifier |
		“query” type Identifier
precondition	≙	“requires” predicate “;”
postcondition	≙	“ensures” predicate “;”
invariantDecl	≙	“invariant” predicate “;” |
		“extent” “invariant” predicate “;”
applicationDecl	≙	“application” Identifier “{” { partDecl } mainActionDecl { partDecl } “}”
partDecl	≙	appVarDecl |
		appQueryDecl |
		actionDecl |
		appInvariantDecl
appVarDecl	≙	type Identifier { “,” Identifier } “;”
appQueryDecl	≙	“let” Identifier “be” predicate “;” |
		“let” “query” type Identifier “be” predicate “;”
actionDecl	≙	“void” routineSignatureTail block |
		type routineSignatureTail block
appInvariantDecl	≙	“invariant” predicate “;”
mainActionDecl	≙	“main” “(” “)” block
localVarDecl	≙	type Identifier [“=” predicate] { “,” Identifier [“=” predicate] } “;”
localQueryDecl	≙	“let” Identifier “be” predicate “;” |
		“let” “query” type Identifier “be” predicate “;”
boundedVarDecl	≙	identifierOrPair “isElementOf” expression |
		Identifier “isItemOf” expression
identifierOrPair	≙	Identifier |
		“pair” “(” identifierOrPair “,” identifierOrPair “)”
type	≙	referenceType |
		valueType
referenceType	≙	elementType |
		extentType
elementType	≙	Identifier |
		Identifier “.” (Identifier | “domain” | “range”) |
		Any
extentType	≙	“Extent” “〈” Identifier “〉”
valueType	≙	setType |
		pairType |
		bagType |
		itemType
setType	≙	“Set” “〈” coordinateType “〉”
pairType	≙	“Pair” “〈” coordinateType “,” coordinateType “〉”
coordinateType	≙	referenceType |
		setType |
		pairType
bagType	≙	“Bag” “〈” itemType “〉” |
		“Bag” “〈” bagType “〉”
itemType	≙	moldType |

```
                        primitiveType
moldType            ≙   "Mold" "〈" Identifier "〉"
primitiveType       ≙   "boolean" |
                        "string" |
                        "int" |
                        "float"
block               ≙   "{" { blockStatement } "}"
blockStatement      ≙   localVarDecl |
                        localQueryDecl |
                        statement
statement           ≙   "assert" predicate ";" |
                        "if" "(" predicate ")" block [ "else" block ] |
                        "while" "(" predicate ")" block |
                        "foreach" "(" boundedVarDecl ")" block |
                        "return" predicate ";" |
                        expression "=" predicate ";" |
                        expression ";" |
                        "these" "." "remove" arguments ";" |
                        "writeBoolean" "(" expression ")" ";" |
                        "writeString" "(" expression ")" ";" |
                        "writeInt" "(" expression ")" ";" |
                        "writeFloat" "(" expression ")" ";"
arguments           ≙   "(" [ predicate {"," predicate } ] ")"
predicate           ≙   andPredicate [ ( "=>" | "<=>" ) andPredicate ]
andPredicate        ≙   notPredicate { ( "&" | "|" ) notPredicate }
notPredicate        ≙   "!" notPredicate |
                        atomicPredicate
atomicPredicate     ≙   quantifiedPredicate |
                        expression [ relationalOp expression | "instanceOf" type ]
quantifiedPredicate ≙   "forAll" "(" boundedVarDecl ":" predicate ")" |
                        "thereExists" "(" boundedVarDecl ":" predicate ")"
relationalOp        ≙   "==" |
                        "!=" |
                        "<=" |
                        ">=" |
                        "<" |
                        ">" |
                        "isSubsetOf" |
                        "isSubbagOf" |
                        "isElementOf" |
                        "isItemOf"
expression          ≙   arithmeticExpr { "cartesianProduct" arithmeticExpr } |
                        arithmeticExpr { "composition" arithmeticExpr } |
                        arithmeticExpr { "union" arithmeticExpr } |
                        arithmeticExpr { "intersection" arithmeticExpr } [ "difference" arithmeticExpr ]
arithmeticExpr      ≙   term { ( "+" | "-" ) term }
term                ≙   factor { ( "*" | "/" | "%" ) factor }
factor              ≙   [ "+" | "-" ] atomicExpr
atomicExpr          ≙   "readBoolean" "(" ")" |
                        "readString" "(" ")" |
                        "readInt" "(" ")" |
                        "readFloat" "(" ")" |
```

```
                    "Set" "(" expression { "," expression } ")" |
                    "Pair" "(" expression "," expression ")" |
                    postfixExpr |
                    literal
postfixExpr         ≙  primaryExpr { "." postfixExprTail }
primaryExpr         ≙  "this" |
                    "these" |
                    "domain" |
                    "range" |
                    "extent" |
                    "result" |
                    "old" "(" predicate ")" |
                    Identifier |
                    Identifier arguments |
                    "these" "." "add" arguments |
                    "new" Identifier "(" ")" |
                    "new" "(" expression "," expression ")" |
                    "new" extentType arguments |
                    "(" predicate ")"
postfixExprTail     ≙  Identifier |
                    "extent" |
                    "domain" |
                    "range" |
                    Identifier arguments |
                    "cast" "(" type ")" |
                    "average" "(" [ Identifier ":" expression ] ")" |
                    "maximum" "(" [ Identifier ":" expression ] ")" |
                    "minimum" "(" [ Identifier ":" expression ] ")" |
                    "sum" "(" [ Identifier ":" expression ] ")" |
                    "map" "(" Identifier ":" expression ")" |
                    "select" "(" Identifier ":" predicate ")" |
                    "aggregate" "(" Identifier "," Identifier "=" predicate ":" predicate ")" |
                    "inverse" "(" ")" |
                    "transitiveClosure" "(" ")" |
                    "isEmpty" "(" ")" |
                    "isSymmetric" "(" ")" |
                    "isAsymmetric" "(" ")" |
                    "isAntisymmetric" "(" ")" |
                    "isReflexive" "(" ")" |
                    "isIrreflexive" "(" ")" |
                    "isTransitive" "(" ")" |
                    "isIdentity" "(" ")" |
                    "isTotalRelation" "(" ")" |
                    "isSurjectiveRelation" "(" ")" |
                    "isPartialFunction" "(" ")" |
                    "isTotalFunction" "(" ")" |
                    "isPartialInjection" "(" ")" |
                    "isTotalInjection" "(" ")" |
                    "isPartialSurjection" "(" ")" |
                    "isTotalSurjection" "(" ")" |
                    "isPartialBijection" "(" ")" |
                    "isTotalBijection" "(" ")" |
```

		“isRightTotal” “(” “)” |
		“isLeftTotal” “(” “)” |
		“isRightUnique” “(” “)” |
		“isLeftUnique” “(” “)” |
		“count” “(” “)”
literal	≙	BooleanLiteral |
		StringLiteral |
		DecimalLiteral |
		FloatingPointLiteral |
		NullLiteral

Bibliography

[1] James 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.

[2] Antonio Albano, Giorgio Ghelli, and Renzo 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.

[3] Gavin M. Bierman and Alisdair 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.

[4] Yu David Liu and Scott F. Smith. Interaction-based programming with Classages. In *20th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'05)*, pages 191–209. ACM, 2005.

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

[6] Stephen Nelson, David J. Pearce, and James Noble. First class relationships for OO languages. In *6th International Workshop on Multiparadigm Programming with Object-Oriented Languages (MPOOL'08)*, 2008.

[7] John McCarthy. A basis for a mathematical theory of computation. In *Western Joint IRE-AIEE-ACM Computer Conference*, pages 225–238. ACM, 1961.

[8] Robert W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*, volume 19, pages 19–32. 19th Symposium in Applied Mathematics, 1967.

[9] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[10] Arnd Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. Habilitation thesis, Technical University of Munich, January 1997.

[11] Kees Huizing and Ruurd Kuiper. Verification of object oriented programs using class invariants. In *3rd International Conference on Fundamental Approaches to Software Engineering (FASE'00)*, volume 1783 of *Lecture Notes in Computer Science*, pages 208–221. Springer, 2000.

[12] Peter Müller. *Modular Specification and Verification of Object-Oriented Programs*, volume 2262 of *Lecture Notes in Computer Science*. Springer, 2002.

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

[14] K. Rustan M. Leino and Peter 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.

[15] Michael Barnett and David 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.

[16] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62(3):253–286, 2006.

[17] K. Rustan M. Leino and Wolfram 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.

[18] Ronald Middelkoop, Cornelis Huizing, Ruurd Kuiper, and Erik J. Luit. Invariants for non-hierarchical object structures. *Electronic Notes in Theoretical Computer Science*, 195:211–229, 2008.

[19] Alexander J. Summers and Sophia Drossopoulou. Considerate reasoning and the Composite design patterns. In *11th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI 2010)*, volume 5944 of *Lecture Notes in Computer Science*, pages 328–344. Springer, 2010.

[20] C.A.R. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[21] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall Professional Technical Reference, 1997.

[22] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *15th International Workshop on Computer Science Logic (CSL'01).*, volume 2142 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2001.

[23] Matthew J. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, 2005.

[24] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 247–258. ACM, 2005.

[25] Matthew J. Parkinson. Class invariants: The end of the road? In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO'07)*, 2007.

[26] Matthew J. Parkinson and Gavin M. Bierman. Separation logic, abstraction and inheritance. In *35th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08)*, pages 75–86. ACM, 2008.

[27] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *14th International Symposium on Formal Methods (FM'06)*, Lecture Notes in Computer Science, page 4085. Springer, 2006.

[28] Bernd Schoeller. *Making Classes Provable through Contracts, Models and Frames*. PhD thesis, ETH Zurich, 2007.

[29] Anindya Banerjee, David A. Naumann, and Stan Rosenberg. Regional logic for local reasoning about global invariants. In *22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *Lecture Notes in Computer Science*, pages 387–411. Springer, 2008.

[30] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *23rd European Conference on Object-Oriented Programming (ECOOP'09)*, volume 5653 of *Lecture Notes in Computer Science*, pages 148–172. Springer, 2009.

[31] Ivar Jacobson, Grady Booch, and James E. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley, 1999.

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

[33] Stephanie Balzer, Alexandra Burns, and Thomas R. Gross. Objects in context: An empirical study of object relationships. Technical Report 594, ETH Zurich, May 2008.

[34] Alexandra Burns. The relationship detector: Uncovering hidden relationships in object-oriented programs. Master's thesis, ETH Zurich, 2006. Supervised by Stephanie Balzer and Thomas R. Gross.

[35] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems, 2nd Edition.* Benjamin/Cummings, second edition, 1994.

[36] James Noble. Basic relationship patterns. In *2nd European Conference on Pattern Languages of Programs (EuroPLoP'97)*, 1997.

[37] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[38] Standard Performance Evaluation Corporation. The SPEC JVM98 benchmarks. http://www.spec.org/jvm98/, 1998.

[39] Standard Performance Evaluation Corporation. The SPEC JBB2005 benchmark. http://www.spec.org/jbb2005/, 2005.

[40] Daniel Jackson and Allison Waingold. Lightweight extraction of object models from bytecode. *IEEE Transactions on Software Engineering (TSE)*, 27(2):156–169, 2001.

[41] Ana Milanova. Precise identification of composition relationships for UML class diagrams. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 76–85. ACM, 2005.

[42] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. http://asm.objectweb.org/, November 2002.

[43] Sarah Matzko, Peter J. Clarke, Tanton H. Gibbs, Brian A. Malloy, James F. Power, and Rosemary Monahan. Reveal: A tool to reverse engineer class diagrams. In *40th International Conference on Tools Pacific (TOOLS Pacific'02)*, pages 13–21. Australian Computer Society, Inc., 2002.

[44] Yann-Gaël Guéhéneuc and Hervé Albin-Amiot. Recovering binary class relationships: Putting icing on the UML cake. In *19nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 301–314. ACM, 2004.

[45] Nick Mitchell. The runtime structure of object ownership. In *20th European Conference on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *Lecture Notes in Computer Science*, pages 74–98. Springer, 2006.

[46] Cormac Flanagan and Stephen N. Freund. Dynamic architecture extraction. In *1st Combined International Workshops on Formal Approaches to Software Testing and Runtime Verification (FATES/RV'06)*, Lecture Notes in Computer Science, pages 209–224, 2006.

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

[48] David J. Pearce and James Noble. Relationship aspect patterns. In *11th European Conference on Pattern Languages of Programs (EuroPLoP'06)*, 2006.

[49] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242. Springer, 1997.

[50] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *15th European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–353. Springer, 2001.

[51] AspectJ Team. The AspectJ™ programming guide. http://www.eclipse.org/aspectj.

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

[53] Eric Bodden, Reehan Shaikh, and Laurie Hendren. Relational aspects as tracematches. In *7th International Conference on Aspect-Oriented Software Development (AOSD '08)*, pages 84–95. ACM, 2008.

[54] Chris Allan, Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Ondřej Lhoták, Oege de Moor, Damien Sereni, Ganesh Sittampalam, and Julian Tibble. Adding trace matching with free variables to AspectJ. In *20th Annual ACM*

SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'05), pages 345–364. ACM, 2005.

[55] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative object identity using relation types. In *21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 54–78. Springer, 2007.

[56] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 23(3):396–450, 2001.

[57] Stephen Nelson, David J. Pearce, and James Noble. Implementing relationships using Affinity. In *2nd Workshop on Relationships and Associations in Object-Oriented Languages (RAOOL'09)*, pages 5–8. ACM, 2009.

[58] Henry G. Baker. Equal rights for functional objects or, the more things change, the more they are the same. *ACM SIGPLAN OOPS Messenger*, 4(4):2–27, 1993.

[59] James Noble and John Grundy. Explicit relationships in object-oriented development. In *Conference on the Technology of Object-Oriented Languages and Systems (TOOLS'95)*, pages 211–226. Prentice-Hall, 1995.

[60] Ciera Jaspan and Jonathan Aldrich. Checking framework interactions with relationships. In *23rd European Conference on Object-Oriented Programming (ECOOP'09)*, volume 5653 of *Lecture Notes in Computer Science*, pages 27–51. Springer, 2009.

[61] Trygve Reenskaug, Per Wold, and Odd Arild Lehne. *Working with Objects: The OOram Software Engineering Method*. Manning/Prentice Hall, 1996.

[62] Dirk Riehle and Thomas R. Gross. Role model based framework design and integration. In *13th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 117–133, 1998.

[63] Dirk Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, ETH Zurich, 2000. 13509.

[64] Friedrich Steimann. On the representation of roles in object-oriented and conceptual modelling. *Data & Knowledge Engineering*, 35(1):83–106, October 2000.

[65] Stephan Herrmann. ObjectTeams: Improving modularity for crosscutting collaborations. In *NetObjectDays*, volume 2591 of *Lecture Notes in Computer Science*, pages 248–264. Springer, 2002.

[66] Stephan Herrmann, Christine Hundt, and Marco Mosconi. ObjectTeams/Java language definition - version 1.0. Technical Report 2007/03, Technical University Berlin, 2007.

[67] Matteo Baldoni, Guido Boella, and Leendert W. N. van der Torre. Interaction between objects in powerJava. *Journal of Object Technology (JOT)*, 6(2):5–30, 2007.

[68] Michael Pradel and Martin Odersky. Scala Roles - a lightweight approach towards reusable collaborations. In *3rd International Conference on Software and Data Technologies (ICSOFT'08)*, pages 13–20, 2008.

[69] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala, A comprehensive step-by-step guide*. Artima Developer, 2008.

[70] E. F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[71] Stephanie Balzer, Thomas R. Gross, and Patrick 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.

[72] Barbara Liskov and Stephen Zilles. Programming with abstract data types. In *ACM SIGPLAN Notices*, volume 9, pages 50–59. ACM, 1974.

[73] Terence Parr. *The Definitive ANTLR Reference: Building Domain-Specific Languages*. The Pragmatic Bookshelf, 2007.

[74] Roderick G.G. Cattell and Douglas K. Barry. *The Object Data Standard: ODMG 3.0*. Morgan Kaufmann, 2000.

[75] Stephanie Balzer and Thomas R. Gross. Verifying multi-object invariants with relationships. In *25th European Conference on Object-Oriented Programming (ECOOP'11)*, volume 6813 of *Lecture Notes in Computer Science*, pages 358–382. Springer, 2011.

[76] Gavin M. Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in Cω. In *19th European Conference on Object-Oriented Programming (ECOOP'05)*, Lecture Notes in Computer Science, pages 287–311. Springer, 2005.

[77] Darren Willis, David J. Pearce, and James Noble. Efficient object querying for Java. In *20th European Conference on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *Lecture Notes in Computer Science*, pages 28–49. Springer, 2006.

[78] Gavin M. Bierman, Erik Meijer, and Mads 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.

[79] Don Box and Anders Hejlsberg. LINQ: .NET Language-Integrated Query. http://msdn2.microsoft.com/en-us/library/bb308959.aspx, February 2007.

[80] Ágnes Sebestyén-Pál. A comprehensive refinement model for Rumer. Master's thesis, ETH Zurich, June 2009. Supervised by Stephanie Balzer and Thomas R. Gross.

[81] Reto Conconi. An extensible and contract-based module system for Rumer. Master's thesis, ETH Zurich, 2011. Supervised by Stephanie Balzer and Thomas R. Gross.

[82] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall Professional Technical Reference, 1991.

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

[84] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *International Journal on Software Tools for Technology Transfer (STTT'05)*, 7(3):212–232, 2005.

[85] Mike Barnett, K. Rustan M. Leino, and Wolfram 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.

[86] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In *International Symposium of Formal Methods Europe (FM'05)*, volume 3582 of *Lecture Notes in Computer Science*, pages 26–42. Springer, 2005.

[87] G. T. Leavens, K. R. M. Leino, and P. Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Aspects of Computing*, 19(2):159–189, 2007.

[88] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 337–350. ACM, 2007.

[89] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[90] John C. Reynolds. Preliminary design of the programming language Forsythe. Technical Report CMU-CS-88-159, Carnegie Mellon University, June 1988.

[91] Jay M. Spitzen and Ben Wegbreit. The verification and synthesis of data structures. *Acta Informatica*, 4(2):127–144, 1975.

[92] John V. Guttag. Notes on type abstraction (version 2). *IEEE Transactions on Software Engineering*, 6(1):13–23, 1980.

[93] Dave Clarke, John Potter, and James 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.

[94] Dave Clarke. *Object Ownership and Containment*. PhD thesis, University of New South Wales, October 2002.

[95] Dave Clarke and Sophia 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.

[96] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology (JOT)*, 4(8):5–32, 2005.

[97] Alex Potanin, James Noble, Dave Clarke, and Robert Biddle. Generic ownership for generic Java. In *21th ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'06)*, pages 311–324. ACM, 2006.

[98] Dave Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. Universe Types for topology and encapsulation. In *6th International Symposium of Formal Methods for Components and Objects (FMCO'07)*, volume 5382 of *Lecture Notes in Computer Science*, pages 72–112. Springer, 2007.

[99] Werner Dietl, Sophia Drossopoulou, and Peter 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.

[100] Werner Dietl. *Universe Types Topology, Encapsulation, Genericity, and Tools*. PhD thesis, ETH Zurich, 2009. 18522.

[101] Stephanie Balzer and Thomas R. Gross. Modular reasoning about invariants over shared state with interposed data members. In *4th ACM SIGPLAN Workshop on Programming Languages meets Program Verification (PLPV'10)*, pages 49–56. ACM, 2010.

[102] Dave Clarke and Tobias Wrigstad. External uniqueness is unique enough. In *17th European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 176–200. Springer, 2003.

[103] Peter Müller and Arsenii Rudich. Ownership transfer in Universe Types. In *22nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'07)*, pages 461–478. ACM, 2007.

[104] Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander Summers. A unified framework for verification techniques for object invariants. In *22nd European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *Lecture Notes in Computer Science*, pages 412–437. Springer, 2008.

[105] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. Universe-type-based verification techniques for mutable static fields and methods. *Journal of Object Technology (JOT)*, 8(4):85–125, 2009.

[106] John Hogg, Doug Lea, Alan Wills, Dennis deChampeaux, and Richard Holt. The geneva convention on the treatment of object aliasing. *ACM SIGPLAN OOPS Messenger*, 3(2):11–16, 1992.

[107] James Noble, Jan Vitek, and John 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.

[108] Chandrasekhar Boyapati, Robert Lee, and Martin C. 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.

[109] Chandrasekhar Boyapati, Alexandru Salcianu, William S. Beebee, and Martin C. Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '03)*, pages 324–337. ACM, 2003.

[110] Yi Lu, John Potter, and Jingling Xue. Validity invariants and effects. In *21st European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 202–226. Springer, 2007.

[111] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for flexible object invariants. In *International Workshop on Aliasing, Confinement and Ownership in object-oriented programming (IWACO'09)*, pages 1–9. ACM, 2009.

[112] K. Rustan M. Leino. This is boogie 2. Technical report, Microsoft Research, 2009. Draft.

[113] Kevin Bierhoff and Jonathan Aldrich. Permissions to specify the Composite design pattern. In *7th International Workshop on Specification and Verification of Component-Based Systems (SAVCBS'08)*, pages 89–94, 2008.

[114] Bart Jacobs, Jan Smans, and Frank Piessens. Verifying the Composite pattern using separation logic. In *7th International Workshop on Specification and Verification of Component-Based Systems (SAVCBS'08)*, pages 83–88, 2008.

[115] Ádám Darvas and Peter Müller. Reasoning about method calls in interface specifications. *Journal of Object Technology (JOT)*, 5(5):59–85, 2006.

[116] Ádám Darvas and K. Rustan M. Leino. Practical reasoning about invocations and implementations of pure methods. In *10th International Conference on Fundamental Approaches to Software Engineering (FASE'07)*, volume 4422 of *Lecture Notes in Computer Science*, pages 336–351. Springer, 2007.

[117] Arsenii Rudich, Ádám Darvas, and Peter Müller. Checking well-formedness of pure-method specifications. In *15th International Symposium on Formal Methods (FM'06)*, volume 5014 of *Lecture Notes in Computer Science*, pages 68–83. Springer, 2008.

[118] K. Rustan M. Leino and Ronald Middelkoop. Proving consistency of pure methods and model fields. In *12th International Conference on Fundamental Approaches to Software Engineering (FASE'09)*, volume 5503 of *Lecture Notes in Computer Science*, pages 231–245. Springer, 2009.

[119] Z3. http://research.microsoft.com/en-us/um/redmond/projects/z3/.

[120] Alexander Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Transactions on Software Engineering (TSE)*, 21(10):785–798, 1995.