A Proof System for Invariants in Layered OO Designs

Ronald Middelkoop Cornelis Huizing Ruurd Kuiper Erik J. Luit
Department of Mathematics and Computer Science,
Technische Universiteit Eindhoven
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
{r.middelkoop, c.huizing, r.kuiper, e.j.luit}@tue.nl

Abstract

Although invariants have a long history, their meaning in OO designs is still under discussion. OO designs often include functionality that is used by different otherwise unrelated objects (shared functionality). We identify a problem with current interpretations of invariants in such designs. OO designs are often layered, where a layer uses functionality of a lower layer (in particular, shared functionality) but has little or no involvement with higher layers. As a result, higher layers can rely on lower layer invariants and lower layers do not rely on higher layer invariants. This is not reflected by current interpretations of invariants. We propose to make layers explicit in specifications and introduce a new interpretation of invariants that exploits these layers. Furthermore, we present a sound, modular verification technique that ensures the new interpretation is satisfied.

1 Introduction

Object-oriented (OO) designs often include methods that provide shared functionality (i.e., that are used by several otherwise unrelated objects). Examples include a canvas in graphics applications, and static method sqrt of class Math from the Java API. Current specification and verification techniques either disallow calls to such methods, or do not guarantee that invariants of objects that are relevant to the proper functioning of the method hold when the method is executed. In this paper, we introduce an approach that resolves these issues.

A (functional) specification captures the desired relation between the prestate and the poststate of a method execution. Sometimes, an informal specification is sufficient: either the ease of specification outweighs any ambiguity in that specification, or there is an implicit understanding of where an implementation is allowed to deviate from the specification. A formal specification, however, can serve as a contract between the specifier and the implementor (which might be two roles of the same person) [19]. Ideally, the correctness of the implementation with respect to the specification can be formally and automatically verified. Together with pre- and postconditions, *class invariants* (or invariants for short) have a central role in most formal OO specification techniques. Conceptually, an invariant is a property that always holds (for instance, see [28]): an invariant of a class C captures a consistency property of objects of C. The use of invariants can significantly reduce the specification overhead [19]. Invariants can also be used for abstraction [12]. They allow the specifier to keep parts of desired input/output relations implicit, thus hiding information [24] and increasing the modularity of the design [21].[1]

A formal specification requires an unambiguous interpretation of the specification constructs in that specification. Although invariants have a long history, the interpretation[2] of invariants in OO designs is still under discussion. A key issue is that commonly (i.e., in many OO designs), an invariant of an object is temporarily violated while the state it depends on is being updated. This is unproblematic as the

[1] Our approach accounts for information hiding. To simplify the technical treatment, it is omitted from the language. Techniques to add it are described in, e.g., [11, 16, 23]

[2] We use *interpretation* instead of the term *semantics* which is used in, e.g., [22] as the latter sometimes leads to confusion.

invariant is irrelevant to the desired relations between the prestates and the poststates of the method executions invoked during the update: we say these method executions do not *rely* on the invariant. Particularly relevant to this paper is that re-establishing an invariant sometimes requires the invocation of shared functionality. For instance, re-establishing an invariant might require the calculation of a square root, which (in Java) can be done by an invocation of method sqrt (which, like most methods from the Java API, will not rely on any user-defined invariants). We show a more involved example in section 3. The interpretation of invariants has to account for such unproblematic scenarios.

The central observation underlying our solution is the following. To achieve loose coupling [24], an OO design is often *layered*, where a layer uses functionality of lower layers, but, in principle, not of higher layers. In particular, lower layer methods do not rely on invariants of higher layer objects. Commonly, an object that provides shared functionality is in a lower layer of the design (for instance, a library like the Java API) than the objects that use it. For example, a class that implements the Singleton Pattern [10] provides a global point of access to its instance which provides shared functionality. This class will often be in a lower layer of the design than the classes that use it.

The contributions of this paper are the following:

1. We analyze a state-of-the-art interpretation for invariants that exploits whole/part relations in OO designs and show that it is not suitable for scenarios that involve shared functionality (section 3).
2. We show how the specifier can make layers in the design explicit with a minimum of specification overhead (section 4).
3. We refine the state-of-the-art interpretation for invariants to exploit layers in OO designs (section 4).
4. We discuss the two existing techniques for reasoning about whole/part relations. One (dynamic reasoning) uses an underlying proof system, the other (static reasoning) is less flexible but allows simple syntactic checks like those of a type system. We extend both techniques to capture additional relations in the whole/part hierarchy, and refine them to reason about layer relations as well (sections 5 and 6).
5. We use these reasoning techniques to present a modular verification technique that ensures the refined interpretation of invariants is satisfied. First, we present a semantic decomposition of the verification technique into separate concerns (section 7). Then, for each of these concerns, we introduce proof obligations and syntactic restrictions (section 8). This two-stage approach gives insight into the role of the individual proof obligations and syntactic restrictions, and allows one to change the verification technique used for one of the concerns without affecting the others.
6. Unlike previous work, we consider a programming language that contains constructors, type casts, static fields and static methods (which were an inspiration for the approach).

2 Programming and Specification Language

We consider the specification of invariants in the context of single-threaded Java-like programming languages. The relevant part of the programming language grammar is shown in figure 1. A statement in the language is either a data-dependent control-flow statement ControlFlowS (e.g. a loop), a sequential composition, an assignment, a local variable declaration, or an assert statement. The assert statement is discussed in section 5.2. There are two kinds of assignments: any expression E can be assigned to a local variable (note that method parameters are treated as local variables), and simple expressions can be assigned to a reference r. A simple expression SimpleE is either a reference r, the constant null, a boolean expression BoolE, or a numeric expression NumE. A reference r is either the keyword variable this, a local variable, or a field access r.f. An extension with static fields is presented in section 9.1. An expression E is either a simple expression, a type cast, a method call, a superclass constructor call or an object creation statement. The language contains two types of methods: instance methods and constructors. An extension with static methods is presented in section 9.2. To make the grammar more manageable,

$C \in classnames$, $T \in types$, $m \in methodnames$, $v \in local\ variablenames$, $f \in fieldnames$
$ownmod \in \{$rep, peer, root$\}$

`Stat`	`::=`	`ControlFlowS` | `Stat; Stat` | v = `E` | `r = SimpleE` | $T\ v$ | assert `BoolE`
`E`	`::=`	`SimpleE` | (T)`r` | `r.`$m(\vec{s})$ | $C(\vec{s})$ | `new` $ownmod\ C(\vec{s})$
`SimpleE`	`::=`	`r` | null | `BoolE` | `NumE`
`r`	`::=`	`this` | v | `r.`f

Figure 1: Statement Grammar (where $\vec{s}$ denotes a vector of simple expressions).

the language does not have a return statement. Instead, every method has an implicit keyword variable result that holds the return value. Furthermore, every method returns a value. Superclass constructors and methods with return type void return null, which is assigned to a dummy variable. A superclass constructor call $v = C(\vec{s})$ may only occur as the first statement of a constructor. Details of object creation can be found in sections 3.2 and 5.1 (in particular, the ownership modifier *ownmod* is discussed in section 3.2). For simplicity, there is no object de-allocation and a subclass is not allowed to define a field with the same name as a superclass field (known as field shadowing [25]). To improve readability, our examples use shorthand notations that are self-explanatory.

The semantics of a program is a set of possible program executions. A *program execution* is a sequence of execution states. For a sequence Σ, $\Sigma[i]$ is the i'th element of Σ. Furthermore, $\Sigma[i..j]$ is the consecutive subsequence of Σ with first element $\Sigma[i]$ and last element $\Sigma[j]$. Finally, $\Sigma[i..]$ is the postfix of Σ with $\Sigma[i]$ as the first element. An *execution state* contains a stack and an object store. A *stack* is a partial mapping from stack variables to values. A *stack variable* is either this or a local variable. A *value* is either an object, null, a boolean, or a number. An *object store* is a partial mapping from locations to values. A *location* is an instance field of an object (written $X.f$). The declared type of the location is the declared type of the field. An execution state σ also contains all the relevant context information: this uniquely identifies σ in the program execution. It includes a program counter, and whether or not σ is a visible state. An execution state is *visible* if it is either a prestate or a poststate. In program execution Σ, $\Sigma[i]$ is a *prestate* either if $i = 0$, or if the program counter of $\Sigma[i-1]$ is at a method call statement. $\Sigma[i]$ is a *poststate* if the program counter of $\Sigma[i]$ is at the end of a method.

Matching pre- and poststates mark the first and last states of a method execution: a prestate $\Sigma[i]$ and a poststate $\Sigma[j]$ *match* if $i < j$ and every prestate $\Sigma[k]$, $i < k < j$, is matched by a poststate $\Sigma[l]$, $l < j$. As in Java, every poststate is matched by exactly one prestate, and if a program execution terminates normally, every prestate is matched by exactly one poststate. A consecutive subsequence Σ' of program execution Σ is a *method execution* if there is a prestate $\Sigma[i]$ such that either (1) $\Sigma' = \Sigma[i..j]$ and $\Sigma[j]$ is the poststate that matches $\Sigma[i]$, or (2) $\Sigma' = \Sigma[i..]$ and $\Sigma[i]$ is unmatched in Σ. As is to be expected, every program execution is a method execution. Further details of the semantics are outside the scope of this paper: intuitively, statements behave as their Java counterparts.

A number of definitions is based on the above. *Control is with* object X in execution state σ if $\sigma(\text{this}) = X$. *Control is in* class C in σ if the program counter of σ is in a method of C. *Control flows to* X in $\Sigma[i]$ if control is with X in $\Sigma[i]$, and either $\Sigma[i]$ is a prestate, or $\Sigma[i-1]$ is a poststate. Notice that control also flows to X when a method executed on X invokes another method on X. X is *constructing* in σ if the program counter of σ is in a constructor and control is with X in σ. X is *non-constructing* in σ if X is allocated and not constructing in σ, so either X's construction is completed or control is with an object (indirectly) called by a constructor of X. Two execution states σ and σ' *differ* on location $X.f$ if $\sigma(X.f) \neq \sigma'(X.f)$. Execution states σ and σ' *differ* on object X if there is a field f such that σ and σ' differ on $X.f$. Let Σ be a sequence of execution states. *type*(X) denotes the dynamic type of X. $D \subset C$ denotes that D is a proper subclass of C. *super*(C) denotes the direct superclass of C.

Few of the details of the specification language are relevant to this paper. For now, only the specification of class invariants is important. Other constructs (in particular, pre- and postconditions) are left implicit. A *class invariant* inv_C is associated with every class C, by means of the specification construct inv BoolE (if the construct is omitted, inv_C is *true*).

Object invariant $inv_C(X)$ *holds* in execution state σ if *type*$(X) \subseteq C$ and inv_C holds in $\sigma[\text{this} \mapsto X]$ (the

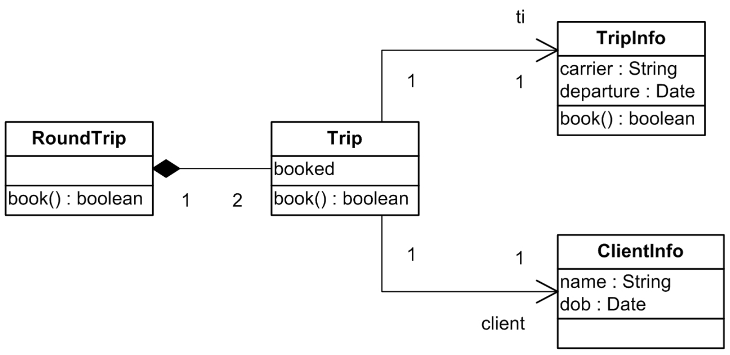

Figure 2: Conceptual Design of a Travel Agent Administration System

state that is like σ, except that its stack maps this to X). X *is consistent for* set of classes S in σ if $inv_C(X)$ holds in σ for every class $C \in S$. $[E, C]$ denotes the set of classes $\{D \mid E \subseteq D \subseteq C\}$. $[E, C\rangle$ denotes the set of classes $\{D \mid E \subseteq D \subset C\}$. X *is consistent* in σ if X *is consistent for* $[type(X), \mathsf{Object}]$ (or equivalently, if $inv_C(X)$ holds in σ for every class C, $type(X) \subseteq C$).

3 Problem Analysis

This section discusses two existing interpretations of invariants and their limitations. In section 3.1, the classical interpretation of invariants and its limitations are discussed. Section 3.2 discusses an existing, state-of-the-art interpretation that addresses some of these limitations. Finally, section 3.3 argues that this interpretation can be improved and sketches how this can be accomplished: the idea behind the paper.

3.1 Classical Invariant Interpretation

The classical interpretation of invariants [22] is based on visible states.

Classical Invariant Interpretation (CII): Program execution Σ satisfies the *Classical Invariant Interpretation* if, for every visible state σ in Σ, for every non-constructing object X, X is consistent.

Object construction deserves special treatment as an object X is (in general) not consistent in the pre- and poststates of its constructors. This treatment is postponed until section 4. The CII is a suitable interpretation for *local* invariants, i.e., invariants that only refer to the state of a single object [22]. However, the consistency of one object often depends on the state of other objects, for instance when the object is a composition of other objects. Many executions of designs that include non-local invariants do not satisfy the CII. As an example, consider the design in figure 2, which represents a simplified travel agent administration system. A ClientInfo object contains the personal information of a specific client of the travel agent, like the client's name and date of birth. A TripInfo object contains the information that pertains to a specific trip that is offered by the travel agent, like the carrier and the arrival and departure times for that trip. TripInfo's method book(ClientInfo c) contacts the carrier's booking system and attempts to reserve a seat for client c. It returns true if the reservation is made, and false otherwise. A Trip object contains the information that pertains to a specific trip selected by a specific client. In this simple example, a Trip object additionally only stores whether or not the trip has already been booked. A RoundTrip consists of two Trips, one outbound and one inbound. Figure 3 shows an implementation of classes Trip and RoundTrip. An invariant has been specified for RoundTrip. This non-local invariant relates the state of RoundTrip's parts. It ensures that a client is not faced with a RoundTrip of which only one leg is booked.

Program executions in which a RoundTrip is successfully booked do not satisfy the CII. Consider an execution of method book on a RoundTrip R. This execution invokes R.outbound.book(). In the poststate

```
class Trip {
  root TripInfo ti; root ClientInfo client; boolean booked = false;
  Trip(root TripInfo t, root ClientInfo c) {
    this.ti = t; this.client = c; this.booked = false;
  }
  boolean book() {
    this.booked = this.ti.book(this.client); return this.booked;
  }
  void cancel() {
    this.ti.cancel(this.client); this.booked = false;
  }
}
class RoundTrip {
  rep Trip outbound; rep Trip inbound;
  inv this.outbound.booked == this.inbound.booked;
  RoundTrip(root TripInfo out, root TripInfo in, root ClientInfo c) {
    this.outbound = new rep Trip(out,c);
    this.inbound = new rep Trip(in,c);
  }
  bool book() {
    bool b = this.outbound.book();
    if (b) { b = this.inbound.book();
             if (!b) { this.outbound.cancel(); }}
    return b;
  }
}
```

Figure 3: Trip and RoundTrip – Hierarchically structured code annotated with ownership information (rep and root, see section 3.2) and an invariant.

of the execution of this method, the invariant of R does not hold. This means it also does not hold in the prestate of R.inbound.book(). Note that booking the inbound Trip re-establishes the invariant of R: in the poststate of R.book(), R is consistent.
The above represents a common scenario: an invariant of an object X is temporarily violated while the parts of X are updated, but this is unproblematic because the design accounts for the inconsistency. The problem is that the CII is too restrictive to allow such scenarios. It has to be refined based on observations of common scenarios. In this case, the design allows for the inconsistency due to a *subordinate relation* between the RoundTrip object and its parts. A subordinate is defined as follows:

subordinate: Object Y is a *subordinate* of object X if (1) the consistency of X is not relied on when control flows to Y, and (2) Y is consistent when control flows to X.

We make the following observation:

observation 3.1: Commonly, if object Y is a part of object X, then Y is a subordinate of X.

3.2 Ownership and the Relevant Invariant Interpretation

Observation 3.1 is reflected in the *ownership technique* [22]. This technique relies on the notion of *ownership* (see for instance [21] and [6]) to make subordinate relations explicit[3]. The idea is that an object owns its subordinate parts. Objects that are not a subordinate part of another object are owned by the special purpose owner **root** (this slightly deviates from the definition of ownership in [22]).

ownership: The set of *owners* consists of the set of objects and the special purpose owner **root**. In any given state, every allocated object is *directly owned* by exactly one owner. This relation is acyclic. The *owned* relation is the transitive closure of the directly owned relation. Finally, $owner(\sigma) = O$ if control is with an object that is directly owned by owner O in execution state σ.

In our language (see section 2), the direct owner of an object is determined by the *ownership modifier ownmod* of an object creation statement $v =$ new *ownmod* $C(\dots)$. Consider a program execution Σ.

[3]Actually, ownership is typically used to enforce *encapsulation* (see section 7). However, the notion of encapsulation is relevant to the verification, but not to the interpretation of invariants.

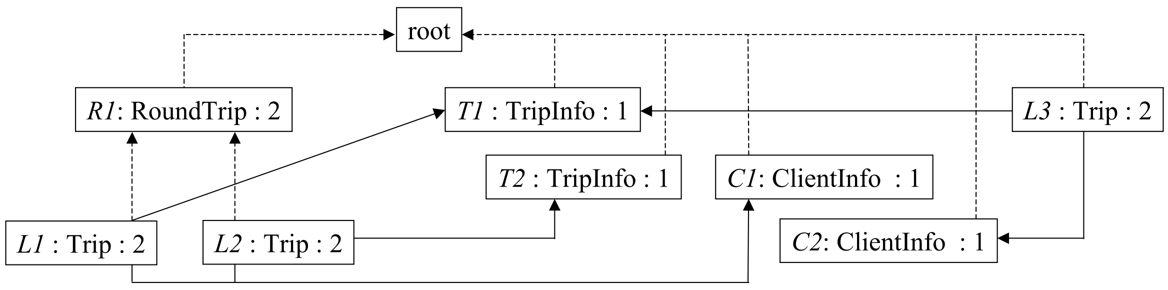

Figure 4: Possible Travel Agent object configuration. Objects refer to their direct owner with a dashed arrow (see section 3.2). Solid arrows refer to objects that provide shared functionality. Objects have a name, a class and a layer (see section 4)

Assume that the program counter in $\Sigma[i]$ is at a statement v = new *ownmod* $C(\ldots)$. Then there is an object X that is constructing in $\Sigma[i+1]$. Assume that control is with object Y with direct owner O in $\Sigma[i]$. The direct owner of X is Y when *ownmod* is rep, O when *ownmod* is peer, and **root** when *ownmod* is root. Note that the default modifier is peer and can be omitted. Ownership transfer [17] is not considered in this paper: the owner of an object X is determined when X is allocated and cannot be changed afterwards.

In figure 3, ownership is expressed by the ownership modifiers in attribute and parameter declarations and in object creation statements. In particular, the keyword rep makes explicit that a RoundTrip directly owns its inbound and outbound Trips (see section 6.1). The TripInfo ti and ClientInfo client of a Trip are owned by **root**, as indicated by the keyword root. Figure 4 shows a possible configuration for the travel agent example. In this configuration, there is a client $C1$ that has selected a RoundTrip $R1$, and a client $C2$ that has selected a Trip $L3$. $R1$'s outbound Trip $L1$ and inbound Trip $L2$ are part of, and owned by, $R1$. The other objects are not part of an object: they are directly owned by **root**. $L3$ and $L1$ share the same TripInfo: $C1$ and $C2$ travel together on $C1$'s outbound Trip. The ownership technique replaces the CII with the weaker *Relevant Invariant Interpretation* (RII) [22]. Roughly, the RII states the following. If control is with an object Y with direct owner O in a visible state σ, then all objects owned by O are consistent in σ.

Relevant Invariant Interpretation (RII): Program execution Σ satisfies the *Relevant Invariant Interpretation* if, for every visible state σ in Σ, for every non-constructing object X, if $owner(\sigma)$ owns X, then X is consistent.

Consider an execution of method book on RoundTrip $R1$. Unlike the CII, the RII allows for the inconsistency of $R1$ in the poststate of the execution of book on $R1$'s outbound Trip and in the prestate of the execution of book on $R1$'s inbound Trip (as $R1$ is the owner of these states). Unfortunately, the RII is not suitable for designs like the travel agent example. As the observations in the next section show, the RII sometimes requires certain objects to be consistent when they are not, and does not always guarantee the consistency of certain objects when they are intended to be consistent.

3.3 Class Level Subordinate Relations

Subordinate relations do not just occur between a whole and it parts. The RII (section 3.2) can be improved when such relations can be identified and made explicit. In this section, we identify a category of subordinate relations at the class level. To achieve loose coupling [24], there typically is a layering of the classes used in a design, where a layer uses functionality of its own layer and lower layers, but has little or no dependency on higher layers. More specifically, we make the following observation.

observation 3.2: Commonly, there are classes C and D in a design such that any object of C is a subordinate of any object of D. In particular, when layers are present in the design and class C is in a lower layer than class D, this property almost always holds.

Especially relevant is that shared functionality is commonly in a lower layer of the design than the classes that use this functionality. Ownership imposes a partial ordering on the object structure. We

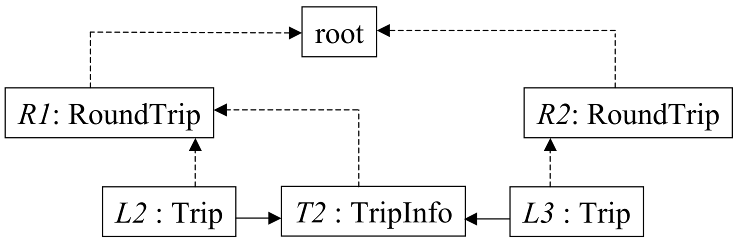

Figure 5: Alternative Travel Agent object configuration (partial).

say an object X provides *shared functionality* if different objects that are not ordered by ownership invoke methods on X. A practical example is the use of a canvas in graphics applications. Note that it is also common that static methods (see section 9.2) provide shared functionality. The travel agent example contains objects that provide shared functionality. In this example, the classes that offer shared functionality (TripInfo and ClientInfo) are in a lower layer of the design than the classes that use this functionality (Trip and RoundTrip). Not accounting for the layering in the interpretation of invariants complicates both specification and verification. More specifically, the RII is well suited to scenarios where re-establishing the consistency of an object X only requires invocations of methods on objects owned by X. However, re-establishing the consistency of X commonly requires the use of lower-layer shared functionality. This is illustrated by the observations below.

observation 3.3: Commonly, to re-establish the consistency of an object X, control needs to flow to a lower-layer object Y with a direct owner that owns X. Given observation 3.2 (and the definition of subordinate), the consistency of X is not relied on when control flows to Y. The RII, however, is not satisfied: it requires that X is consistent when control flows to an object with a direct owner that owns X.

For instance, consider the configuration in figure 4. Successfully booking RoundTrip $R1$ requires an execution of book on TripInfo $T2$ (invoked by $R1$.inbound.book()) such that $R1$ is inconsistent in the prestate of this execution. The RII is too strong to allow this inconsistency.

observation 3.4: Commonly, when control flows to an object X with direct owner O, consistency of a lower layer object Y that is not owned by O is relied on. Given observation 3.2, Y is consistent when control flows to X. The RII, however, is too weak to guarantee this as O does not own Y.

Again consider figure 4. An execution of $L2$.book() invokes $T2$.book(), which is likely to rely on the consistency of $T2$ (among others). Due to the layering of the design, we can expect that $T2$ is consistent in the prestate of methods executed on $L2$. However, the RII does not guarantee this consistency.

Note that these problems cannot be remedied even by a counter-intuitive use of ownership. For instance, consider the alternative configuration of figure 5. It allows the successful booking of RoundTrip $R1$ given the RII (as $R1$ owns $T2$, $T2$ is consistent in the prestate of $L2$.book(), and $R1$ is not required to be consistent in the prestate of $T2$'s book). However, with this configuration, the property that is specified to hold in the poststate of a method executed on $T2$ is much weaker than necessary. In particular, $T2$'s method book is no longer required to preserve the consistency of $R1$. Furthermore, the solution is unsatisfactory as TripInfo objects offer shared functionality: $T2$ is intended to be consistent in, e.g., a prestate of $L4$'s book method, but this is not guaranteed by the RII.

In the next section, we show how the different layers in a design can be made explicit in the specification, and we present a refined interpretation of invariants that takes advantage of these layers.

4 Layers and the Layered Relevant Invariant Interpretation

In this section, the notion of layers from the previous section is formalized, and used to present a interpretation of invariants that takes the observations of the previous section into account.

layers: There is a totally ordered set of layers Λ. Every class is *in* exactly one layer. A class C is in layer $l \in \Lambda$ (also written $layer(C) = l$) if C contains the specification layer l. An object of class C is in layer l if C is in layer l. Finally, $layer(\sigma) = l$ if control is with an object in layer l in execution state σ.

The additional structure provided by layers is used to replace the RII by a more flexible interpretation that reflects the observations of the previous section. This *Layered* Relevant Invariant Interpretation (LRII) is a conservative extension of the RII in the following sense: when every class in a program P is in the same layer, any execution of P that satisfies the RII also satisfies the LRII, and vice versa. Roughly, the LRII states the following. If control is with an object Y with direct owner O in a visible state σ, and Y is in layer l', then all objects in layers below l', and all objects owned by O in layer l', are consistent (item 1 in the definition below). In addition, if σ is the poststate of a method execution Σ', then any object X that is owned by O and in a layer above l', is at least as consistent as it was in the prestate of Σ' (item 2 in the definition below).

Layered Relevant Invariant Interpretation (LRII): Program execution Σ satisfies the *Layered Relevant Invariant Interpretation* if, for every visible state σ in Σ, for every layer l in Λ, for every non-constructing object X in l,
(1) if either $l < layer(\sigma)$, or ($owner(\sigma)$ owns X and $l = layer(\sigma)$), then X is consistent, and
(2) if σ is a poststate and $owner(\sigma)$ owns X and $l > layer(\sigma)$, then X is at least as consistent in σ as in the prestate matching σ.

Observation 3.2 is reflected in two differences between the RII and the LRII. Consider an object X in a layer l. The LRII is stronger than the RII in the sense that with the LRII, in a visible state σ, X is guaranteed to be consistent if $l < layer(\sigma)$. The LRII is weaker than the RII in the sense that with the LRII, the consistency of X is not required in a prestate σ if $owner(\sigma)$ owns X but $l > layer(\sigma)$.

For Λ, we use double (the floating point numbers of our language). In practice, the use of double means that an empty layer (i.e., a layer no class is in) can always be found in between any two non-empty layers. Another advantage of this existing ordering is that it requires very little specification overhead, in particular given default class layers.

default class layers: If class C does not specify its layer, then (1) if C is Object, C is in layer -1, (2) if C is a Java API class and $super(C)$ is Object, C is in layer 1, (3) if C is a user-defined class and $super(C)$ is Object, C is in layer 2, (4) if $super(C)$ is not Object, C is in the same layer as $super(C)$.

The intuition behind these default values is that classes from the Java API do not rely on user-defined invariants. Furthermore, a user-defined class that is intended to provide shared functionality (see section 3.3) can be explicitly placed in layer 1. When a class does not contain the layer construct, that class is in its default layer. Therefore, user-defined classes Trip and RoundTrip (see figure 3) are in layer 2. Classes TripInfo and ClientInfo (not shown) contain the specification layer 1, which explicitly places them in a lower layer than Trip and RoundTrip.

The differences between the RII and the LRII give the extra flexibility needed for designs like that of the travel agent example. Again consider figure 4. Given the LRII, the specification expresses that TripInfo $T1$ and ClientInfo $C1$ are consistent in the prestate of an execution of method book on one of $R1$'s Trips. Furthermore, $R1$ is not required to be consistent in the prestate of $T1$.book(). Note that the LRII guarantees that any invariant of $R1$ that *does* hold in a prestate of $T1$.book, also holds in the poststate, as $R1$ must be at least as consistent in the poststate as in the prestate. In particular, in a poststate σ, an object X owned by $owner(\sigma)$ is consistent if it was consistent in the prestate matching σ.

To deal with object creation, we have to consider (1) invariants of objects that are constructing, and (2) the consistency of objects that are created during a method execution. We make the following observations. For (1), when a new object X is created, none of X's invariants can be expected to hold. Superclass constructors are called before subclass invariants can be initialized. Therefore, the invariants of X cannot be expected to hold in the prestate of superclass constructor executions, and subclass invariants of X cannot be expected to hold in the poststate of a superclass constructor. For (2), it makes sense that an object X that is created during a method execution Σ is consistent in the poststate σ of Σ, even if X is created in a layer above $layer(\sigma)$, or not owned by $owner(\sigma)$ (although the latter is only possible if our language is extended with ownership transfer or object creation with an arbitrary owner). These observations lead to the following definition.

LRII-c: Program execution Σ satisfies *LRII-c* if, for every poststate σ in Σ, for every object X,
1) if X is constructing and control is in class C, then X is consistent for $[C, \text{Object}]$, and
2) if X is not allocated in the prestate matching σ, then X is consistent.

This concludes the first part of this paper. In section 3, we identified problems with the verification of the running example and determined that the interpretation of invariants was the origin of these problems. At the semantical level, the problems have been solved by the introduction of the LRII. Object creation has been accounted for by the LRII-c. To establish the LRII and the LRII-c, one has to reason about layer and ownership relations. In the second part of this paper, we discuss how this can be done. For reasoning about ownership relations, two techniques have been suggested.

1. Static Reasoning [22]: extend the type system to encode ownership relations. The ownership technique uses static reasoning to formulate its proof obligations. The main advantage of static reasoning is that it is *lightweight*: simple syntactic checks (like those used by Java's type system) suffice to establish the desired properties.
2. Dynamic reasoning [8]: encode ownership relations using an auxiliary field that occurs in specifications, but not in regular program statements. In [8], it is shown how to use either static reasoning or dynamic reasoning to establish a given property. Compared to static reasoning, the downside of dynamic reasoning is that it requires more specification overhead and is not lightweight. However, it is more flexible and can be used when static reasoning does not suffice.

In section 5, we discuss dynamic reasoning and extend it to be able to reason dynamically about layer relations. In section 6, we discuss the type system used for static reasoning in [22] and [8]. We first extend this type system to be able to identify two additional ownership relations that are relevant in the context of the LRII. Then we extend the type system further, to reason about layer relations.

5 Dynamic Reasoning

In this section, a dynamic encoding of ownership and layer relations (based on the work in [8] and [17]) is introduced. Then, the role of the assert statement is discussed and illustrated by an example.

5.1 dynamic encoding

When ownership and layer relations are encoded into basic OO concepts, these relations can be reasoned about using any existing OO proof system. The auxiliary fields owner of type Object and layer of type double encode the direct owner and the layer of an object, respectively. These fields are treated as fields defined in class Object. Auxiliary fields may not occur in statements other than the assert statement (i.e., they cannot be read from or written to). Both fields are set when an object is created. Consider an execution state σ in which control is with object Y with direct owner O. When v = new *ownmod* $C(\dots)$ is executed from σ, this allocates a fresh object X of class C, sets X's owner and layer field, invokes the constructor on X and then assigns X to v. X.layer is set to the layer of class C. X.owner is set to Y when *ownmod* is rep, O when *ownmod* is peer, and null when *ownmod* is root. For more details and practical aspects of the dynamic encoding, we refer to [8, 3, 17]. It is obvious that the dynamic encoding establishes the following lemma.

Lemma 5.1 *For every program execution, for every execution state σ, for every allocated object X,*

(1)	X.owner == null *holds*	*if and only if*	**root** *is the direct owner of X,*
	X.owner != null *holds*	*if and only if*	$\sigma(X$.owner$)$ *is the direct owner of X,*
(2)	X.layer == l *holds*	*if and only if*	X *is in layer l.*

5.2 the assert statement

We present the relevant proof obligations in a way that is orthogonal to the other concerns of a proof system. To this end, the statement assert BoolE is included in the programming language (see also [16]), where boolean expression BoolE is side-effect free. A boolean expression in an assert statement may mention owner and layer fields and contain quantifications (both of which are disallowed in other statements). The assert statement causes the program execution to abort when it is executed from a state in which the boolean expression does not hold (and has no effect otherwise).

To guarantee that a certain property holds when the program counter is at a certain point, we augment the original program text with an assert statement. When a method contains a fragment assert B;S, we say that statement S is *guarded* by boolean expression B. Then, when the program execution is at S, B holds.

Let P be a program and let P' be the same program but augmented with statements assert B where necessary. If, for every execution of P', B holds in any execution state with a program counter that is at assert B, then P' is functionally equivalent to P. Following this approach, the question of how to prove that B holds at every assert B can be treated as a separate concern. Our proof obligations for dynamic reasoning can thus be formulated as requirements to insert certain assert statements.

The use of the assert statement is illustrated by the following example. Let statement $v = r.m(\ldots)$ be guarded by r.owner == this && this.layer >= r.layer. Consider two consecutive states σ and σ' in program execution Σ such that the program counter of σ' is at the method call statement. Then the program counter of σ is at the guarding assert statement. Assume that control is with X in σ, and that r refers to Y in σ. As the execution did not abort in σ, r.owner == this && this.layer >= r.layer must hold in σ. As an assert statement does not change the heap or the stack, r.owner == this && this.layer >= r.layer also holds in σ'. Therefore, in σ', X directly owns Y, and X is not in a lower layer than Y (due to lemma 5.1).

6 Static Reasoning

In this section, we introduce a lightweight approach that significantly reduces the specification and verification overhead that is required given only dynamic reasoning. To this end, section 6.1 discusses an existing type system that captures ownership relations statically. We extend this type system to capture two additional ownership relations. Then, in section 6.2, additional type-correctness rules are introduced that allow the type system to capture layer information as well. This allows one to establish layer and ownership properties with simple syntactic checks.

6.1 capturing ownership relations

The ownership technique relies on the Universe Type System (UTS) [21, 22, 8] to capture ownership relations statically. In this section, we show how the UTS uses ownership modifiers to capture these relations. To effectively reason about the LRII, we extend the type system with the ownership modifiers root and owner. We show how to determine the ownership modifier of an expression in this *extended* Universe Type System (eUTS) and briefly discuss its type-correctness rules (as the rules of the eUTS closely resemble the rules of the UTS).

The UTS distinguishes three kinds of references [8]: (1) references between objects with the same direct owner (peer references), (2) references from an object X to an object directly owned by X (rep references), and (3) references between objects with arbitrary direct owners (any references[4]). This classification is expressed in the UTS by adding an ownership modifier peer, rep or any (respectively) to each reference type. For instance, the type rep T is the type of references pointing to objects of class T owned by this. The default modifier is peer and can be omitted. peer T and rep T are subtypes of the corresponding type any T. A type with a peer modifier is referred to as a *peer type* (and likewise for the other ownership modifiers). The ownership modifier of a field f or a local variable v is the ownership modifier of the declared type of f or v.

We introduce two additional kinds of references: (1) references from an object X to an object owned by X (owned references), and (2) references to objects directly owned by **root** (root references). These

[4]also referred to as readonly references in earlier publications

kinds of references are identified by the additional ownership modifiers owned and root. Types owned T and root T are subtypes of the corresponding type any T. Type rep T is a subtype of the corresponding type owned T (as an object owns every object it directly owns). That owned references are relevant follows from the definition of the LRII (see section 4). Root references are relevant because shared functionality (see section 3.3) is often directly owned by **root**.

Like the Java type system, the eUTS ensures that if a reference of a type C refers to an object X, then $\mathit{type}(X) \subseteq C$. Additionally, a type-safety property regarding ownership is established. Informally, this property is the following. If reference r with ownership modifier ownmod refers from object X to object Y in execution state σ, then $\mathit{os}(X, \mathit{ownmod}, Y)$ holds in σ, where

$\mathit{os}(X, \mathit{ownmod}, Y)$ holds in σ if	either	ownmod is owned	and X owns Y,
	or	ownmod is rep	and X directly owns Y,
	or	ownmod is peer	and the direct owner of X directly owns Y,
	or	ownmod is root	and **root** directly owns Y,
	or	ownmod is any.	

We formalize this property at the level of the individual parts of a reference (i.e., stack variables and locations).

ownership safety:[5]
Location $X.f$ is *ownership safe* in execution state σ if: if $X.f$ has a declared type $\mathit{ownmod}\ C$, and $\sigma(X.f)$ is defined, then $\mathit{os}(X, \mathit{ownmod}, \sigma(X.f))$ holds in σ.

If control is with an object X in execution state σ, then stack variable s is *ownership safe* in σ if: if s has a declared type $\mathit{ownmod}\ C$, and $\sigma(s)$ is defined, then $\mathit{os}(X, \mathit{ownmod}, \sigma(s))$ holds in σ.

Execution state σ is *ownership safe* if every location and every stack variable is ownership safe in σ. Method execution Σ is *ownership safe* if every execution state of Σ is ownership safe.

Notice that a location or variable that is not mapped to an object is ownership safe. The type-correctness rules of the eUTS establish ownership safety. These rules differ slightly from those of a standard type system. As in Java, an assignment is type correct only when the type of the right-hand side expression is a subtype of the type of the left-hand side variable. To this end, an ownership modifier is associated with each expression E that is not of a primitive type. ownmod(E), defined below, yields the ownership modifier of the static type of E. In this definition, A represents either a field f, or a method call $m(\ldots)$. The ownership modifier of a method call $m(\ldots)$ is the ownership modifier of the return type of $m(\ldots)$.

shape of E	ownmod(E)
new $\mathit{ownmod}\ C(\ldots)$	ownmod
$(\mathit{ownmod}\ C)r$	ownmod
this	peer
local variable v	ownership modifier of v
this.A	ownership modifier of A
r.A, r different from this	$\mathit{ownmod}(r) \oplus$ ownership modifier of A (see below)

When r differs from this, the ownership modifier of an expression $r.A$ is determined using the ownership combinator $\oplus$ defined below. For example, a reference this.f.g (r=this.f) such that f is of a rep type has a rep modifier when g is of a peer type. It has an owned modifier when g is of a rep type, and an any modifier when g is of an any type.

	$\mathit{omr} \oplus \mathit{omA}$	peer	root	any	rep	owned	omA
omr	peer	peer	root	any	any	any	
	root	root	root	any	any	any	
	any	any	root	any	any	any	
	rep	rep	root	any	owned	owned	
	owned	owned	root	any	owned	owned	

[5]In section 9.1, these definitions are extended to account for static fields.

The rationale behind this table is as follows. Consider a reference $r.f$ where r refers from X to Y and $Y.f$ refers to Z. Let $\mathit{ownmod}(r) = \mathit{omr}$ and let f be of declared type $\mathit{omA}\ C$. If omA is peer, then Y and Z have the same direct owner and the ownership relation between X and Z is the same as that between X and Y. Therefore, $\mathit{ownmod}(r.f) = \mathit{ownmod}(r)$. If omA is root, then Z is directly owned by **root**, independent of omr. Therefore, $\mathit{ownmod}(r.f) =$ root. If omA is any, then Z has an arbitrary direct owner, i.e., the ownership relation between the direct owners of Y and Z is unknown. Then the same is true for X and Z. Therefore, $\mathit{ownmod}(r.f) =$ any. If omA is rep or owned, then Y (directly) owns Z. Then X owns Z if X owns Y (i.e., $\mathit{ownmod}(r.f) =$ owned). If X does not own Y, the relation can only be expressed by any.

As the ownership modifier of a field is relative to the object the field belongs to, one has to be careful when assigning to fields of objects other than the this object. For instance, consider a class Node with a field rep Node next. Then this.next has static type rep Node and this.next.next has static type owned Node. Although rep Node $\subseteq$ owned Node, assignment this.next.next = this.next should not be allowed as this.next.next must refer to an object that is directly owned by this.next. Assignments that do not preserve ownership safety (such as the one above) are prevented by the following type-correctness rule.

syntactic restriction SR6.1: Every assignment $r.f =$ SimpleE is such that either $\mathit{ownmod}(r.f) \notin \{$any, owned$\}$, or r is this, or the ownership modifier of f is any.

This rule resembles the one in [8], but accounts for owned and is more liberal ([8] additionally aims for a strong encapsulation property and therefore disallows assignments where $\mathit{ownmod}(r.f)$ is any altogether). For simplicity, we do not weaken this rule further, and do not show how dynamic reasoning can be used in cases where it is not met.

Method calls require a similar treatment, as the ownership modifier of a formal parameter of a method is relative to the object on which the method is called. For the purpose of the type-correctness rules of the eUTS, a method call $r.m(s_0, \ldots, s_n)$ is treated as a series of assignments $r.p_0{=}s_0, \ldots, r.p_n{=}s_n$, where $p_0, \ldots, p_n$ are the formal parameters of m. So, if the static type of p_i $(0 \leq i \leq n)$ is $o\ C$, and the static type of s_i is $o'\ D$, then $o'\ D \subseteq (\mathit{ownmod}(r) \oplus o)\ C$. Furthermore, if $(\mathit{ownmod}(r) \oplus o) \in \{$any, owned$\}$, then either r is this, or o is any. For the kind of dynamic reasoning that is needed for a downcast we refer to [8].

The above leads to the following lemma.

Lemma 6.1 *If program P meets the type-correctness rules of the eUTS (including SR6.1), then every execution of P is ownership safe.*

The proof is omitted given the similarities with the work in [8] and [21] (for which a very similar lemma is proven). The use of the eUTS is illustrated in section 6.2.

6.2 capturing layer relations

This section shows how to use the layer of the static type of a reference to determine the layer of the object it refers to. More specifically, the following type-safety property is ensured: if a reference r with static type $\mathit{ownmod}\ C$ refers to an object of class D, then C and D are in the same layer, unless r is this, or ownmod is any. Furthermore, if r is this, then D is not in a lower layer than C. The latter allows the static type of this to be used as a lower bound on the layer of an execution state (without which no invariants can be assumed to hold given the LRII). As with reasoning statically about ownership, any is used as a mechanism to deal with atypical cases.

To formalize the above, the type-safety property is formulated at the level of the individual parts of a reference (i.e., stack variables and locations). Then, the type-correctness rules that establish the property are introduced. Finally, the use of static reasoning is illustrated by an example.

layer safety:[5]

this is *layer safe* in execution state σ if: if control in σ is in class C and with an object of class D, then $\mathit{layer}(D) \geq \mathit{layer}(C)$.

Location $X.f$ is *layer safe* in execution state σ if: if $X.f$ has declared type $ownmod\ C$, and $\sigma(X.f)$ is defined, then either $ownmod$ equals any, or $\sigma(X.f)$ is in layer $layer(C)$.

Local variable v is *layer safe* in execution state σ if: if v has declared type $ownmod\ C$, and $\sigma(v)$ is defined, then either $ownmod$ equals any, or $\sigma(v)$ is in layer $layer(C)$.

Execution state σ is *layer safe* if every location and every stack variable is layer safe in σ. Method execution Σ is *layer safe* if every execution state in Σ is layer safe.

Notice that a location or local variable that is not mapped to an object is layer safe. To ensure that all possible executions of a program are layer safe, a number of additional type-correctness rules are introduced (in the form of syntactical restrictions and proof obligations). These are motivated as follows.

motivation: Suppose control in execution state σ is in class C and with an object of class D. Then D is a subclass of C. Due to syntactic restriction SR6.2, $layer(D) \geq layer(C)$, which ensures layer safety of this. Now consider locations and local variables. These can only be changed by assignments, context switches, and object allocation. The latter cannot violate layer safety of a location or local variable because (1) a newly allocated location is not mapped to an object, and (2) no location or local variable is mapped to the newly allocated object.

That leaves assignments and context switches. To treat these uniformly, a method call is treated as a series of assignments of actual to formal parameters in the remainder of this section (see also section 6.1). Consider an assignment $r = E$ to a local variable or location of declared type $ownmod\ C$ from a layer-safe execution state σ. Then either $ownmod(r) = ownmod$, or r is $r'.f$ (with f of declared type $ownmod\ C$) and r' differs from this. Assume that E has static type $ownmod'\ D$ and is mapped to an object in layer l. If $ownmod =$ any, then layer safety is preserved (by definition). Now assume $ownmod \neq$ any. Then SR6.1 ensures that $ownmod(r) \neq$ any. Then standard type correctness ensures that $ownmod' \neq$ any. Three cases can be distinguished. E is either (1) this, or (2) a method call or a reference other than this, or (3) a typecast $(ownmod'\ D)r'$.

For convenience, SR6.3 forbids case (1). Note that a program in which this appears as the right-hand side of an assignment can be rewritten to a program in which it does not, by using a statement like $v =$ (peer D)this, after which v can be used instead of this. In case (2), layer safety ensures that $l = layer(D)$ (if E is a reference, proof by structural induction on references is straightforward, and if E is a method call, proof is only slightly more involved as a method call returns a reference result). Then SR6.4 ensures that $layer(C) = layer(D)$. Therefore, $layer(C) = l$ and the assignment preserves layer safety. In case (3), proof obligation PO6.1 ensures that either (A) r' is an any reference of type D' that differs from this and $layer(D) = layer(D')$, or (B) r.layer == layer(D) holds in σ. In case (A), layer safety ensures that $l = layer(D')$, in case (B) the same is guaranteed by the guard. In either case, the reasoning of case (2) applies.

syntactic restrictions:
SR6.2: If $C = super(D)$, and class D contains the specification layer l, then $l \geq layer(C)$.
SR6.3: this does not appear as the right-hand side of an assignment.[6]
SR6.4: If the left-hand side and right-hand side of an assignment have static type $ownmod\ C$ and $ownmod'\ D$, then either $ownmod$ is any, or $layer(C) = layer(D)$.

proof obligation PO6.1: For every statement $v = (ownmod'\ D)r$ such that the static type of r is $ownmod\ C$, either $ownmod'$ is any,
or $ownmod$ differs from any and r differs from this and $layer(C) = layer(D)$,
or the statement is guarded by r.layer == layer(D)

Given the motivation above, the following lemma holds.

Lemma 6.2 *If program P meets the type-correctness rules of the eUTS (including SR6.2, SR6.3, SR6.4 and PO6.1), then every execution of P is layer safe.*

The use of static reasoning is illustrated by the following example. Consider a program execution Σ that is ownership safe and layer safe. Consider an execution state σ in Σ such that the program counter of σ is at a statement $v = r.m(\ldots)$ in a method of class C. If the static type of r is rep D, and

[6]Note that this does not prohibit assignments like $v =$ this.f and $v =$ (peer C)this.

$layer(C) \geq layer(D)$, then r.owner == this && this.layer >= r.layer holds in σ (due to lemmas 5.1 and 6.2). Note that the simple syntactic check above allows the same conclusion as the assert statement used in the example in section 5.2.

This concludes the second part of this paper. Two techniques for reasoning about layer and ownership relations have been introduced. In the third part of this paper, a verification technique is introduced that uses these techniques to establish the LRII. The restrictions imposed by this technique are similar to those of the ownership technique. Furthermore, the technique is lightweight: in typical cases, simple syntactical checks suffice to discard the proof obligations.

7 Establishing the Layered Relevant Invariant Interpretation

The verification technique is presented in two steps. In this section, five properties are identified. If a program execution Σ satisfies these properties, then Σ satisfies the LRII. How to establish these individual properties is a separate concern, which is treated in the next section. In many ways, these properties are similar to those underlying model-based Abstract Data Type (ADT) specifications, where invariants range over the state of the type and where the state of the type is encapsulated from clients of the type [11]. In turn, this allows a form of reasoning that is similar to the data type induction used for ADTs. That is, establishing the LRII reduces to a *local* property, i.e., a property that only considers the object that has control.

A modular verification technique needs to restrict the invariants that are considered [20]. Our technique restricts invariants such that an invariant of object X can only be invalidated when either X, or an object owned by X is modified. More formally, we define the following property of an invariant.

ownership based: Invariant inv_C is *ownership based* in program P if, for every execution Σ of P, for every two consecutive execution states σ and σ' in Σ, for every object X, if $inv_C(X)$ holds in σ but not in σ', then σ and σ' differ either on X, or on an object that is owned by X in σ'.

Note that invariants that are admissible in the ownership technique are also ownership based. Weakening this restriction is discussed in section 10. When all invariants are ownership based, a change of the state of an object X can (only) invalidate invariants of X and its owners. A method can change the state of X either directly, by an assignment to a field of X, or indirectly, by a method invocation. Field assignment is restricted by classical encapsulation (in particular, it is ensured that assignments to $X.f$ only occur when control is with X).

classical encapsulation: Location $X.f$ is *classically encapsulated* in a program execution Σ if, for every two consecutive execution states σ and σ' in Σ, if σ and σ' differ on $X.f$, then control is with X in σ'. A program execution Σ satisfies *classical encapsulation* if every location $X.f$ is classically encapsulated in Σ.

Method invocation is restricted by ownership encapsulation. In particular, it ensures the following two properties: (1) if object X is the direct owner of object Y, then a method execution on Y can only be invoked by a method execution on X or on an object directly owned by X, and (2) a method execution in layer l cannot invoke a method execution in a layer above l.

ownership encapsulation: Program execution Σ satisfies *ownership encapsulation* if, for every two consecutive execution states σ and σ' in Σ, if σ' is a prestate, then
either $layer(\sigma) \geq layer(\sigma')$ and $owner(\sigma) = owner(\sigma')$,
or $layer(\sigma) \geq layer(\sigma')$ and control is with $owner(\sigma')$ in σ,
or $layer(\sigma) > layer(\sigma')$ and $owner(\sigma') = \mathbf{root}$.

Actually, one can also allow any method execution to invoke executions of methods that are *pure* [22, 4, 7]. Executions of pure methods do not have visible states, do not change the state and do not call non-pure methods. For simplicity, pure methods are ignored in this paper. Extending our technique to accommodate pure methods is straightforward.

Roughly said, the three properties above ensure that method calls that cross a boundary of the layer or ownership hierarchy do not violate the LRII. That is, when control is with X, only the state of X can be changed (classical encapsulation). Such a change can only invalidate invariants of X and its owners (as invariants are ownership based). When control flows to an object that is owned by X or in a lower layer than X, the LRII does not require these invariants to hold. Ownership encapsulation forbids method calls that ascend in the layer or ownership hierarchy. That leaves poststates and horizontal call states:

In program execution Σ, $\Sigma[i]$ is a *horizontal call state* if $\Sigma[i+1]$ is a prestate and $\mathit{owner}(\Sigma[i]) = \mathit{owner}(\Sigma[i+1])$ and $\mathit{layer}(\Sigma[i]) = \mathit{layer}(\Sigma[i+1])$.

As mentioned above, establishing the LRII is reduced to establishing a local property that only considers the object that has control. Consider an execution Σ of a program in which all invariants are ownership based. Assume Σ satisfies classical and ownership encapsulation. Then Σ satisfies the LRII if Σ satisfies *local consistency*: for every i, if control is with X in $\Sigma[i]$, and $\Sigma[i]$ is a either a horizontal call state or a poststate, then X is consistent in $\Sigma[i]$. Unfortunately, matters are slightly complicated by the presence of constructors: object X is not always consistent in a visible state in which X is constructing. Therefore, the notion of a relevant horizontal call state is introduced, and local consistency is split into upward local consistency and downward local consistency.

In program execution Σ, horizontal call state $\Sigma[i]$ is *relevant* if there is no object that is constructing in both $\Sigma[i]$ and $\Sigma[i+1]$.

That is, a horizontal call state is relevant when its program counter is not at a superclass constructor call.

upward local consistency: Program execution Σ satisfies *upward local consistency* if, for every execution state σ in Σ, if control is with object X and in class C in σ, and σ is either a relevant horizontal call state or a poststate, then X is consistent for $[C, \mathsf{Object}]$ in σ.

downward local consistency: Program execution σ satisfies *downward local consistency* if, for every execution state σ in Σ, if control is with non-constructing object X and in class C in σ, and σ is either a relevant horizontal call state or a poststate, then X is consistent for $[\mathit{type}(X), C\rangle$ in σ.

Note that, if X is consistent for both $[\mathit{type}(X), C\rangle$ and $[C, \mathsf{Object}]$, then X if consistent. Given these properties, the following theorem holds.

Theorem 7.1 *Consider an execution Σ of a program P in which all invariants are ownership based. If Σ satisfies classical encapsulation, ownership encapsulation and upward and downward local consistency, then Σ satisfies the LRII and the LRII-c.*

A proof of this theorem can be found in appendix A.3.

8 Proof Techniques

In this section, a proof technique is introduced for each of the properties of the previous section.

8.1 establishing ownership based invariants

To ensure that invariants are ownership based, a syntactical restriction is imposed on invariants.

An invariant inv_C that is defined as inv BoolE is *ownership admissible* if BoolE is composed of (quantifications over) primitive values, the usual unary and binary operators (see for instance [21]), and references $\mathsf{this}.f_1 \ldots .f_i$ ($i \geq 0$) such that if $i > 1$, then $\mathit{ownmod}(\mathsf{this}.f_1 \ldots f_{i-1})$ is either rep or owned.

Note that the invariant of RoundTrip in figure 3 is ownership admissible. It contains two references, this.inbound.booked and this.outbound.booked. As fields inbound and outbound are both declared with a rep modifier, both ownmod(this.inbound) and ownmod(this.outbound) yield rep. Therefore, both references are admissible.

Lemma 8.1 *If invariant inv_C is ownership admissible and every execution of program P is ownership safe, then inv_C is ownership based in P.*

Informally, the reasoning is the following. An object invariant $inv_C(X)$ can only be invalidated by changing the value of a reference $\mathsf{this}.f_1 \dots f_i$ that occurs in inv_C. This requires modification of a location $Y.f_{j+1}$, where $\mathsf{this}.f_1 \dots f_j$ ($j < i$), refers from X to Y. If $j = 0$, then $Y = X$. Otherwise, $ownmod(\mathsf{this}.f_1 \dots f_{i-1})$ is either rep or owned (as inv_C is ownership admissible). Then the same is true for $ownmod(\mathsf{this}.f_1 \dots f_j)$ (by definition of $ownmod$, see section 6.1). Then X owns Y (due to ownership safety). A more formal proof can be found in appendix A.4.

When more details of the grammar of boolean expressions are fixed, a weaker admissibility obligation could be imposed. In particular, one can allow quantifications over owned objects. For instance, invariant $\forall$ X : C $\bullet$ (X.owner == this ==> X.f == 4) (meaning that every directly owned C object has an f-field with a value of 4) is ownership based. Likewise, if head is a field of type rep Node, and Node's next field has a rep modifier, then invariant $\exists i \bullet$ (this.head.next^i.val == 4 (meaning that there is a node in the list that has a value of 4) is ownership based.

8.2 establishing classical encapsulation

Classical encapsulation does not require reasoning about layer or ownership relations. If two consecutive execution states σ and σ' in a program execution differ on a location $X.f$, then the program counter of σ is either at a field assignment (to a reference $r.f$ such that r refers to X in σ), or at an object creation statement (that allocates object X). In the latter case, control is with X in σ' by definition (σ' is the prestate of a constructor on X). Therefore, classical encapsulation can be established by imposing a restriction on field assignment r = SimpleE.

syntactic restriction SR8.1: Every assignment r.f = SimpleE is such that r is this.

Note that the code in figure 3 meets SR8.1: only fields of this are assigned to.

Lemma 8.2 *If program P meets SR8.1, then every execution of P satisfies classical encapsulation.*

Proof is straightforward given the reasoning above.

8.3 establishing ownership encapsulation

In this section, two proof techniques that establish ownership encapsulation are introduced. The first uses only dynamic reasoning. The second offers a lightweight solution for programs that also allow static reasoning.

The assert statement statement (see section 5.2) is used in combination with the dynamic encoding of ownership and layer information (see section 5.1) to formulate a straightforward proof obligation that establishes ownership encapsulation. The proof obligation ensures that method call statements are guarded. More specifically, the notion of guarding a method call for encapsulation is introduced. A method call that is guarded for encapsulation does not invalidate ownership encapsulation. Note that the three cases of the definition of ownership encapsulation can be recognized in the definitions below (and that as in Java, && binds stronger than ||).

Statement $v = r.m(\dots)$ is *guarded for encapsulation* if it is guarded by
(r.owner == this || r.owner == this.owner) && this.layer >= r.layer || r.owner == null && this.layer > r.layer

Statement v = new $ownmod$ $C(\dots)$ is *guarded for encapsulation* if

either	$ownmod$ is rep	and it is guarded by this.layer >= layer(C),
or	$ownmod$ is peer	and it is guarded by this.layer >= layer(C),
or	$ownmod$ is root	and it is guarded by this.layer > layer(C) || this.owner == null && this.layer == layer(C).

Statement $v = C(\dots)$ is always *guarded for encapsulation.*

The reason that a superclass constructor call $v = C(\dots)$ is always guarded for encapsulation (i.e, does not have to be guarded) is the following. Superclass constructor calls only occur in constructors. If an execution Σ of a constructor executed on object X invokes the execution of a superclass constructor Σ', then Σ' is executed on X. That is, Σ and Σ' have the same owner and are in the same layer. The definitions above allow for the following proof obligation and corresponding lemma.

proof obligation PO8.1: Every method call statement is guarded for encapsulation[7].

Lemma 8.3 *If program P meets PO8.1, then every execution of P satisfies ownership encapsulation.*

A proof of this lemma can be found in appendix A.5. Guarding all method calls introduces much verification overhead. Given static reasoning (see section 6), the desired property can typically be established by simple syntactic checks. This allows one to omit many of the assert statements that guard method calls and yet end up with a functionally equivalent program. More specifically, the notion of statically meeting encapsulation is introduced. A method call that statically meets encapsulation does not violate ownership encapsulation.

$encap(C, ownmod, D)$ holds if either $layer(C) \geq layer(D)$ and $ownmod$ is peer,
or $layer(C) \geq layer(D)$ and $ownmod$ is rep,
or $layer(C) > layer(D)$ and $ownmod$ is root,

Let the static type of reference r be $ownmod\ D$. Statement $v = r.m(\ldots)$ in a method of class C *statically meets encapsulation* if $encap(C, ownmod, D)$ holds.

Statement $v =$ new $ownmod\ D(\ldots)$ in a method of class C *statically meets encapsulation* if $encap(C, ownmod, D)$ holds.

Again, there are three cases (in the definition of *encap*) that match the three cases of the ownership encapsulation definition. For programs that use static reasoning, PO8.1 can be replaced by the weaker proof obligation below.

proof obligation PO8.2: Every method call statement either statically meets encapsulation[7], or is guarded for encapsulation.

Note that every method call in the code in figure 3 statically meets encapsulation: class Trip calls methods on this.ti of static type root TripInfo, and $layer(\text{Trip}) > layer(\text{TripInfo})$. Class RoundTrip calls methods on its inbound and outbound trips, which have static type rep Trip, and $layer(\text{RoundTrip}) = layer(\text{Trip})$. Static reasoning is only possible in programs that are layer and ownership safe. This leads to the following lemma (a proof can be found in section A.6).

Lemma 8.4 *If every class of program P meets PO8.2, then every execution of P that is ownership safe and layer safe, satisfies ownership encapsulation.*

We conclude with a small remark. Usually, if a method call $v = r.m(\ldots)$ does not meet encapsulation statically, then it is easy to deduce that the call violates ownership encapsulation. In that case, there is no point in guarding the call for encapsulation as the guard is not met (and the program will terminate abnormally, i.e., cannot be verified by the underlying proof system). However, if r has an any modifier, guarding the call can be useful, for instance if dynamic reasoning can be used to determine that r refers to an object that has the same direct owner and is not in a higher layer.

8.4 establishing upward local consistency

In this section, two proof techniques that establish upward local consistency are introduced. The first uses only dynamic reasoning. The second offers a lightweight solution for programs that also allow static reasoning.

Upward local consistency requires that, in an execution of a method of class C, the object that has control is consistent for $[C, \text{Object}]$ in poststates and relevant horizontal call states. Let $upinv_C$ denote the conjunction of the invariants declared in class C and C's superclasses. The object that has control is guaranteed to be consistent for $[C, \text{Object}]$ in states in which $upinv_C$ holds (as field shadowing is disallowed). Therefore, the proof obligation below ensures the desired property for poststates.

[7]in section 9.2, these definition is extended to account for static methods

proof obligation PO8.3: For every method M in class C, the last statement in M is $\textsf{assert}\ upinv_C$;

By guarding method calls, it can be ensured that $upinv_C$ holds in relevant horizontal call states. In particular, a method call that is guarded for consistency does not violate upward local consistency. Note that the definitions below are the result of a straightforward translation of the definition of a relevant horizontal call state using the dynamic encoding.

Statement $v = r.m(\ldots)$ in a method of class C is *guarded for consistency* if it is guarded by $\textsf{this.owner} \mathrel{!=} r\textsf{.owner} \parallel \textsf{this.layer} \mathrel{!=} r\textsf{.layer} \parallel upinv_C$

Statement $v = \textsf{new}\ ownmod\ D(\ldots)$ in a method of class C is *guarded for consistency* if
either $ownmod$ is $\textsf{rep}$,
or $ownmod$ is $\textsf{peer}$ and it is guarded by $\textsf{this.layer} \mathrel{!=} \textsf{layer}(D) \parallel upinv_C$,
or $ownmod$ is $\textsf{root}$ and it is guarded by $\textsf{this.layer} \mathrel{!=} \textsf{layer}(D) \parallel \textsf{this.owner} \mathrel{!=} \textsf{null} \parallel upinv_C$.

Statement $v = C(\ldots)$ is always *guarded for consistency.*

That a superclass constructor call $v = C(\ldots)$ never violates upward local consistency (and is therefore always guarded for consistency) follows immediately from the definition of relevant horizontal call states in section 8. Together, these definitions allow for the following proof obligation and corresponding lemma (that is proven in appendix A.5).

proof obligation PO8.4: Every method call statement is guarded for consistency[7].

Lemma 8.5 *If program P meets PO8.3 and PO8.4, then every execution of P satisfies upward local consistency.*

As is the case with ownership encapsulation (see section 8.3), static reasoning allows for a lightweight solution. To this end, the notion of a statically relevant call is introduced. A method call that is not statically relevant does not lead to a relevant horizontal call state. For a statically relevant call in a class C, dynamic reasoning has to be used to either establish that it does not lead to a relevant horizontal call state, or establish that the object that has control is consistent for $[C, \textsf{Object}]$.

$statrel(C, ownmod, D)$ holds if either $ownmod$ is $\textsf{any}$,
or $ownmod$ is $\textsf{peer}$ and $layer(C) = layer(D)$,
or $ownmod$ is $\textsf{root}$ and $layer(C) = layer(D)$.

Let the static type of reference r be $ownmod\ D$. In a method of class C, statement $v = r.m(\ldots)$ is *statically relevant* if $statrel(C, ownmod, D)$ holds.

In a method of class C, statement $v = \textsf{new}\ ownmod\ D(\ldots)$ is *statically relevant* if $statrel(C, ownmod, D)$ holds.

Statement $v = C(\ldots)$ is never *statically relevant.*

For programs that use static reasoning, PO8.4 can be replaced by the weaker proof obligation and corresponding lemma below.

proof obligation PO8.5: Every statically relevant[7] method call is guarded for consistency.

Note that none of the method calls in the code in figure 3 is statically relevant.

Lemma 8.6 *If program P meets PO8.3 and PO8.5, then every execution of P that is ownership safe and layer safe, satisfies upward local consistency.*

A proof of this lemma can be found in appendix A.8.

8.5 establishing downward local consistency

Let σ be a horizontal call state or poststate in which control is in class C and with non-constructing object X. To satisfy downward local consistency, X must be consistent for $[type(X), C\rangle$ in σ. Establishing this is complicated by the fact that not all subclasses are available at superclass verification time. Therefore, consistency of X for $[type(X), C\rangle$ has to follow from an inductive argument: it is ensured that for *any* execution state σ', if control is with non-constructing object X and in class C, then X is consistent for $[type(X), C\rangle$.

An inductive proof of this property is straightforward when (1) an assignment cannot invalidate a subclass invariant, and (2) a method call in a constructor does not lead to the execution of a method with the same owner and in the same layer (as consistency for $[type(X), C\rangle$ of a constructing object X cannot be guaranteed), and (3) if a subclass invariant holds in the prestate of an invoked method execution Σ, then it also holds in Σ's poststate. Note that a subclass invariant may be temporarily violated by Σ, as long as it has been re-established in Σ's poststate.

To establish (1), it is ensured that (A) a method does not assign to subclass fields, and (B) a subclass invariant does not depend on a superclass field. (A) is established by 're-using' SR8.1: a method can only assign to a subclass field using a typecast, and SR8.1 forbids such assignments. (B) is established by ownership safety, ownership admissibility and the restriction below.

syntactic restriction SR8.2: If invariant inv_C contains reference $\mathsf{this}.f_0 \ldots . f_i$ $(i \geq 0)$, then field f_0 is defined in class C.

Ownership admissibility and ownership safety ensure that if object invariant $inv_C(X)$ contains a reference $\mathsf{this}.f_0 \ldots . f_i$, and $i \geq 1$, then $X.f_0 \ldots . f_j$ $(0 \leq j < i)$ refers to an object owned by X. Therefore, only $X.f_0$ is a field of X. SR8.2 ensures that f_0 is not a field of a superclass of C.

The simplest way to establish (2) is by means of the syntactic restriction below (a weaker proof obligation that utilizes dynamic reasoning as well is omitted).

syntactic restriction SR8.3: There are no statically relevant method calls in constructors.

(3) is established by the use of a form of encapsulation that goes beyond that provided by ownership. Consider an object X of class D. The objects owned by X are divided into *class frames* [17]. There is a frame for every class C such that $D \subseteq C$ (called the C-frame of X). It is ensured that invariants are *frame based*: if object invariant $inv_C(X)$ holds in $\Sigma[i]$, but not in $\Sigma[i+1]$, then $\Sigma[i]$ and $\Sigma[i+1]$ differ either on a field of X defined in class C, or on an object in the C-frame of X. Furthermore, it is ensured that every program execution Σ satisfies *frame encapsulation*: if $\Sigma[i]$ is a prestate in which control is with an object in the C-frame of X, then in $\Sigma[i-1]$ control was either with an object in the C-frame of X, or with X and in C. These two properties are established by combining the proof techniques presented so far with a number of additional syntactic restrictions refined from those of the ownership technique.

syntactic restrictions:
SR8.4: Every field of a rep type is private.
SR8.5: No field or local variable is of an owned type.
SR8.6: Every method with a parameter or result of a rep type or an owned type is private.
SR8.7: Every statement $v = (ownmod\ C)r$ is either such that r is not of an any type, or such that *ownmod* differs from rep and from owned.
SR8.8: No statement $v = r.m(\ldots)$ is such that r is of an any type.

Note that fields of owned types have been forbidden to simplify the proof (which can be found in appendix A.9). Requiring such fields to be private would be sufficient as well. The proof depends on two intermediate encapsulation properties.

(P1) If $\sigma(X.f) = Y$, and field f is of a rep or owned type and is defined in a class C, then object Y is in the C-frame of object X.

(P2) If Y is an object in the C-frame of object X, and $\sigma(Y.f) = Z$, and field f is of a rep, owned or peer type, then object Z is in the C-frame of X.

For the purpose of these definitions, if control is with an object X an in a class C, then local variables are treated as fields of X defined in C. Roughly, the proof of these intermediate properties is the following. When control is with X in C, SR8.4 and SR8.5 prevent X from reaching owned objects outside the C-frame via rep or owned fields. SR8.6 prevents the exposure of owned objects outside the C-frame to X via a method return. If X can reach an object outside the C-frame via an any reference r, SR8.7 prevents X from casting r to a rep or owned type. When control is not with X or not in C, SR8.4 prevents the update of owned and rep fields of X defined in C. SR8.6 prevents the exposure of owned objects outside the C-frame to X via (the parameters of) a C-method call on X.

That invariants are frame based follows almost directly from ownership admissibility of inv_C, SR8.2, and the intermediate properties. The proof outline for frame encapsulation is more involved: if $\Sigma[i]$ is a prestate in which control is with an object Y in the C-frame of X, then the program counter of $\Sigma[i-1]$ is at a method call $v = r.m(\ldots)$. Due to SR8.8, r is not an any reference. As Y is owned by X, r is not a root reference (ownership safety). Then r is either a peer or a rep reference (as the method call statically meets encapsulation). Let control be with an object Z in $\Sigma[i-1]$. Then, due to ownership safety, either (A) Z is owned by X, or (B) $Z = X$. In case (A), Z can reach Y via r, therefore P2 ensures Z is in the C-frame. In case (B), r has a subreference v or $\mathsf{this}.f$, that refers to an object Z'. As Z' can reach Y (r is mapped to Y), P2 ensures that Z is in the C-frame. Then P1 and SR8.4 ensure that control is with C in $\Sigma[i-1]$.

Now we can present the outline for case (2). Let control in $\Sigma[i-1]$ be with X and in class C. Let $\Sigma[i..j]$ be a method execution. Let $inv_D(X)$ ($D \subset C$) hold in $\Sigma[i]$. If $inv_D(X)$ holds in $\Sigma[k] \in \Sigma[i..j-1]$, but not in $\Sigma[k+1]$, then $\Sigma[k]$ and $\Sigma[k+1]$ differ either (A) on a field of X defined in class D, or (B) on an object Y in the D-frame of X (inv_D is frame based). In either case, there is a state $\sigma \in \Sigma[i..k]$ in which control is with X and in a class E, $E \subseteq D$: in case (A), control in $\Sigma[k]$ is with X and in a class E, $E \subseteq D$ (classical encapsulation, fields of subclasses cannot be assigned to). In case (B), control is with Y in $\Sigma[k]$ (classical encapsulation). Then in the state $\Sigma[l]$ from which Y was called, control was either (C) with X and in C, or (D) with another object in the C-frame of X (frame encapsulation). In either case, $i \geq l < k$. In case (D), to find σ, the reasoning of case (B) can be applied again to $\Sigma[l]$. Given this state σ, we know there must be a poststate $\sigma' \in \Sigma[k..j]$ in which control is with X and in a class E, $E \subseteq D$. Due to PO8.3, $inv_D(X)$ is re-established in σ'. Given this reasoning, it can be concluded that $inv_D(X)$ holds in $\Sigma[j]$. Combining all the above, we formulate the following lemma.

Lemma 8.7 *If every invariant in program P is ownership admissible, and P meets the type-correctness rules of the eUTS, SR8.1-SR8.8, PO8.2, PO8.3, and PO8.5, then every execution of P satisfies local consistency.*

A detailed proof can be found in appendix A.10. Its outline is fairly straightforward: if control in poststate or horizontal call state σ is with X and in C, PO8.3, PO8.5 ensure X is consistent for $[C, \mathsf{Object}]$ in σ. Furthermore, it has been ensured that if control is with X and in class C in execution state σ, then X is consistent for $[type(X), C\rangle$ in σ. If X is consistent for both $[type(X), C\rangle$ and $[C, \mathsf{Object}]$, then C is consistent, which concludes the proof.

9 Static Fields and Static Methods

This section discusses how static fields and static methods can be added to the language. To this end, a number of definitions introduced earlier are extended. Note that the proofs of all lemmas and theorems in this paper (see the appendix) account for the presence of static fields and static methods.

9.1 static fields

The grammar of references (figure 1) is extended in the following way:

$\mathsf{r} ::= \ldots \mid C.f$

The notion of a location (see section 2) is extended: a location is either an instance field of an object, or a static field of a class (written $C.f$). The object store of an execution state maps static fields of a class to values. $\sigma(C.f) = v$ if σ's object store maps location $C.f$ to value v.

It does not make sense for static fields to have a rep type or an owned type, as classes cannot own objects (although an extension that allows this is possible, see [18]). We also do not allow a static field to have a peer type, as a class does not have an owner. In other words, the declared type of a static field must be either a primitive type, or a root type, or an any type.

Extending the definition of the *ownmod* function to account for static fields is straightforward: $ownmod(C.f)$ yields the ownership modifier of the declared type of f. The definitions of ownership safety and layer safety (section 6.1) are extended:

layer and ownership safety (extended):
A location $C.f$ is *ownership safe* in execution state σ if, if f has declared type *ownmod* D, and $\sigma(C.f)$ is defined, then *ownmod* is any, or *ownmod* is root and **root** directly owns $\sigma(C.f)$.

A location $C.f$ is *layer safe* in execution state σ if: if $C.f$ has declared type *ownmod* D, and $\sigma(C.f)$ is defined, then either *ownmod* equals any, or $\sigma(C.f)$ is in layer $layer(D)$.

No additional type-correctness rules are needed for lemmas 6.1 and 6.2. Ownership admissibility (section 8.1) ensures that static fields do not occur in invariants. Therefore, invariants cannot be invalidated by a modification of a static field. The definitions and proof techniques for classical encapsulation, ownership encapsulation, and upward and downward local consistency account for the extension with static fields.

9.2 static methods

Static methods are a fairly straightforward, but important extension to the language. Static methods often provide shared functionality (see section 3.3). For instance, re-establishing an invariant might require the calculation of a square root, which (in Java) can be done by an invocation of method sqrt of class Math from the Java API. A static method call is difficult to deal with given the RII (in particular when the static method relies on invariants). Our approach ensures that a lower layer static method can be called from a higher layer method. Furthermore, it guarantees that objects reachable via a static field or parameter of a root type are consistent in the prestate of a static method execution.

The grammar of expressions (figure 1) is extended in the following way:

E ::= ... | $C.m(\vec{s})$

The extension with static methods requires a number of earlier definitions to be extended. First of all, we extend the definitions of the function *owner* (section 3.2) and *layer* (section 4). If control is in a static method of class C in execution state σ, then $owner(\sigma) = \mathbf{root}$ and $layer(\sigma) = layer(C)$.

As a static method is not executed on an object, local variables (which include parameters) of a static method are required to be of a primitive, root, or any type. The same is true for the return type of a static method.

Extending the definition of the *ownmod* function (section 6.1) to account for static methods is straightforward: $ownmod(C.m(\ldots))$ yields the ownership modifier of the return type of m. No additional type-correctness rules are needed for lemmas 6.1 and 6.2.

Adding static methods does not affect lemma 8.1 (establishing ownership based invariants), or lemma 8.2 (establishing classical encapsulation). Adding static methods does require a re-evaluation of how to establish ownership encapsulation. In particular, the definitions of 'guarded for encapsulation' and 'statically meeting encapsulation' (see section 8.3) are extended to account for static methods. Section 8.3 defines when a method call to a non-static method in a non-static method is guarded for encapsulation, and when it statically meets encapsulation. Two extensions are required to account for static methods: the definitions have to account for (1) method calls to static methods from non-static methods, and (2) method calls from static methods. First consider case (1). A statement $v = C.m(\vec{s})$ leads to a method execution with a prestate σ such that $owner(\sigma) = \mathbf{root}$ and $layer(\sigma) = layer(C)$. Note that the same is true for a statement $v =$ new root C. Therefore, the same requirements can be imposed for both statements:

in a non-static method of class C, statement $v = D.m(\vec{s})$ is *guarded for encapsulation* if it is guarded by this.layer > layer(D) || this.owner == null && this.layer == layer(D). Furthermore, the statement *statically meets encapsulation* if $encap(C, \mathbf{root}, D)$ holds.

Accounting for case (2) is fairly straightforward, given that the execution of a method call in a static method of class C is done from a state in which $owner(\sigma) = \mathbf{root}$ and $layer(\sigma) = layer(C)$.

In a static method of class C, statement $v = r.m(\ldots)$ is *guarded for encapsulation* if it is guarded by r.owner = null && $layer(C) \geq$ r.layer. In a static method of class C, statement $v = r.m(\ldots)$ with *ownmod* D as static type of r *statically meets encapsulation* if $ownmod(r) =$ root and $layer(C) \geq layer(D)$. In a static method of class C, statement $v =$ new *ownmod* $D(\ldots)$ is *guarded for encapsulation* if *ownmod* is root and $layer(C) \geq layer(D)$. In a static method of class C, statement $v = D.m(\ldots)$ is *guarded for encapsulation* if $layer(C) \geq layer(D)$. In static methods, object creation statements and static method calls never statically meet encapsulation.[8]

Given these extensions, lemmas 8.3 and 8.4 hold in the presence of static methods. Establishing upward local consistency only poses a requirement for states in which control is with an object X, which is not the case if control is in a static method. However, calls to static methods have to be accounted for. In particular, the notions of 'guarded for consistency' and 'statically relevant' have to be extended. Again, such a call $v = D.m(\ldots)$ is treated in the same way as a call $v =$ new root D.

In a method of class C, statement $v = D.m(\ldots)$ is *guarded for consistency* if it is guarded by this.layer ! = layer(D) || this.owner != null || $upinv_C$. Furthermore, the statement is *statically relevant* if $layer(C) = layer(D)$.

Given these extensions, lemmas 8.5 and 8.6 hold in the presence of static methods. Lemma 8.7 holds in the presence of static methods as a static method cannot invalidate any ownership admissible invariants.

10 Related and Future Work

Non-modular verification techniques for the CII are suggested in [13, 27]. That the CII is not suitable for non-local invariants is also observed in [5] (in the context of OCL specifications [28]). It is proposed in [5] to make components explicit at the level of OCL designs. An invariant can be associated either with a component or with a class. A component invariant is interpreted to hold when control is outside the component. A class invariant is interpreted to hold in all visible states. The notion of a component seems closely related to the concept of ownership. In some sense, the objects inside a component are subordinates of that component. A problem might be that components cannot be used to easily capture class level subordinate relations. A more detailed analysis is considered future work.

Next, we consider programs with executions where the LRII does not capture the intention of the specifier and where refactoring is either impossible or undesirable. Two cases can be distinguished.

(A) The specifier intends that an invariant holds where the LRII does not require it (i.e., the LRII is too weak). In this case, if the execution represents a common scenario that can be identified, then a further refinement of the interpretation of invariants might be warranted. Otherwise, the invariant property can be made explicit in the precondition or postcondition where that property is expected to hold. The additional specification overhead is not really an issue as the scenario is uncommon. If the definition of an invariant is intended to be hidden, *predicate abstraction* techniques [23, 1, 15, 17, 20] can be used. These identify a predicate by an abstraction, which allow one to reason about the predicate without exposing its definition. These techniques are orthogonal to the discussion on the interpretation of invariants.

(B) An invariant is not intended to hold where the LRII guarantees it (i.e., the LRII is too strong). Due to observations 3.1 and 3.2, such an execution represents an atypical scenario. One could use specification constructs that make explicit that a certain invariant does *not* have to hold in a specific pre- or postcondition (note that this constitutes a refinement of the interpretation). One such construct (called inc) is provided in [20]. This construct allows the specifier of a method M to make explicit that M does not rely on a certain invariant. Combining the technique in [20] with the one in this paper is future work.

Note that [1, 17] use the Boogie methodology for invariants. In this methodology, invariants are used *only* for predicate abstraction: the specification of an invariant does not express that it holds in any given state. The Spec# verification tool [2] combines this methodology with a RII-like interpretation. Extensions of this methodology treat invariants that depend on static fields [18] or quantifications over objects [26], as well as multi-threaded programs [14]. Furthermore, [3] suggests a modular verification technique for invariants that are not ownership based. Other such techniques are discussed in [22, 20].

[8]Note that a simple syntactic check suffices to show that such statements are guarded for encapsulation

Finally, using a partially ordered set of layers might provide some additional flexibility at the cost of some additional specification and verification overhead. Adapting the definitions in section 4 is straightforward.

11 Conclusion

The topic of this paper is the formal specification and verification of invariants in OO designs. Several common scenarios are identified in which current interpretations do not allow for (easy) specification. The reason is that current interpretations do not account for the layering that is present in many OO designs. In particular, these interpretations do not exploit the subordinate relations at the class level that result from the layering. The paper shows how these layers can be made explicit with a minimum of specification overhead. Furthermore, it introduces the LRII, a refined interpretation of invariants that exploits these explicit layers. Together, this allows for easy specification of the identified problematic scenarios without adding much overhead to the specification of other scenarios. A sound, modular, and lightweight verification technique is introduced that ensures this refined interpretation is satisfied.

References

[1] Barnett, M., R. DeLine, M. Fähndrich, K. R. M. Leino and W. Schulte, *Verification of object-oriented programs with invariants*, Journal of Object Technology **3** (2004), pp. 27–56, special issue: ECOOP 2003 workshop on Formal Techniques for Java-like Programs.
URL `http://www.jot.fm/issues/issue_2004_06/article2`

[2] Barnett, M., K. R. M. Leino and W. Schulte, *The Spec# programming system: An overview*, in: G. Barthe, L. Burdy, M. Huisman, J.-L. Lanet and T. Muntean, editors, *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS '04)*, LNCS **3362** (2005).

[3] Barnett, M. and D. A. Naumann, *Friends need a bit more: Maintaining invariants over shared state*, in: D. Kozen and C. Shankland, editors, *Mathematics of Program Construction (MPC '04)*, LNCS **3125** (2004), pp. 54–84.

[4] Barnett, M., D. A. Naumann, W. Schulte and Q. Sun, *99.44% pure: Useful abstractions in specifications*, in: *The ECOOP'04 workshop on Formal techniques for Java-like programs (FTfJP)* (2004), pp. 11–18.
URL `http://www.cs.ru.nl/ftfjp/2004/Purity.pdf`

[5] Baumeister, H., R. Hennicker, A. Knapp and M. Wirsing, *Ocl component invariants*, in: *Monterey Workshop 2001, Engineering Automation for Software Intensive System Integration, Monterey, USA*, 2001, pp. 208–215.

[6] Clarke, D., "Object Ownership and Containment," Ph.D. thesis, University of New South Wales (2001).

[7] Darvas, Á. and K. R. M. Leino, *Practical reasoning about invocations and implementations of pure methods*, in: M. B. Dwyer and A. Lopes, editors, *Fundamental Approaches to Software Engineering (FASE '07)*, LNCS **4422** (2007), pp. 336–351.

[8] Dietl, W. and P. Müller, *Universes: Lightweight Ownership for JML*, Journal of Object Technology **4** (2005), pp. 5–32.
URL `http://www.jot.fm/issues/issue_2005_10/article1`

[9] Fitzgerald, J., I. J. Hayes and A. Tarlecki, editors, "FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, July 18-22, 2005, Proceedings," LNCS **3582**, Springer, 2005.

[10] Gamma, E., R. Helm, R. Johnson and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software," Addison-Wesley, 1995.

[11] Guttag, J. V. and J. J. Horning, "Larch: Languages and Tools for Formal Specification," Springer, New York, NY, USA, 1993.

[12] Hoare, C. A. R., *Proof of correctness of data representations*, Acta Informatica **1** (1972), pp. 271–281.

[13] Huizing, K. and R. Kuiper, *Verification of object oriented programs using class invariants*, in: T. S. E. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE '00)*, LNCS **1783** (2000), pp. 208–221.

[14] Jacobs, B., F. Piessens, K. R. M. Leino and W. Schulte, *Safe concurrency for aggregate objects with invariants*, in: B. K. Aichernig and B. Beckert, editors, *Software Engineering and Formal Methods (SEFM '05)* (2005), pp. 137–147.

[15] Kassios, I. T., *Dynamic frames: Support for framing, dependencies and sharing without restrictions*, in: J. Misra, T. Nipkow and E. Sekerinski, editors, *Formal Methods (FM '06)*, LNCS **4085** (2006), pp. 268–283.

[16] Leavens, G. T., E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Müller and J. Kiniry, "JML Reference Manual (draft)," 2007.
URL `http://www.jmlspecs.org`

[17] Leino, K. R. M. and P. Müller, *Object invariants in dynamic contexts*, in: M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP '04)*, LNCS **3086** (2004), pp. 491–516.

[18] Leino, K. R. M. and P. Müller, *Modular verification of static class invariants*, in: Fitzgerald et al. [9], pp. 26–42.

[19] Meyer, B., "Object-Oriented Software Construction, Second Edition," Prentice-Hall, New Jersey, 1997.

[20] Middelkoop, R., C. Huizing, R. Kuiper and E. J. Luit, *Invariants for non-hierarchical object structures*, in: L. Ribeiro and A. M. Moreira, editors, *Proceedings of the 9th Brazilian Symposium on Formal Methods (SBMF'06)*, Natal, Brazil, 2006.

[21] Müller, P., "Modular Specification and Verification of Object Oriented Programs," LNCS **2262**, Springer-Verlag, 2002.

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

[23] Parkinson, M., "Local Reasoning for Java," Ph.D. thesis, University of Cambridge (2005).

[24] Parnas, D. L., *On the criteria to be used in decomposing systems into modules*, Communications of the ACM **15** (1972), pp. 1053–1058.

[25] Pierik, C., "Validation Techniques for Object-Oriented Proof Outlines," Ph.D. thesis, Universiteit Utrecht (2006).

[26] Pierik, C., D. Clarke and F. S. de Boer, *Controlling object allocation using creation guards*, in: Fitzgerald et al. [9], pp. 59–74.

[27] Poetzsch-Heffter, A., "Specification and Verification of Object-Oriented Programs," Habilitationsschrift, Technische Universität München, 1997.

[28] Warmer, J. and A. Kleppe, "The Object Constraint Language: Precise modeling with UML," Addison-Wesley, 1999.

A Appendix

A.1 preliminaries

The proofs in this appendix rely on a number of additional definitions and lemmas.

First, three definitions are introduced. If Σ is a sequence, then $top(\Sigma) = \Sigma[0]$. Sequence concatenation is denoted by $\#$. A number of inductive proofs need a notion that is more involved than that of matching pre- and poststates. To this end, the notion of a *callstack* is introduced. Informally, $callstack(\Sigma, i)$ is the sequence of prestates $\Sigma[j]$, $j \leq i$, that are unmatched in $\Sigma[0..i]$, where the first element of $callstack(\Sigma, i)$ is the last unmatched element of $\Sigma[0..i]$ (i.e., the order is reversed). Formally, the definition is the following.

$$callstack(\Sigma, 0) = \Sigma[0]$$
$$callstack(\Sigma, i+1) = \begin{cases} \Sigma[i+1]\#callstack(\Sigma, i) & \text{if } \Sigma[i+1] \text{ is a prestate} \\ \Sigma' & \text{if } \Sigma[i+1] \text{ is a poststate, and} \\ & callstack(\Sigma, i) = \sigma\#\Sigma' \text{ for some prestate } \sigma \\ callstack(\Sigma, i) & \text{otherwise} \end{cases}$$

The semantics of Java-like programs allow for the following axiom (because after a method execution, control returns to the method execution that invoked it).

Axiom A.1 *If Σ is a program execution, and $callstack(\Sigma, i) = callstack(\Sigma, j)$, then if control is with an object X and in a class C in $\Sigma[i]$, then the same is true in $\Sigma[j]$.*

Next, a number of lemmas is formulated.

Lemma A.2 *If Σ is a program execution and $top(callstack(\Sigma, i)) = top(callstack(\Sigma, j))$, then $callstack(\Sigma, i) = callstack(\Sigma, j)$*

Lemma A.2 follows directly from the definition of callstack.

Lemma A.3 *If Σ is a program execution, and $\Sigma[i..j]$ is a method execution, then $top(callstack(\Sigma, j)) = \Sigma[i]$.*

Lemma A.4 *If Σ is a program execution, and $\Sigma[i..j]$ is a method execution, and $\Sigma[i..] \neq \Sigma[i..j]$, then $callstack(\Sigma, j + 1) = callstack(\Sigma, i - 1)$.*

Lemma A.5 *If Σ is a program execution, and $\Sigma[i..j]$ is a method execution, and $i \leq k \leq j$, and $top(callstack(\Sigma, k)) = l$, then $i \leq l < j$.*

Proof of these three lemmas is straightforward given te definitions of method execution and *callstack*(). For lemma A.4, note that $\Sigma[i..] \neq \Sigma[i..j]$ says that j is not the last element of Σ. This implies that $\Sigma[j + 1]$ exists and that $i > 0$.

Lemma A.6 *If Σ is a program execution that satisfies ownership encapsulation, and in $\Sigma[i]$, an object in layer l owns an object in layer l', then $l \geq l'$.*

Proof of lemma A.6 is by induction on i.
Base $i = 0$: In $\Sigma[0]$, no objects are allocated (language property) and the induction hypothesis holds trivially.
Step: Assume the induction hypothesis holds for $i = n$, and prove that it holds for $i = n+1$. Two cases can be distinguished.
Case 1: $\Sigma[n + 1]$ and $\Sigma[n]$ have the same set of allocated objects. Then the induction hypothesis holds for $i = n + 1$ (as the layer of an allocated object can not be changed).
Case 2: $\Sigma[n + 1]$ and $\Sigma[n]$ do not have the same set of allocated objects. Then the program counter of $\Sigma[n]$ is at an object creation statement v = new *ownmod* $C()$, and there is exactly one object X that is allocated in $\Sigma[n+1]$ but not in $\Sigma[n]$ (objects can only be allocated by an object creation statement, and object cannot be de-allocated). Two cases can be distinguished
Case 2.1: X is directly owned by **root**. Then there is no object that owns X. Then the induction hypothesis holds for $i = n + 1$ (as it holds for $i = n$).
Case 2.2: X is not directly owned by **root**. Then *ownmod* differs from root. Then control in $\Sigma[n]$ is with an object Y (only v = new root $C()$ occurs in a static method). Then the direct owner of X either is Y, or owns Y (as *ownmod* is peer or rep). Let object Z in layer l own X. Then either $Z = Y$, or Z owns Y. Then $l \geq layer(\Sigma[n])$ (as Y is in layer $layer(\Sigma[n])$, and as the induction property holds for $i = n$). Due to ownership encapsulation, $layer(\Sigma[n]) \geq layer(\Sigma[n + 1])$. Then $l \geq layer(\Sigma[n + 1])$. As X is in layer $layer(\Sigma[n + 1])$ (by definition), the induction hypothesis holds for $i = n + 1$.
In all cases, the induction hypothesis holds for $i = n + 1$, which concludes the proof.

Lemma A.7 *If Σ is a program execution, and object X owns object Y in $\Sigma[i]$, then X is allocated in $\Sigma[i]$.*

Proof by induction on i is straightforward. The outline is the following. In the base case (i=0), no objects are allocated and the hypothesis holds trivially. In the step case (i = n+1), the hypothesis holds trivially unless an object is newly allocated in $\Sigma[n+1]$. A newly allocated object X is either directly owned by **root** (no object owns X), or owned by the object with which control is in $\Sigma[i]$ (and control is always with an allocated object), or directly owned by $owner(\Sigma[n])$ (which is allocated as the induction property holds for $i = n$). All objects that own the direct owner of X are allocated as the induction property holds for $i = n$, which concludes the proof sketch.

Lemma A.8 *Assume σ is an execution state in a program execution that is ownership safe and layer safe. Assume σ maps reference r to object X of class D. Assume r has static type ownmod C and differs from* this*. Then either ownmod is* any*, or $layer(C) = layer(D)$. Furthermore, either ownmod is* root *and* **root** *directly owns X, or control is with an object Y in σ and $ownmod(Y, ownmod, X)$ holds.*

This lemma essentially extends the notions of layer and ownership safety (see section 6) to references. Note that $ownmod(Y, ownmod, X)$is defined in section 6.1. Proof by structural induction on r is straightforward.

A.2 towards theorem 7.1

Theorem 7.1 is proven in two steps. Here, it is proven that if an object X is consistent in a prestate $\Sigma[j]$, and $top(callstack(\Sigma, i)) = j$, then X is consistent in $\Sigma[i]$ unless control is with X or with an object owned by X. In appendix A.3, this lemma is used to prove the theorem.

Lemma A.9 *Let P be a program in which all invariants are ownership based. Let Σ be an execution of P that satisfies classical encapsulation, ownership encapsulation and upward and downward local consistency. Then for every i, j such that $top(callstack(\Sigma, i)) = j$, for every X that is allocated in $\Sigma[i]$,*
either X is consistent in $\Sigma[i]$,
or X is allocated but not consistent in any execution state in $\Sigma[j..i]$,
or control is with an object Y in $\Sigma[i]$ and either $X = Y$, or X owns Y.

Let $IH(n)$ be equal to lemma A.9, but with "For every i, j such that $top(callstack(\Sigma, i)) = j$" replaced by "For every i, j such that $i - j = n$ and $top(callstack(\Sigma, i)) = j$". From the definition of *callstack*, it follows immediately that $j \leq i$, i.e., that $n \geq 0$. We prove (by induction on n) that $IH(n)$ holds for every n.

Base ($n = 0$)**:** If $i - j = 0$, then $i = j$. Therefore, $\Sigma[j..i] = \Sigma[i]$. Therefore, $IH(0)$ is implied by the following proposition: For every X that is allocated in $\Sigma[i]$, either X is consistent in $\Sigma[i]$, or X not consistent in $\Sigma[i]$. This proposition is trivially true, which concludes the proof of this case.

Step: We assume $IH(m)$ holds for every m such that $m < n + 1$, and prove $IH(n + 1)$. Let Σ be an execution of a program P in which all invariants are ownership based. Let Σ satisfies classical encapsulation, ownership encapsulation and upward and downward local consistency. Let $i - j = n + 1$ and let $top(callstack(\Sigma, i)) = j$. Let X be an object that is allocated $\Sigma[i]$.

As $i - j = n + 1$, $i \neq j$, i.e., $i \neq top(callstack(\Sigma, i))$. Therefore, $\Sigma[i]$ is not a prestate (by definition of *callstack*). As $\Sigma[i]$ is not a prestate and X is allocated in $\Sigma[i]$, X is allocated in $\Sigma[i - 1]$.

Let $top(callstack(\Sigma, i - 1)) = k$. Two cases can be distinguished.

Case 1: $k = j$.

Case 2: $k \neq j$. Then $callstack(\Sigma[i - 1]) \neq callstack(\Sigma[i])$. Then, as $\Sigma[i]$ is not a prestate, $\Sigma[i - 1]$ is a poststate (by definition of *callstack*). Then $\Sigma[i - 1]$ matches $\Sigma[top(callstack(\Sigma, i - 1))]$ (proof by induction is straightforward), i.e., then $\Sigma[k..i - 1]$ is a method execution. Then it follows from lemma A.4 that $callstack(\Sigma, i) = callstack(\Sigma, k - 1)$. As $top(callstack(\Sigma, k - 1)) < k$, it follows that $k > j$.

So, either $k = j$, or $\Sigma[k..i - 1]$ is a method execution and $k > j$. In both cases, $k \geq j$. Then $(i - 1) - k \leq n + 1$. Then $IH((i - 1) - k)$ holds (by assumption). Then three cases can be distinguished.

Case 1: X is consistent in $\Sigma[i - 1]$. Then, as all invariants are ownership based, either X is consistent in $\Sigma[i]$, or $\Sigma[i - 1]$ and $\Sigma[i]$ differ on an object Y and $X = Y$ or X owns Y. In the latter case, as Σ satisfies classical encapsulation, control is with Y in $\Sigma[i]$. That concludes the proof of this case.

Case 2: X is allocated but not consistent in any execution state in $\Sigma[k..i-1]$. Two cases can be distinguished (see above).

Case 2.1: $k = j$. Then either X is consistent in $\Sigma[i]$, or X is allocated but not consistent in any execution state in $\Sigma[j..i]$. That concludes the proof of this case.

Case 2.2: $\Sigma[k..i-1]$ is a method execution and $k > j$. Two cases can be distinguished.

Case 2.2.1: X is not allocated in $\Sigma[k-1]$. Then X is newly allocated in $\Sigma[k]$. Then $\Sigma[k]$ is a prestate in which X is constructing, and in which control is in class $type(X)$ (language property). Then X is consistent for $[type(X), \mathsf{Object}]$ in poststate $\Sigma[i-1]$ (Σ satisfies upward local consistency). Then X is consistent in $\Sigma[i-1]$. As this contradicts that X is not consistent in any execution state in $\Sigma[j..i]$, this case is not feasible.

Case 2.2.2: X is allocated in $\Sigma[k-1]$. Note that $callstack(\Sigma, k-1) = callstack(\Sigma, i)$ (lemma A.4). Furthermore, $(k-1) - j \leq n+1$ (as $j > k < i$). Then the the following cases can be distinguished (as $IH((k-1)-j)$ holds by assumption).

Case 2.2.2.1: X is consistent in $\Sigma[k-1]$. Then X is consistent in $\Sigma[k-1]$, but not in $\Sigma[k]$. Then $\Sigma[k-1]$ and $\Sigma[k]$ differ on an object Y and either $X = Y$, or X owns Y (all invariants in P are ownership based). Then control is with Y in $\Sigma[k]$ (Σ satisfies classical encapsulation). Then Y is not allocated in $\Sigma[k-1]$ (due to a language property: a context switch does not change the object store. Therefore, Y must be newly created). Then $X \neq Y$ (X is allocated in $\Sigma[k-1]$). Then X owns Y. Let O be the direct owner of Y, i.e., $owner(\Sigma[k]) = O$. Then either $X = O$, or X owns O (as X owns Y). Then $O \neq \mathbf{root}$. Then either $owner(\Sigma[k-1]) = O$, or control is with O in $\Sigma[k-1]$ (Σ satisfies ownership encapsulation). Then control is with an object Z in $\Sigma[k-1]$ and either $X = Z$, or X owns Z. Then the same is true in $\Sigma[i]$ (axiom A.1). That concludes the proof of this case.

Case 2.2.2.2: X is allocated but not consistent in any execution state in $\Sigma[j..k-1]$. Then either X is consistent in $\Sigma[i]$, or X is allocated but not consistent in any execution state in $\Sigma[j..i]$. That concludes the proof of this case.

Case 2.2.2.3: Control is with an object Y in $\Sigma[k-1]$ and either $X = Y$, or X owns Y. Then the same is true in $\Sigma[i]$ (due to axiom A.1). That concludes the proof of this case.

Case 3: Control is with an object Y in $\Sigma[i-1]$ and either $X = Y$, or X owns Y. Two cases can be distinguished (see above).

Case 3.1: $k = j$. Then control is with Y in $\Sigma[i]$ (lemma A.2 and axiom A.1). That concludes the proof of this case.

Case 3.2: $\Sigma[k..i-1]$ is a method execution and $k > j$. Two cases can be distinguished.

Case 3.2.1: $X = Y$. Two cases can be distinguished.

Case 3.2.1.1: X is not constructing in $\Sigma[i-1]$. Then X is consistent in poststate $\Sigma[i-1]$ (Σ satisfies upward and downward local consistency). Then X is consistent in $\Sigma[i]$ (a context switch does not change the object store). That concludes the proof of this case.

Case 3.2.1.2: X is constructing in $\Sigma[i-1]$. Then X is constructing in $\Sigma[k]$. Then two cases can be distinguished (language property: a constructor on X can only be executed as a result of a superclass constructor call, or an object creation statement).

Case 3.2.1.2.1: X is constructing in $\Sigma[k-1]$. Then control is with X in $\Sigma[k-1]$ (by definition). Note that $callstack(\Sigma, k-1) = callstack(\Sigma, i)$ (lemma A.4). Then control is with X in $\Sigma[i]$ (axiom A.1). That concludes the proof of this case.

Case 3.2.1.2.2: X is not allocated in $\Sigma[k-1]$. Then control is in class $type(X)$ in $\Sigma[k]$ (language property). Then control is in $type(X)$ in poststate $\Sigma[i-1]$ (axiom A.1). Then X is consistent for $[type(X), \mathsf{Object}]$ in $\Sigma[i-1]$ (Σ satisfies upward local consistency). Then X is consistent in $\Sigma[i-1]$. Then X is consistent in $\Sigma[i]$ (as a context switch does not change the object store). That concludes the proof of this case.

Case 3.2.2: X owns Y. Recall that control is with Y in $\Sigma[i-1]$. Then control is with Y in $\Sigma[k]$ (axiom A.1). Let O be the direct owner of Y, i.e., $owner(\Sigma[k]) = O$. Then either $X = O$, or X owns O (as X owns Y). Then $O \neq \mathbf{root}$. Then either $owner(\Sigma[k-1]) = O$, or control is with O in $\Sigma[k-1]$ (Σ satisfies ownership encapsulation). Then control is with an object Z in $\Sigma[k-1]$ and either $X = Z$, or X owns Z. Then the same is true in $\Sigma[i]$ (axiom A.1). That concludes the proof of this case.

In every feasible case in the proof above, it is deduced that
either X is consistent in $\Sigma[i]$,
or X allocated but not consistent in any execution state in $\Sigma[j..i]$,
or control is with an object Y in $\Sigma[i]$ and either $X = Y$, or X owns Y.
Then $IH(n+1)$ holds. That concludes the proof of the step case of the induction proof.

A.3 proof of theorem 7.1

Proof of theorem 7.1 is by induction on the number of visible states in Σ:

$IH(n)$**:** Let P be a program in which all invariants are ownership based. Let Σ be an execution of P that satisfies classical encapsulation, ownership encapsulation and upward and downward local consistency. If i is such that $\Sigma[0..i]$ contains at most n visible states, then $\Sigma[0..i]$ satisfies the LRII and the LRII-c.

Base $(n = 0)$**:** If there are no visible states in $\Sigma[0..i]$, then $\Sigma[0..i]$ trivially satisfies the LRII and the LRII-c. That concludes the proof of the base case.

Step: We assume $IH(n)$, and prove $IH(n+1)$.
Let P be a program in which all invariants are ownership based. Let Σ be an execution of P that satisfies classical encapsulation, ownership encapsulation and upward and downward local consistency. Let i be such that (1) $\Sigma[i]$ is a visible state, and (2) the number of visible states in $\Sigma[0..i]$ is $n+1$.

Let X be an object that is allocated in $\Sigma[i]$. Let X be in layer l. Two cases can be distinguished.

Case 1: $\Sigma[i]$ is a poststate. Let $top(callstack(\Sigma, i)) = j$. Then lemma A.9 allows the following case distinction (informally, the intention is to show that X satisfies the requirements of the LRII and LRII-c in each (sub)case).

Case 1.1: X is consistent in $\Sigma[i]$.

Case 1.2: X is allocated but not consistent in any execution state in $\Sigma[j..i]$. As $\Sigma[j]$ is a prestate (by definition of *callstack*), and as $j < i$, it follows from $IH(n)$ that $\Sigma[0..j]$ satisfies the LRII and the LRII-c. Two cases can be distinguished.

Case 1.2.1: X is constructing in $\Sigma[j]$. Then X is constructing in $\Sigma[i]$ (lemma A.2 and axiom A.1). Let control be in class C in $\Sigma[i]$. As Σ satisfies upward local consistency, X is consistent for $[C, \mathsf{Object}]$ in $\Sigma[i]$.

Case 1.2.2: X is not constructing in $\Sigma[j]$. Recall that X is not consistent in $\Sigma[j]$. Then either $l > layer(\Sigma[j])$, or $owner(\Sigma[j])$ does not own X and $l = layer(\Sigma[j])$ (due to LRII). Due to lemma A.2 and axiom A.1, $layer(\Sigma[j]) = layer(\Sigma[i])$ and $owner(\Sigma[j]) = owner(\Sigma[i])$. Then either $l > layer(\Sigma[i])$, or $owner(\Sigma[i])$ does not own X and $l = layer(\Sigma[i])$.

Case 1.3: Control is with X in $\Sigma[i]$. Let control be in class C in $\Sigma[i]$. Then, as Σ satisfies upward and downward local consistency, X is consistent for $[C, \mathsf{Object}]$, and either X is constructing in $\Sigma[i]$ or X is consistent for $[type(X), C\rangle$. Then either X is consistent in $\Sigma[i]$, or X is constructing and consistent for $[C, \mathsf{Object}]$ in $\Sigma[i]$. Note that if X is constructing in $\Sigma[i]$, then X is constructing (and thus allocated) in $\Sigma[j]$ (lemma A.2 and axiom A.1).

Case 1.4: Control is with an object Y in $\Sigma[i]$ and X owns Y. Then $owner(\Sigma[i])$ does not own X and X is not constructing in $\Sigma[i]$. Furthermore, $l \geq layer(\Sigma[i])$ (lemma A.6). Also, due to lemma A.2 and axiom A.1, control is with Y in $\Sigma[j]$. Then X owns Y in $\Sigma[j]$ (language property). Then X is allocated in $\Sigma[j]$ (lemma A.7).

From the four cases above, it follows that if $\Sigma[i]$ is a poststate, then
either X is consistent in $\Sigma[i]$,
or X is allocated in $\Sigma[j]$ and
either in $\Sigma[i]$, X is constructing and control is in class C and X is consistent for $[C, \mathsf{Object}]$,
or in $\Sigma[i]$, X is non-constructing and
either $l > layer(\Sigma[i])$, or $owner(\Sigma[i])$ does not own X and $l = layer(\Sigma[i])$.

Case 2: $\Sigma[i]$ is a prestate. As X is allocated in $\Sigma[i]$, $i > 0$ (language property: no objects are allocated in $\Sigma[0]$). Two cases can be distinguished.

Case 2.1: X is not allocated in $\Sigma[i-1]$. Then X is constructing in $\Sigma[i]$ (language property).

Case 2.2: X is allocated in $\Sigma[i-1]$. Let $top(callstack(\Sigma, i-1)) = j$. Then lemma A.9 allows the following case distinction (informally, the intention is to show that X satisfies the requirements of the LRII and LRII-c in each (sub)case).

Case 2.2.1: X is consistent in $\Sigma[i-1]$. Two cases can be distinguished.

Case 2.2.1.1: X is consistent in $\Sigma[i]$.

Case 2.2.1.2: X is not consistent in $\Sigma[i]$. Then $\Sigma[i-1]$ and $\Sigma[i]$ differ on an object Y and either $X = Y$, or X owns Y (all invariants are ownership based). Then Y is not allocated in $\Sigma[i-1]$ (a context switch does not change the object store). Then $X \neq Y$ (X is allocated in $\Sigma[i-1]$). Then X owns Y in $\Sigma[i]$. Then $l \geq layer(\Sigma[i])$ (lemma A.6), and $owner(\Sigma[i])$ does not own X (by definition).

Case 2.2.2: X is allocated but not consistent in any execution state in $\Sigma[j..i-1]$, and control is not with X in $\Sigma[i-1]$. Then X is non-constructing in $\Sigma[i-1]$. Then X is non-constructing in $\Sigma[j]$ (lemma A.2 and axiom A.1). As $\Sigma[j]$ is a prestate (by definition of *callstack*), and as $j < i$, it follows from $IH(n)$ that $\Sigma[0..j]$ satisfies the LRII. Then either $l > layer(\Sigma[j])$, or $owner(\Sigma[j])$ does not own X and $l = layer(\Sigma[j])$ (X is non-constructing but not consistent in $\Sigma[j]$). Due to lemma A.2 and axiom A.1, $layer(\Sigma[j]) = layer(\Sigma[i])$ and $owner(\Sigma[j]) = owner(\Sigma[i])$. Then either $l > layer(\Sigma[i])$, or $owner(\Sigma[i])$ does not own X and $l = layer(\Sigma[i])$.

Case 2.2.3: In $\Sigma[i-1]$, X is not consistent and control is with X. Two cases can be distinguished.

Case 2.2.3.1: $\Sigma[i-1]$ is a horizontal call state. Then $\Sigma[i-1]$ is not relevant (X is not consistent in $\Sigma[i-1]$ and Σ satisfies upward and downward local consistency). Then X is constructing in $\Sigma[i]$ (by definition of 'relevant').

Case 2.2.3.2: $\Sigma[i-1]$ is not a horizontal call state. As Σ satisfies ownership encapsulation, three cases can be distinguished.

Case 2.2.3.2.1: $layer(\Sigma[i-1]) \geq layer(\Sigma[i])$ and $owner(\Sigma[i-1]) = owner(\Sigma[i])$. Then $layer(\Sigma[i-1]) > layer(\Sigma[i])$ ($\Sigma[i-1]$ is not a horizontal call state). Then $l > layer(\Sigma[i])$ (control is with X in $\Sigma[i-1]$).

Case 2.2.3.2.2: $layer(\Sigma[i-1]) \geq layer(\Sigma[i])$ and control is with $owner(\Sigma[i])$ in $\Sigma[i-1]$. Then $X = owner(\Sigma[i-1])$. Then $owner(\Sigma[i-1])$ does not own X (by definition).

Case 2.2.3.2.3: $layer(\Sigma[i-1]) > layer(\Sigma[i])$ and $owner(\Sigma[i]) = \mathbf{root}$. Then $l > layer(\Sigma[i])$.

Case 2.2.4: In $\Sigma[i-1]$, X is not consistent, and control is with an object Y, and X owns Y. Then $l \geq layer(\Sigma[i-1])$ (due to lemma A.6). As Σ satisfies ownership encapsulation, three cases can be distinguished.

Case 2.2.4.1: $layer(\Sigma[i-1]) = layer(\Sigma[i])$ and $owner(\Sigma[i-1]) = owner(\Sigma[i])$. Then $l \geq layer(\Sigma[i])$, and $owner(\Sigma[i])$ does not own X (by definition).

Case 2.2.4.2: $layer(\Sigma[i-1]) \geq layer(\Sigma[i])$ and control is with $owner(\Sigma[i])$ in $\Sigma[i-1]$. Then $l \geq layer(\Sigma[i])$, and $owner(\Sigma[i])$ does not own X (by definition).

Case 2.2.4.3: $layer(\Sigma[i-1]) > layer(\Sigma[i])$ and $owner(\Sigma[i]) = \mathbf{root}$. Then $l > layer(\Sigma[i]$.

From the two cases above, it follows that if $\Sigma[i]$ is a prestate, then
either X is consistent in $\Sigma[i]$,
or in $\Sigma[i]$, X is constructing,
or in $\Sigma[i]$, X is non-constructing and
either $l > layer(\Sigma[i])$, or $owner(\Sigma[i])$ does not own X and $l = layer(\Sigma[i])$,

From the two cases above, it follows that if X is non-constructing in $\Sigma[i]$, then
1) if either $l < layer(\sigma)$, or $owner(\sigma)$ owns X and $l = layer(\sigma)$, then X is consistent, and
2) if σ is a poststate and $owner(\sigma)$ owns X and $l > layer(\sigma)$, then X is at least as consistent in σ as in the prestate matching σ.

Therefore, $\Sigma[0..i]$ satisfies the LRII. It also follows that if $\Sigma[i]$ is a poststate, then
1) if X is constructing and control is in class C, then X is consistent for $[C, \mathsf{Object}]$, and
2) if X is not allocated in $top(callstack(\Sigma[i]))$, then X is consistent.

As $top(callstack(\Sigma[i]))$ is the prestate matching $\Sigma[i]$ (proof is straightforward), $\Sigma[0..i]$ satisfies the LRII-c. As $\Sigma[0..i]$ contains $n+1$ visible states, if j is such that $\Sigma[0..j]$ contains $n+1$ visible states, then $\Sigma[0..j]$ satisfies the LRII and the LRII-c. Then $IH(n+1)$ holds (as $IH(n)$ holds by assumption). That concludes the proof of the step case of the induction proof.

A.4 proof of lemma 8.1

This section sketches the proof of lemma 8.1.

Assume that invariant inv_C is ownership admissible, and that every execution of program P is ownership safe. Consider two consecutive execution states σ and σ' in execution Σ of P. Assume that object invariant $inv_C(X)$ holds in σ but not in σ'. Let $\bar{\sigma}$ and $\bar{\sigma}'$ be the execution states like σ and σ', but with this mapped to X. Then, by definition, inv_C holds in $\bar{\sigma}$, but not in $\bar{\sigma}'$. As inv_C is ownership admissible, it must be the case that inv_C contains a reference this.$f_1 \ldots . f_i$ $(i \geq 1)$ that is not mapped to the same value in both $\bar{\sigma}$ and $\bar{\sigma}'$. Then there is a j, $j < i$, such that (1) this.$f_1 \ldots f_j$ is mapped to an object Y by both $\bar{\sigma}$ and $\bar{\sigma}'$, and (2) $\bar{\sigma}$ and $\bar{\sigma}'$ differ on $Y.f_{j+1}$. Then σ and σ' differ on $Y.f_{j+1}$ as well. Two cases can be distinguished.

Case 1: $j = 0$. Then $Y = X$. Then σ and σ' differ on X.

Case 2: $j > 0$. Then, as inv_C is ownership admissible, $ownmod$(this.$f_1 \ldots f_{i-1}$) is either rep or owned. Then, by definition of $ownmod$, $ownmod$(this.$f_1 \ldots f_j$) is either rep or owned (see the table in section 6.1). Then, as Σ is ownership safe, X owns Y (this follows from a straightforward proof by structural induction on references).

In both cases, σ and σ' differ either on X, or on an object owned by X. That concludes the proof.

A.5 proof of lemma 8.3

This section contains a proof of the ownership encapsulation lemma, lemma 8.3. The lightweight variant of this lemma is proven in appendix A.6.

Assume program P meets PO8.1. Consider an arbitrary execution Σ of P. Consider two consecutive execution states $\Sigma[i]$ and $\Sigma[i+1]$ in Σ such that $\Sigma[i+1]$ is a prestate. The goal is to prove that

either $layer(\Sigma[i]) \geq layer(\Sigma[i+1])$ and $owner(\Sigma[i]) = owner(\Sigma[i+1])$,

or $layer(\Sigma[i]) \geq layer(\Sigma[i+1])$ and control is with $owner(\Sigma[i+1])$ in $\Sigma[i]$,

or $layer(\Sigma[i]) > layer(\Sigma[i+1])$ and $owner(\Sigma[i+1]) = \mathbf{root}$.

The proof is as follows. By definition, the program counter in $\Sigma[i]$ is at a method call statement $Stat$. Then, as P meets PO8.1, $Stat$ is guarded for encapsulation (see sections 8.3 and 9.2). Then, by definition, either (1) $Stat$ is a superclass constructor call, or (2) the program counter in $\Sigma[i-1]$ is at a statement assert $BoolE$. In case (2), $BoolE$ holds in $\Sigma[i-1]$ (as the execution of this assert statement did not cause the program execution to abort). Then $BoolE$ holds in $\Sigma[i]$ as well (an assert statement does not change the stack or object store). Two cases can be distinguished.

Case 1: Control is with an object X in $\Sigma[i]$. Then the program counter in $\Sigma[i]$ is not in a static method. Then, by definition, $owner(\Sigma[i]) = X.owner$ and $layer(\Sigma[i]) = X.layer$. Four cases can be distinguished (and in each case, it is deduced that the goal is met).

Case 1.1: $Stat$ is an instance method call $v = r.m(\ldots)$. Then $BoolE$ is

(r.owner == this || r.owner == this.owner) && this.layer >= r.layer || r.owner == null && this.layer > r.layer

($Stat$ is guarded for encapsulation, see section 8.3). Note that in $\Sigma[i+1]$, control is with the object referred to by r in $\Sigma[i]$ (language property). Three cases can be distinguished.

Case 1.1.1: r.owner == this && this.layer >= r.layer holds in $\Sigma[i]$. Then $owner(\Sigma[i+1]) = X$ and $X.layer \geq layer(\Sigma[i+1])$ (lemma 5.1). Then $layer(\Sigma[i]) \geq layer(\Sigma[i+1])$ and control is with $owner(\Sigma[i+1])$ in $\Sigma[i]$.

Case 1.1.2: r.owner == this.owner && this.layer >= r.layer holds in $\Sigma[i]$. Then $owner(\Sigma[i+1]) = owner(\Sigma[i])$ and $X.layer \geq layer(\Sigma[i+1])$ (lemma 5.1). Then $layer(\Sigma[i]) \geq layer(\Sigma[i+1])$ and $owner(\Sigma[i]) = owner(\Sigma[i+1])$.

Case 1.1.3: r.owner == null && this.layer > r.layer holds in $\Sigma[i]$. Then $owner(\Sigma[i+1]) = \mathbf{root}$ and $X.layer > layer(\Sigma[i+1])$ (lemma 5.1). Then $layer(\Sigma[i]) > layer(\Sigma[i+1])$ and $owner(\Sigma[i+1]) = \mathbf{root}$.

Case 1.2: $Stat$ is an object creation statement v = new $ownmod$ $C(\ldots)$. Assume, without loss of generality, that control is with an object Y in $\Sigma[i+1]$. Three cases can be distinguished.

Case 1.2.1: $ownmod$ is rep. Then, by definition (see section 5.1), $Y.owner = X$ and $Y.layer = layer(C)$. Then control is with $owner(\Sigma[i+1])$ in $\Sigma[i]$. Furthermore $BoolE$ is this.layer >= layer(C)

(Stat is guarded for encapsulation). Then $X.\mathit{layer} \geq Y.\mathit{layer}$. Then $\mathit{layer}(\Sigma[i] \geq \mathit{layer}(\Sigma[i+1]))$ and control is with $\mathit{owner}(\Sigma[i+1])$ in $\Sigma[i]$.

Case 1.2.2: *ownmod* is peer. Then, by definition (see section 5.1), $Y.\mathit{owner} = X.\mathit{owner}$ and $Y.\mathit{layer} = \mathit{layer}(C)$. Then $\mathit{owner}(\Sigma[i]) = \mathit{owner}(\Sigma[i+1])$. Furthermore BoolE is this.layer >= layer(C) (Stat is guarded for encapsulation). Then $X.\mathit{layer} \geq Y.\mathit{layer}$. Then $\mathit{layer}(\Sigma[i] \geq \mathit{layer}(\Sigma[i+1]))$ and $\mathit{owner}(\Sigma[i]) = \mathit{owner}(\Sigma[i+1])$.

Case 1.2.3: *ownmod* is root. Then, by definition (see section 5.1), $Y.\mathit{owner} = \mathbf{root}$ and $Y.\mathit{layer} = \mathit{layer}(C)$. Furthermore BoolE is this.layer > layer(C) || this.owner == null && this.layer == layer(C) (Stat is guarded for encapsulation). Then either $X.\mathit{layer} > Y.\mathit{layer}$, or $X.\mathit{owner} = \mathbf{root}$ and $X.\mathit{layer} = Y.\mathit{layer}$ (lemma 5.1). Then either $\mathit{layer}(\Sigma[i]) > \mathit{layer}(\Sigma[i+1])$ and $\mathit{owner}(\Sigma[i+1]) = \mathbf{root}$, or $\mathit{layer}(\Sigma[i] \geq \mathit{layer}(\Sigma[i+1]))$ and $\mathit{owner}(\Sigma[i]) = \mathit{owner}(\Sigma[i+1])$.

Case 1.3: Stat is an superclass constructor call $v = C(\ldots)$. Then control is with X in $\Sigma[i+1]$ (language property). Then $\mathit{layer}(\Sigma[i]) \geq \mathit{layer}(\Sigma[i+1])$ and $\mathit{owner}(\Sigma[i]) = \mathit{owner}(\Sigma[i+1])$.

Case 1.4: Stat is an static method call $v = C.m(\ldots)$. Then, by definition, $\mathit{owner}(\Sigma[i+1]) = \mathbf{root}$ and $\mathit{layer}(\Sigma[i+1]) = \mathit{layer}(C)$. Furthermore, BoolE is this.layer > layer(C) || this.owner == null && this.layer == layer(C) (Stat is guarded for encapsulation). Then either $X.\mathit{layer} > \mathit{layer}(C)$, or $X.\mathit{owner} = \mathbf{root}$ and $X.\mathit{layer} = \mathit{layer}(C)$ (lemma 5.1). Then either $\mathit{layer}(\Sigma[i]) > \mathit{layer}(\Sigma[i+1])$ and $\mathit{owner}(\Sigma[i+1]) = \mathbf{root}$, or $\mathit{layer}(\Sigma[i] \geq \mathit{layer}(\Sigma[i+1]))$ and $\mathit{owner}(\Sigma[i]) = \mathit{owner}(\Sigma[i+1])$.

Case 2: Control is not with an object in $\Sigma[i]$. Then the program counter in $\Sigma[i]$ is in a static method. Let control in class C be in $\Sigma[i]$. Note that superclass constructor calls do not occur in static methods. Therefore, three cases can be distinguished (and in each case, it is deduced that ownership encapsulation is not violated).

Case 2.1: Stat is an instance method call $v = r.m(\ldots)$. Then BoolE is (r.owner == null && layer(C) >= r.layer (Stat is guarded for encapsulation). Note that in $\Sigma[i+1]$, control is with the object referred to by r in $\Sigma[i]$ (language property). Then $\mathit{layer}(\Sigma[i]) \geq \mathit{layer}(\Sigma[i+1])$ and $\mathit{owner}(\Sigma[i]) = \mathit{owner}(\Sigma[i+1])$ (lemma 5.1).

Case 2.2: Stat is an object creation statement $v =$ new *ownmod* $D(\ldots)$. Then *ownmod* is root and $\mathit{layer}(C) \geq \mathit{layer}(D)$ (Stat is guarded for encapsulation). Let control be with an object Y in $\Sigma[i+1]$. Then $Y.\mathit{owner} = \mathbf{root}$ and $Y.\mathit{layer} = \mathit{layer}(C)$ (lemma 5.1). Then $\mathit{layer}(\Sigma[i]) \geq \mathit{layer}(\Sigma[i+1])$ and $\mathit{owner}(\Sigma[i]) = \mathit{owner}(\Sigma[i+1])$.

Case 2.3: Stat is an static method call $v = D.m(\ldots)$. Then, by definition, $\mathit{owner}(\Sigma[i+1]) = \mathbf{root}$ and $\mathit{layer}(\Sigma[i+1]) = \mathit{layer}(D)$. Furthermore, $\mathit{layer}(C) \geq \mathit{layer}(D)$ (Stat is guarded for encapsulation). Then $\mathit{layer}(\Sigma[i]) \geq \mathit{layer}(\Sigma[i+1])$ and $\mathit{owner}(\Sigma[i]) = \mathit{owner}(\Sigma[i+1])$.

That concludes the proof (as the goal is met in each (sub)case).

A.6 proof of lemma 8.4

This section contains a proof of lightweight ownership encapsulation lemma, lemma 8.4.

Assume program P meets PO8.2. Consider an execution Σ of P that is ownership safe and layer safe. Consider two consecutive execution states $\Sigma[i]$ and $\Sigma[i+1]$ in Σ such that $\Sigma[i+1]$ is a prestate. The goal is to prove that

either $\mathit{layer}(\Sigma[i]) \geq \mathit{layer}(\Sigma[i+1])$ and $\mathit{owner}(\Sigma[i]) = \mathit{owner}(\Sigma[i+1])$,
or $\mathit{layer}(\Sigma[i]) \geq \mathit{layer}(\Sigma[i+1])$ and control is with $\mathit{owner}(\Sigma[i+1])$ in $\Sigma[i]$,
or $\mathit{layer}(\Sigma[i]) > \mathit{layer}(\Sigma[i+1])$ and $\mathit{owner}(\Sigma[i+1]) = \mathbf{root}$.

The proof is as follows. The program counter in $\Sigma[i]$ is at a method call statement Stat ($\Sigma[i+1]$ is a prestate). Then Stat either statically meets encapsulation, or is guarded for encapsulation (P meets PO8.2, see sections 8.3 and 9.2). It was proven in appendix A.5 that the goal is met if Stat is guarded for encapsulation. Here, it is proven that the goal is met under the assumption that Stat statically meets encapsulation. Let control be in class C in $\Sigma[i]$. Two cases can be distinguished.

Case 1: Control is with an object X in $\Sigma[i]$. Then the program counter in $\Sigma[i]$ is not in a static method. Then, by definition, $\mathit{owner}(\Sigma[i]) = X$.owner and $\mathit{layer}(\Sigma[i]) = X$.layer. Then, as Σ is layer safe, $\mathit{layer}(\Sigma[i]) \geq \mathit{layer}(C)$ (i.e., the layer of X is at least $\mathit{layer}(C)$). Four cases can be distinguished (and in each case, it is deduced that the goal is met).

Case 1.1: *Stat* is a superclass constructor call $v = B(\ldots)$. Then control in $\Sigma[i+1]$ is still with X (language property). Then, trivially, $layer(\Sigma[i+1]) = layer(\Sigma[i]$ and $owner(\Sigma[i+1] = owner(\Sigma[i])$.

Case 1.2: *Stat* is an instance method call $v = r.m(\ldots)$, and the static type of reference r is *ownmod* D. Let r refer to an object Y of a class E in $\Sigma[i]$. Then control is with Y in $\Sigma[i+1]$ (language property), and Y is in layer $layer(E)$ (by definition). Then $layer(\Sigma[i+1]) = layer(E)$ (by definition). Two cases can be distinguished.

Case 1.2.1: r is this. Then $X = Y$. Then, trivially, $layer(\Sigma[i]) \geq layer(\Sigma[i+1])$ and $owner(\Sigma[i]) = owner(\Sigma[i+1])$.

Case 1.2.2: r differs from this. Then $encap(C, ownmod, D)$ holds (*Stat* statically meets encapsulation, section 8.3). Then, due to lemma A.8, $layer(D) = layer(E)$, and $ownmod(X, ownmod, Y)$ holds (control is with X in $\Sigma[i]$ and *encap* ensures that *ownmod* differs from any). Then $layer(\Sigma[i+1]) = layer(D)$. Three cases can be distinguished (due to the definition of *encap*).

Case 1.2.2.1: $layer(C) \geq layer(D)$ and *ownmod* is peer. Then $layer(\Sigma[i]) \geq layer(\Sigma[i+1])$ (as $layer(\Sigma[i]) \geq layer(C)$). Furthermore the direct owner of X directly owns Y (as $ownmod(Y, \text{peer}, X)$ holds). Then, by definition, $owner(\Sigma[i]) = owner(\Sigma[i+1])$.

Case 1.2.2.2: $layer(C) \geq layer(D)$ and *ownmod* is rep. Then $layer(\Sigma[i]) \geq layer(\Sigma[i+1])$ (as $layer(\Sigma[i]) \geq layer(C)$). Furthermore X directly owns Y (as $ownmod(Y, \text{rep}, X)$ holds). Then, by definition, control is with $owner(\Sigma[i+1])$ in $\Sigma[i]$.

Case 1.2.2.3: $layer(C) > layer(D)$ and *ownmod* is root. Then $layer(\Sigma[i]) > layer(\Sigma[i+1])$ (as $layer(\Sigma[i]) \geq layer(C)$). Furthermore **root** directly owns Y (as $ownmod(Y, \text{root}, X)$ holds). Then, by definition, $owner(\Sigma[i+1])$ is **root**.

Case 1.3: *Stat* is an object creation statement $v =$ new *ownmod* $D(\ldots)$. Then $layer(\Sigma[i+1]) = layer(D)$ (by definition). Furthermore, $encap(C, ownmod, D)$ holds (*Stat* statically meets encapsulation, section 8.3). Then three cases can be distinguished.

Case 1.3.1: $layer(C) \geq layer(D)$ and *ownmod* is peer. Then $layer(\Sigma[i]) \geq layer(\Sigma[i+1])$ ($layer(\Sigma[i]) \geq layer(C)$). Furthermore, by definition (see section 5.1), the direct owner of X directly owns Y. Then $owner(\Sigma[i] = owner(\Sigma[i+1]))$.

Case 1.3.2: $layer(C) \geq layer(D)$ and *ownmod* is peer. Then $layer(\Sigma[i]) \geq layer(\Sigma[i+1])$ (as $layer(\Sigma[i]) \geq layer(C)$). Furthermore, by definition (see section 5.1), the direct owner of X directly owns Y. Then $owner(\Sigma[i] = owner(\Sigma[i+1]))$.

Case 1.3.3: $layer(C) > layer(D)$ and *ownmod* is root. Then $layer(\Sigma[i]) > layer(\Sigma[i+1])$ (as $layer(\Sigma[i]) \geq layer(C)$). Furthermore, by definition (see section 5.1), **root** directly owns Y. Then, by definition, $owner(\Sigma[i+1])$ is **root**.

Case 1.4: *Stat* is an static method call $v = D.m(\ldots)$. Then, by definition (see section 9.2), $layer(\Sigma[i+1]) = layer(D)$ and $owner(\Sigma[i+1] = \textbf{root})$. As *Stat* statically meets encapsulation (see section 9.2), $encap(C, \text{root}, D)$ holds. Then, by definition (see section 8.3), $layer(C) > layer(D)$. Then $layer(\Sigma[i]) > layer(\Sigma[i+1])$ (as $layer(\Sigma[i]) \geq layer(C)$).

Case 2: Control is not with an object in $\Sigma[i]$. Then the program counter in $\Sigma[i]$ is in a static method. Then, by definition (see section 9.2), $layer(\Sigma[i]) = layer(C)$ and $owner(\Sigma[i]) = \textbf{root}$. In a static method, the notion of 'statically meeting encapsulation' is only defined for instance method calls. Therefore, it can be assumed without loss of generality that *Stat* is an instance method call $v = r.m(\ldots)$ such that the static type of reference r is *root* D, and that $layer(C) \geq layer(D)$. Assume r refers to an object X of class E in $\Sigma[i]$. Then control is with X in $\Sigma[i+1]$ (language property), and $layer(D) = layer(E)$ and **root** directly owns X (due to lemma A.8 and language property: this does not occur in static methods). Then, by definition, $owner(\Sigma[i+1]) = \textbf{root}$. Also, as $layer(D) = layer(E)$, $layer(\Sigma[i+1]) = layer(D)$. Then $layer(\Sigma[i]) \geq layer(\Sigma[i+1])$ and $owner(\Sigma[i]) = owner(\Sigma[i+1])$.

That concludes the proof (as the goal is met in each (sub)case).

A.7 proof of lemma 8.5

This section contains a proof of the upwards local consistency lemma, lemma 8.5. The lightweight variant of this lemma is proven in appendix A.8.

Assume program P meets PO8.3 and PO8.4. Assume Σ is an arbitrary execution of P. Assume control is with object X and in class C in execution state $\Sigma[i]$. Assume $\Sigma[i]$ is either a relevant horizontal call state or a poststate. Then the goal is to prove that X is consistent for $[C, \mathsf{Object}]$ in $\Sigma[i]$. Let $upinv_C$ be the conjunction of the invariants of class C and C's superclasses. Two cases can be distinguished.

Case 1: $\Sigma[i]$ is a poststate. Then the program counter in $\Sigma[i]$ is at the end of a method (by definition). As P meets PO8.3, the program counter in $\Sigma[i]$ is at a statement $\mathsf{assert}\ upinv_C$. As the execution of this assert statement did not cause the program execution to abort, $upinv_C$ holds in $\Sigma[i-1]$. As an assert statement does not change the stack or object store, $upinv_C$ holds in $\Sigma[i]$ as well. Then, by definition, X is consistent for $[C, \mathsf{Object}]$ in $\Sigma[i]$ (as control is with X in $\Sigma[i]$).

Case 2: $\Sigma[i]$ is a relevant horizontal call state. Then $\Sigma[i+1]$ is a prestate and $owner(\Sigma[i]) = owner(\Sigma[i+1])$ and $layer(\Sigma[i]) = layer(\Sigma[i+1])$ (due to the definition of 'horizontal call state', see section 7). Then, by definition, the program counter in $\Sigma[i]$ is at a method call statement $Stat$. Furthermore, X is not constructing in $\Sigma[i+1]$ (due to the definition of 'relevant', see section 7). Then $Stat$ is not a superclass constructor call. Therefore, three cases can be distinguished.

Case 2.1: $Stat$ is an instance method call $v = r.m(\ldots)$. As P meets PO8.4, $Stat$ is guarded for consistency. Then the program counter in $\Sigma[i-1]$ is at a statement
$\mathsf{assert\ this.owner} \mathrel{!=} r.\mathsf{owner} \parallel \mathsf{this.layer} \mathrel{!=} r.\mathsf{layer} \parallel upinv_C$ (see section 8.4). Let r refer to an object Y in $\Sigma[i]$. Then control is with Y in $\Sigma[i+1]$ (language property). Then the direct owner of X is the direct owner of Y, and X and Y are in the same layer (as $\Sigma[i]$ is a horizontal call state, see above). Then $\mathsf{this.owner} \mathrel{!=} r.\mathsf{owner} \parallel \mathsf{this.layer} \mathrel{!=} r.\mathsf{layer}$ does not hold in $\Sigma[i]$ (lemmas 5.1 and A.8). Then the same is true in $\Sigma[i+1]$ (an assert statement does not change the stack or object store). Then $upinv_C$ holds in $\Sigma[i-1]$ (the execution of this assert statement did not cause the program execution to abort). Then $upinv_C$ also holds in $\Sigma[i]$. Then X is consistent for $[C, \mathsf{Object}]$ in $\Sigma[i]$ (by definition, as control is with X in $\Sigma[i]$).

Case 2.2: $Stat$ is an object creation statement $v = \mathsf{new}\ ownmod\ D(\ldots)$. Let control be with an object Y in $\Sigma[i+1]$. Then Y is in layer $layer(D)$ (by definition, see section 4). As $\Sigma[i]$ is a horizontal call state, X and Y are in the same layer. Then X is in layer $layer(D)$. Then $\mathsf{this.layer} = \mathsf{layer}(D)$ holds in $\Sigma[i]$ (lemma 5.1). As P meets PO8.4, $Stat$ is guarded for consistency, and three cases can be distinguished (see section 8.4).

Case 2.2.1: $ownmod$ is rep. Then X directly owns Y (by definition, see section 3.2). Then $owner(\Sigma[i+1]) = X$. Then $owner(\Sigma[i]) \neq owner(\Sigma[i+1])$. Then $\Sigma[i]$ is not a horizontal call state, which contradicts the above. This case is not feasible.

Case 2.2.2: $ownmod$ is peer and the program counter in $\Sigma[i-1]$ is at a statement
$\mathsf{assert\ this.layer} \mathrel{!=} \mathsf{layer}(D) \parallel upinv_C$. Then $\mathsf{this.layer} = \mathsf{layer}(D)$, which holds in $\Sigma[i]$, also holds in $\Sigma[i-1]$ (as an assert statement does not change the stack or object store). Then $upinv_C$ holds in $\Sigma[i-1]$ (the the execution of the assert statement from $\Sigma[i-1]$ did not cause the program execution to abort). Then $upinv_C$ also holds in $\Sigma[i]$. Then X is consistent for $[C, \mathsf{Object}]$ in $\Sigma[i]$ (by definition, as control is with X in $\Sigma[i]$).

Case 2.2.3: $ownmod$ is root and the program counter in $\Sigma[i-1]$ is at a statement
$\mathsf{assert\ this.layer} \mathrel{!=} \mathsf{layer}(D) \parallel \mathsf{this.owner} \mathrel{!=} \mathsf{null} \parallel upinv_C$. Then $\mathsf{this.layer} = \mathsf{layer}(D)$, which holds in $\Sigma[i]$, also holds in $\Sigma[i-1]$ (as an assert statement does not change the stack or object store). Furthermore, **root** directly owns Y (by definition, see section 3.2). As $\Sigma[i]$ is a horizontal call state, X and Y have the same owner. Then **root** directly owns X. Then $\mathsf{this.layer} = \mathsf{null}$ holds in $\Sigma[i]$ (lemma 5.1). Then $\mathsf{this.layer} = \mathsf{null}$ also holds in $\Sigma[i-1]$. Then $upinv_C$ holds in $\Sigma[i-1]$ (the the execution of the assert statement from $\Sigma[i-1]$ did not cause the program execution to abort). Then $upinv_C$ also holds in $\Sigma[i]$. Then X is consistent for $[C, \mathsf{Object}]$ in $\Sigma[i]$ (by definition, as control is with X in $\Sigma[i]$).

Case 2.3: $Stat$ is an static method call $v = D.m(\ldots)$. Then $owner(\Sigma[i+1]) =$ **root** and $layer(\Sigma[i+1]) = layer(D)$ (by definition). Then, **root** directly owns X, and X is in layer $layer(D)$ ($\Sigma[i]$ is a horizontal call state). Then $\mathsf{this.layer} = \mathsf{layer}(D)\ \&\&\ \mathsf{this.owner} = \mathsf{root}$ holds in $\Sigma[i]$ (due to lemma 5.1). As P meets PO8.4, $Stat$ is guarded for consistency. Then the program counter in $\Sigma[i-1]$ is at a statement $\mathsf{assert\ this.layer} \mathrel{!=} \mathsf{layer}(D) \parallel \mathsf{this.owner} \mathrel{!=} \mathsf{null} \parallel upinv_C$ (see the definition in 9.2). As an assert statement does not change the heap or object store, $\mathsf{this.layer} = \mathsf{layer}(D)\ \&\&\ \mathsf{this.owner} = \mathsf{root}$ holds in $\Sigma[i-1]$ as well. Then $upinv_C$ holds in $\Sigma[i-1]$ (the the execution of the assert statement from $\Sigma[i-1]$ did not cause the program execution to abort). Then $upinv_C$ also holds in $\Sigma[i]$. Then X is consistent for $[C, \mathsf{Object}]$ in $\Sigma[i]$ (by definition, as control is with X in $\Sigma[i]$).

In all feasible (sub)cases, the goal is met, which concludes the proof.

A.8 proof of lemma 8.6

This section contains a proof of the lightweight upwards local consistency lemma, lemma 8.5.

Assume program P meets PO8.3 and PO8.5. Assume Σ is an execution of P that is ownership safe and layer safe. Assume control is with object X and in class C in execution state $\Sigma[i]$. Assume $\Sigma[i]$ is either a relevant horizontal call state or a poststate. Then the goal is to prove that X is consistent for $[C, \mathsf{Object}]$ in $\Sigma[i]$. It was proven in appendix A.5 that the goal is met if $\Sigma[i]$ is a poststate. Here, it is proven that the goal is met under the assumption that $\Sigma[i]$ is a relevant horizontal call state. Then $\Sigma[i+1]$ is a prestate and $owner(\Sigma[i]) = owner(\Sigma[i+1])$ and $layer(\Sigma[i]) = layer(\Sigma[i+1])$ (due to the definition of 'horizontal call state', see section 7). Then the program counter in $\Sigma[i]$ is at a method call statement *Stat* (by definition of 'prestate'). As P meets PO8.5, *Stat* is either not statically relevant, or guarded for consistency. It was proven in appendix A.5 that the goal is met if *Stat* is guarded for consistency. Here, it is proven that the goal is met under the assumption that *Stat* is not statically relevant. X is not constructing in $\Sigma[i+1]$ ($\Sigma[i]$ is 'relevant', see section 7). Then *Stat* is not a superclass constructor call. Therefore, three cases can be distinguished. It is proven that none of these cases is feasible, i.e., that the execution of a method call that is not statically relevant, does not lead to a horizontal call state.

Case 1: *Stat* is an instance method call $v = r.m(\ldots)$, and the static type of r is $ownmod\ D$. Let r refer to an object Y in $\Sigma[i]$. Then control is with Y in $\Sigma[i+1]$ (language property). As *Stat* is not statically relevant, $statrel(C, ownmod, D)$ does not hold (see the definition in section 8.4). Then two cases can be distinguished.

Case 1.1: either $ownmod$ is rep, or $ownmod$ is owned. Then r differs from this (as $ownmod(\mathsf{this}) = \mathsf{peer}$). Then X owns Y (lemma A.8, as Σ is layer safe and ownership safe). Then X and Y do not have the same direct owner. Then $owner(\Sigma[i]) \neq owner(\Sigma[i+1])$. Then $\Sigma[i]$ is not a horizontal call state, which contradicts the above. This case is not feasible.

Case 1.2: $layer(C) \neq layer(D)$, and $ownmod$ is either peer, or root. Then r differs from this (as the static type of this is $\mathsf{peer}\ C$). Then Y is in layer $layer(D)$ (lemma A.8, as Σ is layer safe and ownership safe). As Σ is ownership safe, X is not in a layer that below $layer(C)$. Then $layer(\Sigma[i]) \neq layer(\Sigma[i+1])$. Then $\Sigma[i]$ is not a horizontal call state, which contradicts the above. This case is not feasible.

Case 2: *Stat* is an object creation statement $v = \mathsf{new}\ ownmod\ D(\ldots)$. Let control be with Y in $\Sigma[i+1]$. As *Stat* is not statically relevant, $statrel(C, ownmod, D)$ does not hold (see the definition in section 8.4). Then two cases can be distinguished.

Case 2.1: $ownmod$ is rep. Then X directly owns Y (see section 3.2). Then X and Y do not have the same direct owner. Then $owner(\Sigma[i]) \neq owner(\Sigma[i+1])$. Then $\Sigma[i]$ is not a horizontal call state, which contradicts the above. This case is not feasible.

Case 2.2: $layer(C) \neq layer(D)$, and $ownmod$ is either peer, or root. Note that Y is in layer $layer(D)$ (see section 4). As Σ is ownership safe, X is not in a layer that is below $layer(C)$. Then $layer(\Sigma[i]) \neq layer(\Sigma[i+1])$. Then $\Sigma[i]$ is not a horizontal call state, which contradicts the above. This case is not feasible.

Case 3: *Stat* is an static method call $v = D.m(\ldots)$. Then $layer(\Sigma[i+1]) = layer(D)$ (by definition, see section 9.2). As *Stat* is not statically relevant, $layer(C) \neq layer(D)$. As Σ is ownership safe, X is not in a layer that is below $layer(C)$. Then $layer(\Sigma[i]) \neq layer(\Sigma[i+1])$. Then $\Sigma[i]$ is not a horizontal call state, which contradicts the above. This case is not feasible.

That concludes the proof.

A.9 towards lemma 8.7: frames

As sketched in section 8.5, proof of the downwards consistency lemma (lemma 8.7) is done in two steps, and depends on the notion of class frames (or frames for short).

frames: A *frame* is a tuple of an owner and a class. In any given state, every allocated object is *directly framed* by exactly one frame. This relation is acyclic. Let frame (X, C) directly frame object Y. Then frame (Z, D) *frames* Y if either $(Z, D) = (X, C)$, or (Z, D) frames X. Finally, $frame(\sigma) = (O, C)$ if control is with an object that is directly owned by owner O and in a class C in execution state σ.

Frames imposes a partial ordering on the object structure as well as on states. In our language (see section 2), the direct frame of an object is determined by the *ownership modifier ownmod* of an object creation statement $v =$ new *ownmod* $C(\dots)$ (and can not be changed afterwards). Consider a program execution Σ. Assume that the program counter in $\Sigma[i]$ is at a statement $v =$ new *ownmod* $C(\dots)$. Assume that control is with object X with direct frame (O, D) in $\Sigma[i]$, and assume that control is in class E in $\Sigma[i]$. Then there is an object Y that is constructing in $\Sigma[i+1]$. The direct frame of Y is (X, D) when *ownmod* is rep, (O, D) when *ownmod* is peer, and (**root**, Object) when *ownmod* is root. Note that the default modifier is peer and can be omitted.

The above allows the following lemma. Proof is straightforward given how both the direct frame and direct owner of an object X are set on creation of X.

Lemma A.10 *If X is owned by Y, then there is exactly one class C such that (Y, C) frames X.*

In this section, a number of properties are formulated (and proven to hold) that match those in the proof outline in section 8.5. These properties use the following notion of reachability.

$X \in reach(Y, \sigma)$ if either $X = Y$, or there is field f of a rep or peer type such that $Y.f$ is mapped to object Y by σ's object store, and $X \in reach(Y, \sigma)$.

For the purpose the definition of *reach*, if control is in class C and with an object X, then stack variables are treated as fields of X defined in class C. Next, the lemma proven in section is formulated.

Lemma A.11 *If P1 and object X is framed by (Y, C) in $\Sigma[i]$, then $C1_i$ and $C2_i$ and $C3_i$ and $C4_i$.*

The shorthands used in lemma A.11 are defined below.

$P1$:	Program P meets the type-correctness rules of the eUTS, and every class of P meets SR8.1 and SR8.4-SR8.8, and Σ satisfies ownership encapsulation.
$C1_i$:	If control is with X in $\Sigma[i]$, then there is a prefix Σ' of $callstack(\Sigma[i])$ such that the last element of Σ' is a prestate in which control is with Y and in a class $D \subseteq C$, and every other element of Σ' is a prestate in which control is with an object owned by Y.
$C2_i$:	If object $Z \in reach(X, \Sigma[i])$, then Z is framed by (Y, C) in $\Sigma[i]$.
$C3_i$:	If location $Y.f$ is mapped to object X in $\Sigma[i]$, and f is of a rep type, then f is defined in class C.
$C4_i$:	If control is with Y in $\Sigma[i]$, and $\Sigma[i]$'s stack maps stack variable v to X, and v is of a rep type, then control is in class C.

Note that what $C3_i$ expresses for fields, $C4_i$ expresses for stack variables. The relation between the properties above and those in the proof outline of downwards local consistency in section 8.5 is as follows. $C1_i$ is a more elaborate version of frame encapsulation, $C2_i$ is a more elaborate version of intermediate property P2, and $C3_i$ and $C4_i$ combine into intermediate property P1.

Proof is by generalized induction on i. That is, we prove that given $P1$, for every i, $IH(i)$ holds, where

$IH(i)$: if X is framed by (Y, C) in $\Sigma[i]$, then $C1_i$ and $C2_i$ and $C3_i$ and $C4_i$.

Base ($i = 0$): Assume $P1$. Then $IH(0)$ holds trivially as no objects are allocated in $\Sigma[0]$ (language property). More specifically, there is no object X that is is framed.

Step: Assume $P1$. Then Σ is layer safe and ownership (lemmas 6.1 and 6.2). Assume that $IH(j)$ holds for every j, $j \leq i$. Then the goal is to prove $IH(i+1)$. Assume that X is framed by (Y, C) in $\Sigma[i]$. Three cases can be distinguished.

Case 1: The program counter in $\Sigma[i]$ is at a non-methodcall statement *Stat*. Then X is framed by (Y, C) in $\Sigma[i]$ (X was already allocated in $\Sigma[i]$ as *Stat* is not a object creation statement). Then $C1_i$, $C2_i$, $C3_i$ and $C4_i$ hold ($IH(i)$ holds). Furthermore, control is with the same object in $\Sigma[i]$ and $\Sigma[i+1]$ (language property) and $callstack(\Sigma[i]) = callstack(\Sigma[i+1])$ (by definition). Then $C1_{i+1}$ holds ($C1_i$ holds). To prove $C2_{i+1}$, $C3_{i+1}$ and $C4_{i+1}$, the following cases are distinguished (due to the grammar of statements, see figure 1 and section 9.2).

Case 1.1: *Stat* is a controlflow statement ControlFlowS, or an assert statement assert BoolE. Then $\Sigma[i]$ and $\Sigma[i+1]$ have the same stack and object store (language property, as boolean expressions do not have side effects, see also section 5.2). Then $C2_{i+1}$, $C3_{i+1}$ and $C4_{i+1}$ hold. Then $IH(i+1)$ holds.

Case 1.2: *Stat* is a local variable declaration $T\ v$. Then $\Sigma[i]$ and $\Sigma[i+1]$ have the same object store, and stacks that only differ on v. Then $C3_{i+1}$ holds (same object store), and as v is not mapped to an object in $\Sigma[i+1]$, $C2_{i+1}$ and $C4_{i+1}$ hold (same object store, stack only differs on v). Then $IH(i+1)$ holds.

Case 1.3: *Stat* is a field assignment $C.f = \mathit{SimpleE}$. Then $\Sigma[i]$ and $\Sigma[i+1]$ have the same stack, and object stores that only differ on $C.f$. Then $C2_{i+1}$ holds (same $\mathit{reach}()$), and $C3_{i+1}$ holds (the values of locations $Y.f$ are unchanged), and $C4_{i+1}$ holds (same stack). Then $IH(i+1)$ holds.

Case 1.4: *Stat* is a field assignment $r.f = \mathit{SimpleE}$. Then r is this (SR8.1). Let $\Sigma[i]$'s stack map this to object Z. Then $\Sigma[i]$ and $\Sigma[i+1]$ have the same stack, and object stores that only differ on location $Z.f$.

Then $C4_{i+1}$ holds.

To prove $C2_{i+1}$, assume object $Z' \in \mathit{reach}(X, \Sigma[i+1])$. Two cases can be distinguished.

Case 1.4.1: $Z' \in \mathit{reach}(X, \Sigma[i])$. Then Z' is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ holds). Then Z' is framed by (Y, C) in $\Sigma[i+1]$.

Case 1.4.2: $Z' \notin \mathit{reach}(X, \Sigma[i])$. Then there is an object W such that *SimpleE* is mapped to W in $\Sigma[i]$, and $Z' \in \mathit{reach}(W, \Sigma[i])$, and $Z \in \mathit{reach}(X, \Sigma[i])$ and f is of a peer or rep type (only $Z.f$ is changed by the assignment). Then $\mathit{ownmod}(\mathsf{this}.f) = \mathsf{rep}$ or $\mathit{ownmod}(\mathsf{this}.f) = \mathsf{peer}$. Then $\mathit{ownmod}(\mathit{SimpleE}) = \mathsf{rep}$ or $\mathit{ownmod}(\mathit{SimpleE}) = \mathsf{peer}$ (standard type correctness). Then $W \in \mathit{reach}(Z, \Sigma[i])$ (given ownership safety, proof by structural induction on SimpleE is straightforward). Then $Z' \in \mathit{reach}(X, \Sigma[i])$ (transitivity of *reach*). Then Z' is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ holds). Then Z' is framed by (Y, C) in $\Sigma[i+1]$.

In both cases, $C2_{i+1}$ holds.

To prove $C3_{i+1}$, two cases can be distinguished.

Case 1.4.1: $Z \neq Y$ or *SimpleE* is not mapped to X in $\Sigma[i]$. Then $C3_{i+1}$ holds as $C3_i$ holds and no location $Y.f$ was changed to X by the assignment.

Case 1.4.2: $Z = Y$ and *SimpleE* is mapped to X in $\Sigma[i]$. Note that Y owns X ((Y, C) frames X). Then $\mathit{ownmod}(\mathit{SimpleE}) \neq \mathsf{root}$ ($\Sigma[i]$ is ownership safe). Two cases can be distinguished.

Case 1.4.2.1: $\mathit{ownmod}(\mathit{SimpleE}) = \mathsf{any}$. Then $\mathit{ownmod}(\mathsf{this}.f) = \mathsf{any}$ (standard type correctness). Then f has an any modifier (by definition of $\mathit{ownmod}()$). Then $C3_{i+1}$ holds as (1) $C3_i$ holds and (2) only location $Y.f$ is changed by the assignment, and (3) f is not of a rep type.

Case 1.4.2.2: $\mathit{ownmod}(\mathit{SimpleE}) = \mathsf{rep}$ or $\mathit{ownmod}(\mathit{SimpleE}) = \mathsf{owned}$. Then *SimpleE* is not this (this is of a peer type). Two cases can be distinguished (language property).

Case 1.4.2.2.1: *SimpleE* is a stack variable v. Then v is of a rep type (SR8.5). Then control is in class C ($C4_i$ holds), and this.f is of a rep type or an owned type (standard type correctness). Then f is of a rep type or of an owned type (definition of $\mathit{ownmod}()$). Then f is of a rep type (SR8.5).

Case 1.4.2.2.2: *SimpleE* is a reference $r'.f'$. Then r' is this (by definition of $\mathit{ownmod}()$ and, if $\mathit{ownmod}(\mathit{SimpleE}) = \mathsf{owned}$, by SR6.1). Then location $Y.f'$ is mapped to X in $\Sigma[i]$ (*SimpleE* is mapped to X and this is mapped to Y), and f' is of a rep type (by definition of $\mathit{ownmod}()$ and SR8.5). Then control is in class C (due to $C4_i$), and *SimpleE* is of a rep type (f' is of a rep type). Then this.f is of a rep type (standard type correctness). Then f is of a rep type (by definition of $\mathit{ownmod}()$).

In either case, control is in class C in $\Sigma[i]$, and f is of a rep type. Then f is private (SR8.4). Then f is defined in class C (control is in C). Then $C3_{i+1}$ holds.

In all cases, $C3_{i+1}$ holds.

Then $IH(i+1)$ holds.

Case 1.5: *Stat* is an assignment $v = E$. Then $\Sigma[i]$ and $\Sigma[i+1]$ have the same object store, and stacks that only differ on v. E is either a reference r, or a type cast $(T)r$ (as E is not a method call).

To prove $C2_{i+1}$, assume object $Z \in \mathit{reach}(X, \Sigma[i+1])$. Two cases can be distinguished.

Case 1.5.1: $Z \in \mathit{reach}(X, \Sigma[i])$. Then Z is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ holds). Then Z is framed by (Y, C) in $\Sigma[i+1]$. Then $C2_{i+1}$ holds.

Case 1.5.2: $Z \notin \mathit{reach}(X, \Sigma[i])$. Then there are objects W and W' such that E is mapped to W' in $\Sigma[i]$, and $Z \in \mathit{reach}(W', \Sigma[i])$, and control is with W in $\Sigma[i]$, and $W \in \mathit{reach}(X, \Sigma[i])$ and v is of a peer or rep type (as only v is changed by the assignment). Then $\mathit{ownmod}(E) = \mathsf{rep}$ or $\mathit{ownmod}(E) = \mathsf{peer}$ (standard type correctness). Note that W is owned by Y (as X is owned by Y, and $W \in \mathit{reach}(X, \Sigma[i])$, and as $\Sigma[i]$ is ownership safe).

Then **root** is not the direct owner of W' (v is mapped to W' in $\Sigma[i+1]$, and Σ is ownership safe). Then r is not of a root type (r is mapped to W' in $\Sigma[i]$ and Σ is ownership safe). Furthermore, r is not of an any type (standard type correctness: v is not of an any type and down-casting of any types is disallowed by SR8.7). Then r is of a peer, rep or owned type. Then $W' \in \mathit{reach}(W, \Sigma[i])$. Then $Z \in \mathit{reach}(X, \Sigma[i])$ (transitivity of reach). Then Z is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ holds). Then Z is framed by (Y, C) in $\Sigma[i+1]$. Then $C2_{i+1}$ holds.

In both cases, $C2_{i+1}$ holds.

$C3_{i+1}$ holds as $C3_i$ holds and as $\Sigma[i]$ and $\Sigma[i+1]$ have the same object store.

To prove $C4_{i+1}$, assume control is with Y in $\Sigma[i+1]$, and assume the stack of $\Sigma[i+1]$ maps stack variable w to X, and w is of a rep type. Then control is with Y in $\Sigma[i]$. The following cases can be distinguished.

Case 1.5.1: The stack of $\Sigma[i]$ maps w to X. Then control is in class C in $\Sigma[i]$ ($C4_i$ holds). Then control is in class C in $\Sigma[i+1]$. Then $C4_{i+1}$ holds.

Case 1.5.2: The stack of $\Sigma[i]$ does not map w to X. Then $v = w$ and E is mapped to X in $\Sigma[i]$ (as the stacks only differ on v). Then E is of a rep type (standard type correctness: w is of a rep type). Then r is mapped to X in $\Sigma[i]$ (E is mapped to X). Then r is not of a root type (X is owned by Y and Σ is ownership safe). Then r is of a rep, owned, or any type. r is not of an owned type (by definition of ownmod: SR8.5 forbids fields and local variables of owned types, and fields of rep types are private (SR8.4). r is not of an any type (standard type correctness: w is not of an any type and down-casting of any types is disallowed by SR8.7). Therefore, r is of a rep type. Then r differs from this (this is of a peer type). Furthermore, r is not a reference $C.f_0 \ldots f_j$ (static field f_0 is not of a rep type, see section 9.1). Two cases can be distinguished.

Case 1.5.2.1: r is a stack variable w. Then $\Sigma[i]$'s stack maps w to X, and w is of a rep type. Then control is in C in $\Sigma[i]$ ($C4_i$ holds and control is with Y in $\Sigma[i]$). Then control is in C in $\Sigma[i+1]$. Then $C4_{i+1}$ holds.

Case 1.5.2.2: r is a reference $\mathsf{this}.f_0 \ldots f_j$ ($j \geq 0$). Then f_0 is of a rep type, and $f_1, \ldots, f_j$ are of a peer type (proof by structural induction on the shape of references is straightforward given the definition of $\mathit{ownmod}()$). Let location $Y.f_0$ be mapped to an object Z in $\Sigma[i]$. Then Y directly owns Z (Σ is ownership safe), and $X \in \mathit{reach}(Z, \Sigma[i])$ (r is mapped to X). Then Z is framed by (Y, C) in $\Sigma[i]$ (from $C2_i$ and lemma A.10). Then f_0 is defined in C ($C3_i$ holds). Then control is in C in $\Sigma[i]$ (SR8.4: fields of rep types are private). Then control is in C in $\Sigma[i+1]$. Then $C4_{i+1}$ holds.

In all cases, $C4_{i+1}$ holds.

Then $\mathit{IH}(i+1)$ holds.

Case 2: The program counter in $\Sigma[i]$ is at a methodcall statement $v = E$. Then $\Sigma[i+1]$ is a prestate (by definition of 'prestate').

To prove $C1_{i+1}$, assume control is with X in $\Sigma[i+1]$. X is owned by Y (as X is framed by (Y, C), proof is straightforward). Then X is not directly owned by **root**. Then $\mathit{owner}(\Sigma[i+1) \neq \mathbf{root}$. Then Stat is not a static method call. As Σ satisfies ownership encapsulation, two cases can be distinguished.

Case 2.1: $\mathit{owner}(\Sigma[i+1]) = \mathit{owner}(\Sigma[i])$. Then $\mathit{owner}(\Sigma[i]) \neq \mathbf{root}$. Then control in $\Sigma[i]$ is with an object Z that has the same direct owner as X. Then Z is owned by Y (X is owned by Y). Three cases can be distinguished.

Case 2.1.1: E is an object construction $\mathsf{new}\ \mathit{ownmod}\ D(\ldots)$. Then ownmod is peer (Z and X have the same direct owner, which differs from **root**). Then Z has the same direct frame as X (by definition). Then Z is framed by (Y, C) in $\Sigma[i]$.

Case 2.1.2: E is a superclass constructor call $D(\ldots)$. Then $Z = X$. Then Z is framed by (Y, C) in $\Sigma[i]$.

Case 2.1.3: E is a call $r.m(\ldots)$. Then r is mapped to X in $\Sigma[i]$ (control is with X in $\Sigma[i+1]$). Then r is of a peer or any type (X and Z have the same direct owner and $\Sigma[i]$ is ownership safe). Then r is of a peer type (SR8.8). Then $X \in \mathit{reach}(Z, \Sigma[i])$. Then Z is framed by (Y, C) in $\Sigma[i]$ (due to $C2_{i+1}$ and lemma A.10, as Z is owned by Y).

Note that in all three cases, Z is framed by (Y, C) in $\Sigma[i]$.

Case 2.2: Control is with $\mathit{owner}(\Sigma[i+1])$ in $\Sigma[i]$. Then control in $\Sigma[i]$ is with an object Z that directly owns X. Then E is not a superclass constructor call $D(\ldots)$ (in that case, control would be with X in $\Sigma[i]$). Two cases can be distinguished.

Case 2.2.1: E is an object construction new $\mathit{ownmod}\ D(\ldots)$. Then ownmod is rep (as Z directly owns X). Let control in $\Sigma[i]$ be in a class E. Then the direct frame of X is (Z, E). Then two cases can be distinguished (by definition of 'framed': X is framed by (Y, C)).

Case 2.2.1.1: Z is framed by (Y, C).

Case 2.2.1.2: $Y = Z$. Then $E = C$ (lemma A.10). Then control is with Y and in C in $\Sigma[i]$ (direct frame is set on allocation). Then control is in C and with Y in $\mathit{top}(\mathit{callstack}(\Sigma[i]))$. Then $\Sigma[i+1]\#\mathit{top}(\mathit{callstack}(\Sigma[i]))$ is a prefix of $\mathit{callstack}(\Sigma[i+1])$ such that (1) the last element of the prefix is a prestate in which control is with Y and in a class $D \subseteq C$, and (2) every other element of the prefix is a prestate in which control is with an object owned by Y. Then $C1_{i+1}$ holds.

Case 2.2.2: E is a call $r.m(\ldots)$. Then r is mapped to X in $\Sigma[i]$. Then r is not of a root type (as Y is the direct owner of X and $\Sigma[i]$ is ownership safe). Then r is of a rep, owned or any type. r is not of an any type (SR8.8), and r is not of an owned type (SR8.5 forbids fields and stack variables of owned types, and SR8.4 requires fields of rep types to be private). Then r is of a rep type. Two cases can be distinguished (as Y owns X and Z directly owns X).

Case 2.2.2.1: Y owns Z. Then $X \in \mathit{reach}(Z, \Sigma[i])$ (r is mapped to X). Then Z is framed by (Y, C) in $\Sigma[i]$ (due to $C2_{i+1}$ and lemma A.10, as Z is owned by Y).

Case 2.2.2.2: $Y = Z$. Then either (1) r is a reference this.$f_0 \ldots f_j$ ($j \geq 0$), and f_0 is of a rep type, and $f_1, \ldots, f_j$ are of a peer type, or (2) r is a reference $w.f_1 \ldots f_j$, ($j \geq 0$), and w is of a rep type, and $f_1, \ldots, f_j$ are of a peer type (proof by structural induction on the shape of references is straightforward given the definition of $\mathit{ownmod}()$). As proof of both cases is similar, we only consider the first case (the second case only uses $C4_i$ instead of $C3_i$). Let location $Y.f_0$ be mapped to an object W in $\Sigma[i]$. Then Y directly owns W (Σ is ownership safe), and $X \in \mathit{reach}(W, \Sigma[i])$ (r is mapped to X). Then W is framed by (Y, C) in $\Sigma[i]$ (due to $C2_i$ and lemma A.10). Then f_0 is defined in C ($C3_i$ holds). Then control is in C in $\Sigma[i]$ (SR8.4: a field of a rep type is private). Then control is in C and with Y in $\mathit{top}(\mathit{callstack}(\Sigma[i]))$. Then $\Sigma[i+1]\#\mathit{top}(\mathit{callstack}(\Sigma[i]))$ is a prefix of $\mathit{callstack}(\Sigma[i+1])$ such that (1) the last element of the prefix is a prestate in which control is with Y and in a class $D \subseteq C$, and (2) every other element of the prefix is a prestate in which control is with an object owned by Y. Then $C1_{i+1}$ holds.

In all cases above, either $C1_{i+1}$ holds, or control in $\Sigma[i]$ is with an object Z that is framed by (Y, C). In the latter case, there is a prefix Σ' of $\mathit{callstack}(\Sigma[i])$ such that the last element of Σ' is a prestate in which control is with Y and in a class $D \subseteq C$, and every other element of Σ' is a prestate in which control is with an object owned by Y (due to $C1_i$). Then $\Sigma[i+1]\#\Sigma'$ is a prefix of $\mathit{callstack}(\Sigma[i+1])$. Then $C1_{i+1}$ holds (control is with an object owned by Y in $\Sigma[i+1]$). Then $C1_{i+1}$ holds in all cases.

To prove $C2_{i+1}$, assume object $Z \in \mathit{reach}(X, \Sigma[i+1])$. Two cases can be distinguished.

Case 2.1: $Z \in \mathit{reach}(X, \Sigma[i])$. Then Z is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ holds), and therefore in $\Sigma[i+1]$. Then $C2_{i+1}$ holds.

Case 2.2: $Z \notin \mathit{reach}(X, \Sigma[i])$. The following cases can be distinguished.

Case 2.2.1: E is an object construction new $\mathit{ownmod}\ D(\ldots)$. Then control is with a newly allocated object W in $\Sigma[i+1]$ (language property). Then $W \notin \mathit{reach}(X, \Sigma[i+1])$. Let σ be the state like $\Sigma[i+1]$, but with an empty stack. Then $Z \in \mathit{reach}(X, \sigma)$ (the stack of $\Sigma[i+1]$ only maps formal parameters to values, which are treated as fields of W). Then $Z \in \mathit{reach}(X, \Sigma[i])$ (the object stores of $\Sigma[i]$ and $\Sigma[i+1]$ differ only on fields of W). As this contradicts the above, this case is not feasible.

Case 2.2.2: E is a static method call $C.m(\ldots)$. Then control is not with an object in $\Sigma[i+1]$. Let σ be the state like $\Sigma[i+1]$, but with an empty stack. Then $Z \in \mathit{reach}(X, \sigma)$ (as the stack is not relevant when control is not with an object). Then $Z \in \mathit{reach}(X, \Sigma[i])$ (as $\Sigma[i]$ and $\Sigma[i+1]$ have the same object store). As this contradicts the above, this case is not feasible.

Case 2.2.3: E is a call $r.m(\ldots)$ or a superclass constructor call $D(\ldots)$. Then $\Sigma[i]$ and $\Sigma[i+1]$ have the same object store, and the stack of $\Sigma[i+1]$ only maps formal parameters to values, and control is with an object W in $\Sigma[i+1]$ (language property). Then $W \in \mathit{reach}(X, \Sigma[i])$ (same object store, and the stack is not relevant as control is with W in $\Sigma[i+1]$). Then W is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with W for Z). Furthermore, there is a formal parameter p of a rep or a peer type that is mapped to an object V in $\Sigma[i+1]$, and $W \in \mathit{reach}(X, \Sigma[i+1])$ and $Z \in \mathit{reach}(V, \Sigma[i+1])$ ($Z \notin \mathit{reach}(X, \Sigma[i])$). Then there is an actual parameter r' of a rep or peer type that is mapped to V in $\Sigma[i]$ (actual parameters are assigned to formal parameters), and $Z \in \mathit{reach}(V, \Sigma[i])$ (V can be picked such that the stack is not relevant). Two cases can be distinguished.

Case 2.2.3.1: E is a superclass constructor call $D(\ldots)$. Then control in $\Sigma[i]$ is with W. Then actual parameter r' is of a rep or a peer type (standard type correctness: formal parameter p is of a rep or peer type). Then $V \in \mathit{reach}(W, \Sigma[i])$. Then $Z \in \mathit{reach}(W, \Sigma[i])$ (transitivity of $\mathit{reach}()$). Then Z is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with W for X). Z is framed by (Y, C) in $\Sigma[i+1]$. Then $C2_{i+1}$ holds.

Case 2.2.3.2: E is a call $r.m(\ldots)$. Note that Y owns W (as W is framed by (Y, C)). Then $\mathit{owner}(\Sigma[i+1]) \neq \mathbf{root}$ (control is with W in $\Sigma[i+1]$). Then $\mathit{owner}(\Sigma[i]) \neq \mathbf{root}$ (Σ satisfies ownership encapsulation). Then control is with an object U in $\Sigma[i]$ (control is not in a static method). Note that r is mapped to W in $\Sigma[i]$ (control is with W in $\Sigma[i+1]$). r is not of an any type (SR8.8), and not of an owned type (SR8.5 and the definition of $\mathit{ownmod}()$), and not of a root type (W is owned by Y and Σ is ownership safe). Then $W \in \mathit{reach}(U, \Sigma[i])$ (r is mapped to W, and r is of a rep or peer type). Then two cases can be distinguished.

Case 2.2.3.2.1: r is of a rep type. Two conclusions can be drawn. (1) Then r differs from this (this is of a peer type). Then $m()$ is not private (language property). Then p is not of a rep type (SR8.6. Then p is of a peer type. Then $\mathit{ownmod}(r') = \mathit{ownmod}(r) \oplus \mathsf{peer} = \mathsf{rep} \oplus \mathsf{peer} = \mathsf{rep}$ (standard type correctness). Then $V \in \mathit{reach}(U, \Sigma[i])$ (r' is mapped to V), and U directly owns V (Σ is ownership safe and r' is of a rep type). (2) Then $W \in \mathit{reach}(U, \Sigma[i])$ and U directly owns W (Σ is ownership safe). Then two cases can be distinguished (as Y owns W).

Case 2.2.3.2.1.1: U is owned by Y in $\Sigma[i]$. Then there is a class D such that U is framed by (Y, D) in $\Sigma[i]$ (lemma A.10 with U for X). Then U is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with U for X and W for Z, as W is framed by (Y, C)). Then V is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with U for X and V for Z). Then Z is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with V for X). Then $C2_{i+1}$ holds.

Case 2.2.3.2.1.2: $Y = U$. Then either (1) r is a reference this.$f_0 \ldots f_j$ ($j \geq 0$), and f_0 is of a rep type, and $f_1, \ldots, f_j$ are of a peer type, or (2) r is a reference $w.f_1 \ldots f_j$, ($j \geq 0$), and w is of a rep type, and $f_1, \ldots, f_j$ are of a peer type. As proof of both cases is similar, we only consider the first case (the second case only uses $C4_i$ instead of $C3_i$). Let $U.f_0$ be mapped to an object T in $\Sigma[i]$. Then $W \in \mathit{reach}(T, \Sigma[i])$ (r is mapped to Z), and T is directly owned by Y (Σ is ownership safe). Then there is a class D such that T is framed by (Y, D) in $\Sigma[i]$ (lemma A.10 with T for X). Then T is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with T for X and W for Z, as W is framed by (Y, C)). Then f_0 is defined in C ($C3_i$ with T for X). Then control is in C in $\Sigma[i]$ (SR8.4: a field of a rep type is private). As actual parameter r' is of a rep type, either (1) r' is a reference this.$g_0 \ldots g_j$ ($j \geq 0$), and g_0 is of a rep type, and $g_1, \ldots, g_j$ are of a peer type, or (2) r' is a reference $w'.g_0 \ldots g_j$ ($j \geq 0$), and w' is of a rep type, and $g_1, \ldots, g_j$ are of a peer type. As proof of both cases is similar, we only consider the first case. Let $Y.g_0$ be mapped to an object S in $\Sigma[i]$. Then $V \in \mathit{reach}(S, \Sigma[i])$ (r' is mapped to V), and S is directly owned by Y (g_0 is of a rep type and Σ is ownership safe). Then there is a class D such that S is framed by (Y, D) in $\Sigma[i]$ (lemma A.10 with S for X). Note that g_0 is defined in class C (control is in class C and SR8.4 ensures fields of a rep type are private). Then S is framed by (Y, C) in $\Sigma[i]$ ($C3_i$ with S for X and g_0 for f). Then V is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with S for X and V for Z). Then Z is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with V for X). Then $C2_{i+1}$ holds.

Case 2.2.3.2.2: r is of a peer type. Then U and W have the same direct owner (Σ is ownership safe). Then U is owned by Y in $\Sigma[i]$. Then there is a class D such that U is framed by (Y, D) in $\Sigma[i]$ (lemma A.10 with U for X). Then U is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with U for X and W for Z, as W is framed by (Y, C)). Then two cases can be distinguished.

Case 2.2.3.2.2.1: p is of a rep type. Note that $\mathit{ownmod}(r) \oplus \mathsf{rep} \in \{\mathsf{any}, \mathsf{owned}\}$. Then r

is this (SR6.1, a context switch is treated as an assignment of actual to formal parameters). Then actual parameter r' is of a rep type (standard type correctness: $\mathit{ownmod}(r.p) = \mathsf{rep}$).
Case 2.2.3.2.2.2: p is of a peer type. Then actual parameter r' is of a peer type (standard type correctness: $\mathit{ownmod}(r.p) = \mathsf{peer}$).
In either case, r' is of a rep or a peer type. Then $V \in \mathit{reach}(U, \Sigma[i])$ (as r is mapped to V). Then V is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with U for X and V for Z). Then Z is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with V for X). Then $C2_{i+1}$ holds.
In each of these cases, C2(i+1 holds).

$C3_{i+1}$ holds as $C3_i$ holds and as every location that is mapped to a value in $\Sigma[i]$, is mapped to the same value in $\Sigma[i+1]$.

To prove $C4_{i+1}$, assume control is with Y in $\Sigma[i+1]$, and assume $\Sigma[i+1]$'s stack maps stack variable v to X, and assume v is of a rep type. Then v is a formal parameter p ($\Sigma[i+1]$'s stack only maps formal parameters to values). Then E is a private method call (SR8.6). X is not newly allocated in $\Sigma[i]$ (as X is framed by (Y, C)). Then E is not an object construction new $\mathit{ownmod}\ D(\ldots)$. E is not a static method call (control is with Y in $\Sigma[i+1]$). E is not a superclass constructor call (E is a private method call). Then E is an instance method call this.$m(\ldots)$. Then control is with Y in $\Sigma[i]$. Then there is an actual parameter r that is mapped to X in Σ (p is mapped to X in $\Sigma[i+1]$), and r is of a rep type (parameters are treated as fields of this, standard type correctness: $\mathit{ownmod}(\mathsf{this}.p) = \mathsf{rep}$). Then either (1) r is a reference this.$f_0 \ldots f_j$ ($j \geq 0$), and f_0 is of a rep type, and $f_1, \ldots, f_j$ are of a peer type, or (2) r is a reference $w.f_1 \ldots f_j$, ($j \geq 0$), and w is of a rep type, and $f_1, \ldots, f_j$ are of a peer type. As proof of both cases is similar, we only consider the first case (the second case only uses $C4_i$ instead of $C3_i$). Let $Y.f_0$ be mapped to an object Z in $\Sigma[i]$. Then $X \in \mathit{reach}(Z, \Sigma[i])$ (r is mapped to X), and Z is directly owned by Y (Σ is ownership safe, f_0 is of a rep type). Then there is a class D such that Z is framed by (Y, D) in $\Sigma[i]$ (lemma A.10 with Z for X). Then Z is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with Z for X and X for Z). Then f_0 is defined in class C ($C3_i$ with Z for X). Then control is in C in $\Sigma[i]$ (SR8.4: a field of a rep type is private). Then control is in C in $\Sigma[i+1]$ (E is a private method call). Then $C4_i$ holds.

Then $IH(i+1)$ holds.

Case 3: The program counter in $\Sigma[i]$ is not at a statement (i.e., it is at the end of a method). Then $\Sigma[i]$ is a poststate (by definition of 'poststate'). Let $\Sigma[j]$ be the prestate matching $\Sigma[i]$. Then $\Sigma[j..i]$ is a method execution (by definition), and the program counter in $\Sigma[j-1]$ is at a methodcall statement $v = E$ (by definition of 'prestate'). Then $\Sigma[i]$ and $\Sigma[i+1]$ have the same object store, and $\Sigma[j-1]$ and $\Sigma[i+1]$ have stacks that only differ on v (language property). Then $\mathit{callstack}(\Sigma, i+1) = \mathit{callstack}(\Sigma, j-1)$ (lemma A.4).

To prove $C1_{i+1}$, assume control is with X in $\Sigma[i+1]$. Then control is with X in $\Sigma[j-1]$ (axiom A.1). Then X is framed by (Y, C) in $\Sigma[j-1]$ (frame does not change after allocation). Then there is a prefix Σ' of $\mathit{callstack}(\Sigma, j-1)$ such that the last element of Σ' is a prestate in which control is with Y and in a class $D \subseteq C$, and every other element of Σ' is a prestate in which control is with an object owned by Y ($C1_{j-1}$ holds). Then $C1_{i+1}$ holds ($\mathit{callstack}(\Sigma, i+1) = \mathit{callstack}(\Sigma, j-1)$).

To prove $C2_{i+1}$, assume object $Z \in \mathit{reach}(X, \Sigma[i+1])$. Two cases can be distinguished.
Case 3.1: $Z \in \mathit{reach}(X, \Sigma[i])$. Then Z is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ holds). Then Z is framed by (Y, C) in $\Sigma[i+1]$. Then $C2_{i+1}$ holds.
Case 3.2: $Z \notin \mathit{reach}(X, \Sigma[i])$. Then control in $\Sigma[i+1]$ is with an object W, and $W \in \mathit{reach}(X, \Sigma[i+1])$, and $Z \in \mathit{reach}(W, \Sigma[i+1])$ ($\Sigma[i]$ and $\Sigma[i+1]$ have the same object store, so the stack must be relevant, which means control is not in a static method). Then there is a stack variable w of a rep or peer type that is mapped to an object V in $\Sigma[i+1]$, and $Z \in \mathit{reach}(V, \Sigma[i+1])$, and $Z \in \mathit{reach}(V, \Sigma[i])$($\Sigma[i]$ and $\Sigma[i+1]$ have the same object store). Let σ be the state like $\Sigma[i+1]$. Then $W \in \mathit{reach}(X, \sigma)$ (control is in W in $\Sigma[i+1]$, so $\Sigma[i+1]$'s stack is not relevant). Then $W \in \mathit{reach}(X, \Sigma[i])$ (as $\Sigma[i]$ and $\Sigma[i]$ have the same object store). Then W is framed by (Y, C) in $\Sigma[i]$ ($C1_i$ with W for Z). Note that control is with W in $\Sigma[j-1]$ (lemma A.4 and axiom A.1). Two cases can be distinguished.
Case 3.2.1: $w \neq v$. Then w is mapped to V in $\Sigma[j-1]$ (language property). Then $V \in$

$reach(W, \Sigma[j-1])$. Then V is framed by (Y, C) in $\Sigma[j-1]$ ($C2_{j-1}$ with W for X and V for Z). Then V is framed by (Y, C) in $\Sigma[i]$. Then Z is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with V for X).

Case 3.2.2: $w = v$. Then result is mapped to V in $\Sigma[i]$ (language property). Then E is not a superclass constructor call $D(\ldots)$ (which returns null), and E is of a peer or rep type (standard type correctness: v is of a peer or per type). Then E is not a static method call (which cannot be of a peer or rep type). Then control is with an object U in $\Sigma[i]$, and two cases can be distinguished.

Case 3.2.2.1: E is an object construction new $ownmod$ $D(\ldots)$. Then $ownmod$ is rep or peer (E is of a peer or rep type). Then two cases can be distinguished.

Case 3.2.2.1.1: There is a class D such that (W, D) is the direct frame of U. Then U is framed by (Y, C) (by definition of framed: Y, C frames W).

Case 3.2.2.1.2: U has the same direct frame as W. Then U is framed by (Y, C) (by definition of framed: Y, C frames W).

In either case, U is framed by (Y, C). As $V = U$ (the newly constructed object is returned), $Z \in reach(U, \Sigma[i])$. Then Z is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with U for X). Then Z is framed by (Y, C) in $\Sigma[i+1]$. Then $C2_{i+1}$ holds.

Case 3.2.2.2: E is a method call $r.m(\ldots)$. Then r is mapped to U in $\Sigma[j-1]$ (as control is with U in $\Sigma[j]$), and both $ownmod(r)$ and the return type of $m()$ are either rep or peer (standard type correctness: $ownmod(E) =$ peer). Then $U \in reach(W, \Sigma[j-1])$. Then U is framed by (Y, C) in $\Sigma[j-1]$ ($C2_{j-1}$ with W for X and U for Z). Furthermore, result is of a rep or a peer type in $\Sigma[i]$ (the static type of result is the return type of $m()$). Then $V \in reach(U, \Sigma[i])$ (control is with U and result is mapped to V in $\Sigma[i]$). Then $Z \in reach(U, \Sigma[i])$ (transitivity of $reach()$). Then Z is framed by (Y, C) in $\Sigma[i]$ ($C2_i$ with U for X). Then Z is framed by (Y, C) in $\Sigma[i+1]$. Then $C2_{i+1}$ holds.

In all case above, $C2_{i+1}$ holds.

$C3_{i+1}$ holds as $C3_i$ holds and as $\Sigma[i]$ and $\Sigma[i+1]$ have the same object store.

To prove that $C4_{i+1}$ holds, assume control is with Y in $\Sigma[i+1]$, and assume $\Sigma[i+1]$'s stack maps stack variable w to X, and assume w is of a rep type. Note that $callstack(\Sigma, i+1) = callstack(\Sigma, j-1)$ (lemma A.4). Then control is with Y in $\Sigma[j-1]$ (axiom A.1). Two cases can be distinguished.

Case 3.1: $\Sigma[j-1]$'s stack maps w to X. Then control is in class C in $\Sigma[j-1]$ ($C4_{j-1}$ holds). Then control is in class C in $\Sigma[i+1]$ (axiom A.1). Then $C4_{i+1}$ holds.

Case 3.2: $\Sigma[j-1]$'s stack does not map w to X. Then $v = w$ (language property: $\Sigma[j-1]$ and $\Sigma[i+1]$ have stacks that only differ on v). Then result is mapped to X by $\Sigma[i]$'s stack (v is mapped to X in $\Sigma[i+1]$). As v is of a rep type, E is of a rep type. Two cases can be distinguished.

Case 3.2.1: E is an object construction new rep $D(\ldots)$. Then X is newly allocated in $\Sigma[j]$, and Y directly owns X (by definition). Then (Y, C) is the direct frame of X (by definition of 'direct owner' and 'direct frame'). Then control is in C in $\Sigma[j-1]$ (by definition). Then control is in C in $\Sigma[i+1]$ (axiom A.1). Then $C4_{i+1}$ holds.

Case 3.2.2: E is a method call $r.m(\ldots)$. Then r is not a reference $C'.f_0 \ldots f_k$ (as E is of a rep type and f_0 is not, see section 9.1). Two cases can be distinguished.

Case 3.2.2.1: r is this and $m()$ has a rep return type. Then control is with Y in $\Sigma[j]$ (language property). Then control is with Y in $\Sigma[i]$ (lemma A.2 and axiom A.1). Then control is in class C in $\Sigma[i]$ (from $C4_i$, as result has a rep type and is mapped to X). As $m()$ has a rep return type, $m()$ is private (SR8.6). Then control is in class C in $\Sigma[j-1]$. Then control is in class C in $\Sigma[i+1]$ (axiom A.1). Then $C4_{i+1}$ holds.

Case 3.2.2.2: r is a reference this.$f_0 \ldots f_k$, $0 \leq k$, such that f_0 is of a rep type, and $f_1, \ldots, f_k$ are of a peer type and $m()$ has a peer return type. Let location $Y.f_0$ be mapped to an object Z in $\Sigma[j-1]$. Then Y directly owns Z (Σ is ownership safe), and $X \in reach(Z, \Sigma[j-1])$ (as r is mapped to X). Then there is a class D such that Z is framed by (Y, D) in $\Sigma[j-1]$ (lemma A.10). Then Z is framed by (Y, C) in $\Sigma[j-1]$ ($C2_i$ with Z for X and X for Z). Then f_0 is defined in C ($C3_i$ holds). Then control is in C in $\Sigma[j-1]$ (SR8.4: fields of rep types are private). Then control is in C in $\Sigma[i+1]$ (axiom A.1). Then $C4_{i+1}$ holds.

Case 3.2.2.3: r is a reference $v'.f_1 \ldots f_k$, $0 \leq k$, such that v' is of a rep type, and $f_1, \ldots, f_k$ are of a peer type and $m()$ has a peer return type. Proof of this case is omitted as it is similar to the previous case (it only uses $C4_i$ instead of $C3_i$).

In all cases, $C4_{i+1}$ holds.

Then $IH(i+1)$ holds.
In all cases, IH(i+1) holds. That concludes the proof of the step case.

A.10 proof of lemma 8.7

This section contains a proof of the downward local consistency lemma, lemma 8.7.

In this section, *Prem* is used as a shorthand for the premisse of lemma 8.7:

Prem: Every invariant in program P is ownership admissible, and P meets the type-correctness rules of the eUTS, SR8.1-SR8.8, PO8.2, PO8.3, and PO8.5.

Proof is by generalized induction on the length of program execution Σ in lemma 8.7. That is, we prove that given *Prem*, if Σ is an execution of P, then for every i, $IH(i)$ holds, where

$IH(i)$: if control is with object X and in class C in $\Sigma[i]$, and $\Sigma[i]$ is either a relevant horizontal call state or a poststate in which X is not constructing, then X is consistent for $[type(X), C\rangle$ in $\Sigma[i]$.

Base ($i = 0$): $IH(0)$ holds trivially as $\Sigma[0]$ is neither a relevant horizontal call state nor a poststate.

Step: Assume *Prem*. Assume $IH(j)$ holds for every j such that $0 \leq j < i$. Then the goal is to prove $IH(i)$. Let Σ be an arbitrary execution of P. Then Σ satisfies classical encapsulation (lemma 8.2). Furthermore, Σ is ownership safe (lemma 6.1) and layer safe (lemma 6.2). Then Σ satisfies ownership encapsulation (lemma 8.4) and upward local consistency (lemma 8.6). $\Sigma[0..i-1]$ satisfies downward local consistency ($IH(j)$ holds for every j such that $0 \leq j < i$). Then $\Sigma[0..i-1]$ satisfies the LRII and the LRII-c (follows from the proof of theorem 7.1 in appendix A.3).

Assume that control is with object X and in class C in $\Sigma[i]$, and assume $\Sigma[i]$ is either a relevant horizontal call state or a poststate in which X is not under construction. Proof is by contradiction: Assume that X is not consistent for $[type(X), C\rangle$ in $\Sigma[i]$. Let X be of a class E. Then there is a class D, $E \subseteq D \subset C$, such that $inv_D(X)$ does not hold in $\Sigma[i]$. Let $top(callstack(\Sigma, i)) = \Sigma[j]$. Then $\Sigma[j]$ is a prestate (by definition of $callstack()$). Then $top(callstack(\Sigma, j)) = \Sigma[j]$ (by definition of $callstack()$). Then control is with X and in C in $\Sigma[j]$ (lemma A.3 and axiom A.1), and X is not constructing in $\Sigma[j]$ (as it is not constructing in $\Sigma[i]$). Then X is consistent for $[type(X), C\rangle$ in $\Sigma[j]$ (LRII holds for $\Sigma[0..i-1]$). Then there is a k, $j \leq k < i$, such that $inv_D(X)$ holds in $\Sigma[k]$, but not in any state in $\Sigma[k+1..i]$.

As inv_D is ownership admissible, inv_D is ownership based (lemma 8.7). Then two cases can be distinguished.

Case 1: $\Sigma[k]$ and $\Sigma[k+1]$ differ on X. Then $\Sigma[k]$ and $\Sigma[k+1]$ differ on a location $X.f$. Then control is with X in $\Sigma[k+1]$ (classical encapsulation), and inv_D contains a reference $\mathsf{this}.f$ (inv_D is ownership admissible). Then f is defined in class D (SR8.2). X is not newly allocated in $\Sigma[k+1]$ (as it is allocated in $\Sigma[j]$). Then the program counter of $\Sigma[k]$ is at an assignment $r.f = SimpleE$ (locations can only be changed by allocation an field assignment). Then r is this (SR8.1). Then control in $\Sigma[k]$ and $\Sigma[k+1]$ is in a class $E \subseteq D$ (SR8.1, f is defined in class D). Then $k+1 < i$ (control is in C in $\Sigma[i]$). Let $top(callstack(\Sigma, k+1)) = \Sigma[l]$. Then $\Sigma[l]$ is a prestate (by definition of $callstack()$). Then $top(callstack(\Sigma, l)) = \Sigma[l]$ (by definition of $callstack()$). Then control is with X and in a class $E \subseteq D$ in $\Sigma[l]$ (lemma A.3 and axiom A.1). Then $l \neq j$ (control is in C in $\Sigma[j]$). Then $l > j$ (lemma A.5).

Case 2: $\Sigma[k]$ and $\Sigma[k+1]$ differ on an object Y owned by X in $\Sigma[k+1]$. Then control is with Y in $\Sigma[k+1]$ (classical encapsulation), and inv_D contains a reference $\mathsf{this}.f_0 \dots f_i$ ($i > 0$) of a rep or owned type, such that $X.f_0 \dots f_j$ ($0 \leq j < i$) is mapped to Y in $\Sigma[k]$ (inv_D is ownership admissible). Then f_0 is defined in class D (SR8.2), and f_0 is of a rep or owned type (by definition of $ownmod()$: $\mathsf{this}.f_0 \dots f_i$ is of a rep or owned type). Then f_0 is of a rep type (SR8.5 forbids fields of a owned type). Let $X.f_0$ be mapped to an object Z in $\Sigma[k]$. Then X directly owns Z (Σ is ownership safe). Then there is a class F such that Z is framed by (X, F) in $\Sigma[k]$. Then Z is framed by (X, D) in $\Sigma[k]$ (from $C3_k$ of lemma A.11 with X for Y and Z for X, as f_0 is defined in class D). Note that $Y \in reach(Z, \sigma[k])$ (as $X.f_0 \dots f_j$ is mapped to Y). Then Y is framed by (X, D) in $\Sigma[k]$ (from $C2_k$ of lemma A.11 with (X, D) for (Y, C), Z for X and Y for Z). Then Y is framed by (X, D) in $\Sigma[k+1]$. Then there is a prefix Σ' of $callstack(\Sigma, k+1)$ such that (1) the last element of Σ' is a prestate $\Sigma[l]$ in which control is with X and in a class $E \subseteq D$, and (2) every other element of Σ' is a prestate in which control is with an object owned by X (from $C1_{k+1}$ of lemma A.11 with (X, D) for (Y, C) and Y for X). Then

$\Sigma[j] \notin \Sigma'$ (as control is with X and in C in $\Sigma[j]$). Then $l > j$ (proof by induction, using lemma A.5, is straightforward).

In either case, there is a prestate $\Sigma[l]$, $l > j$, such that control is with X and in a class $E \subseteq D$ in $\Sigma[l]$. Then there is a poststate $\Sigma[m]$, $m < i$, such that $top(callstack(\Sigma, m)) = \Sigma[l]$ (by definition of $callstack()$, as $top(callstack(\Sigma, i)) = j$). Then control is with X and in a class $E \subseteq D$ in $\Sigma[m]$ (lemma A.2 and axiom A.1). Then X is consistent in $\Sigma[m]$ ($\Sigma[0..i-1]$ satisfies LRII and LRII-c). Then $inv_D(X)$ holds in $\Sigma[m]$. As this contradicts that $inv_D(X)$ does not hold in any state in $\Sigma[k+1..i]$, X is consistent for $[type(X), C\rangle$ in $\Sigma[i]$. That concludes the proof of the step case.