A Unified Framework for Verification Techniques for Object Invariants

Sophia Drossopoulou Adrian Francalanza
Imperial College
{scd,adrianf}@doc.ic.ac.uk

Peter Müller
Microsoft Research, Redmond
mueller@microsoft.com

Abstract

Verification of object-oriented programs relies on object invariants to express consistency criteria of objects. The semantics of object invariants is subtle, mainly because of call-backs, multi-object invariants, and subclassing.

Several verification techniques for object invariants have been proposed. These techniques are complex and differ in restrictions on programs (for instance, which fields can be updated), restrictions on invariants (what an invariant may refer to), use of advanced type systems (such as ownership types), meaning of invariants (in which execution states are invariants assumed to hold), and proof obligations (when should an invariant be proven). As a result, it is difficult to compare and understand whether/why these techniques are sound, or to develop better techniques.

In this paper, we develop a unified framework to describe verification techniques for object invariants. We separate type system concerns from verification strategy concerns. We distil seven parameters that characterise a verification technique, and identify sufficient conditions on these parameters under which a verification technique is sound. To illustrate the generality of our framework, we instantiate it with six verification techniques from the literature. We show how our framework facilitates the assessment and comparison of the soundness and expressiveness of these techniques.

1. Introduction

Object invariants play a dominant role in the specification and verification of object-oriented programs, and have been an integral part of all major specification languages for object-oriented programs such as Eiffel [28], the Larch languages [4, 16, 17], the Java Modeling Language JML [18], and Spec# [2]. Object invariants express consistency criteria for objects, which guarantee their correct working. These criteria range from simple properties of single objects (for instance, that a field is non-null) to complex properties of whole object structures (for instance, the sorting of a tree).

Most of the existing verification techniques expect object invariants to hold in the pre-state and post-state of method executions, often referred to as *visible states* [31]. Invariants may be broken temporarily between visible states. This semantics is illustrated by class C in Fig. 1. The invariant is established by the constructor. It may be assumed in the pre-state of method m. Therefore, the first statement in m's body can be proven not to cause a division-by-zero error. The invariant might temporarily be broken by the subsequent assignment to a, but it is later re-established by m's last statement; thus, the invariant holds in m's post-state.

```
class C {                                class Client {
  int a, b;                                C c;
  invariant 0 <= a < b;                    invariant c.a <= 10;

  C() { a := 0; b := 3; }                  /* methods omitted */
                                         }
  void m() {
    int k := 100 / (b − a);
    a := a + 3;
    n();                                 class D extends C {
    b := (k + 4) ∗ b;                      invariant a <= 10;
  }
  void n() { m(); }                        /* methods omitted */
}                                        }
```

Figure 1. An example (adapted from [21]) illustrating the three main challenges for the verification of object invariants.

While the basic idea of object invariants is simple, verification techniques for practical OO-programs face challenges. These challenges are more daunting for *modular* verification where classes are verified without knowledge of their clients and subclasses:

Call-backs: Methods that are called while the invariant of an object o is temporarily broken might call back into o and find the object in an inconsistent state. In our example (Fig. 1), during execution of **new** C().m() the assignment to a breaks the invariant, and the call-back via n() leads to a division by zero.

Multi-object invariants: When the invariant of an object p depends on the state of another object o, modifications of o potentially break the invariant of p. In our example, a call o.m might break the invariant of a Client object p where p.c $= o$. Aliasing makes the proof of preservation of p's invariant difficult. In particular, for *modular* verification of m, Client's invariant is not known and, thus, cannot be expected to be preserved.

Subclassing: When the invariant of a subclass D refers to fields declared in the superclass C then methods of C potentially break D's invariant by assigning to C's fields. In particular, for modular verification of C, the subclass invariant is in general not known and, thus, cannot be expected to be preserved.

Several verification techniques address some or all of these problems [1, 3, 14, 19, 21, 26, 29, 30, 31, 35]. They share many commonalities, but differ in the following important aspects:

1. *Invariant semantics:* What invariants are expected to hold in which execution states? Some techniques require all invariants

to hold in all visible states, whereas others address the multi-object invariant challenge by excluding certain invariants.

2. *Proof obligations:* Where are the proofs required? Some techniques require proofs for invariants relating to the current active object whereas others require invariant proofs for all objects in the heap.
3. *Invariant restrictions:* What objects may invariants depend on? Some techniques use unrestricted invariants, whereas others address the subclassing challenge by preventing invariants from referring to inherited fields.
4. *Program restrictions:* What objects may be used as receivers of field updates and method calls? Some techniques permit arbitrary field updates, whereas others simplify modular verification by allowing updates to fields of the current receiver only.
5. *Type systems:* What syntactic information is used for reasoning? Some techniques are designed for arbitrary programs, whereas others use ownership types to facilitate verification.

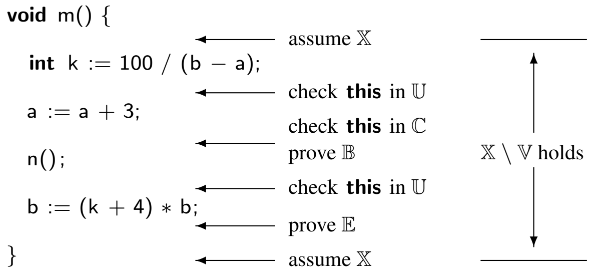
Furthermore, most techniques have no formal presentation. All these factors complicate verification technique comparisons and hinder proper understanding of why these techniques satisfy claimed properties such as soundness. Thus, it is hard to decide which technique to adopt or to develop new, sound techniques.

In this paper, we present a unified framework that formalises the proposed verification techniques, abstracts away from their differences, and allows comparisons. We concentrate on techniques that require invariants to hold in visible states, because they constitute the vast majority of those described in the literature. We formalise soundness for such verification techniques. Our formalisation advocates the separation of concerns between the type system and verification strategy. Even though the soundness of the verification technique relies on type system soundness, the two are intrinsically distinct: With such an approach we can investigate the verification strategy independently of the type system expressiveness. Such a separation allowed us to quickly discover the lack of soundness in one of the published techniques—more in section 5.

Approach. Our framework uses seven parameters to capture the first four aspects in which verification techniques differ. To describe these parameters, we use *object-areas* (*areas* for short) and *invariant-regions* (*regions* for short). The former statically characterise sets of objects, while the latter statically characterise sets of object-class pairs, and thus represent object invariants (each of which is an object, and the class that declares the invariant – the class component handles subclassing).

Thus, we describe the first four aspects as follows (the differences in type systems are discussed later):

1. *Invariant semantics:* The region $\mathbb{X}$ describes the invariants that are expected to hold in visible states. The region $\mathbb{V}$ describes the invariants that are *vulnerable* to a given method, that is, the invariants that the method may break while the control is within the method. The latter parameter is necessary because most techniques require some invariants to hold even between visible states, for instance, subclass invariants.
2. *Invariant restrictions:* The region $\mathbb{D}$ describes the invariants that may depend on a given heap location. This characterises indirectly the locations an invariant may depend on (*cf.* Def. 2).
3. *Proof obligations:* The regions $\mathbb{B}$ and $\mathbb{E}$ describe the invariants that have to be proven to hold before a method call and at the end of a method body, respectively.
4. *Program restrictions:* The areas $\mathbb{U}$ and $\mathbb{C}$ describe the permitted receivers for field updates and method calls, respectively.



```
void m() {
  int k := 100 / (b − a);
  a := a + 3;
  n();
  b := (k + 4) ∗ b;
}
```

assume $\mathbb{X}$
check **this** in $\mathbb{U}$
check **this** in $\mathbb{C}$
prove $\mathbb{B}$
check **this** in $\mathbb{U}$
prove $\mathbb{E}$
assume $\mathbb{X}$
$\mathbb{X} \setminus \mathbb{V}$ holds

Figure 2. Role of framework parameters for method m (Fig. 1).

Fig. 2 illustrates the role of these parameters. In the pre- and post-state of a method, $\mathbb{X}$ may be assumed to hold. Between these visible states, some object invariants may be broken, but $\mathbb{X} \setminus \mathbb{V}$ is known to hold throughout the method body. Field updates and method calls are allowed if the receiver object (here, **this**) is in $\mathbb{U}$ and $\mathbb{C}$, respectively. Before a method call, $\mathbb{B}$ must be proven. At the end of the method body, $\mathbb{E}$ must be proven. Finally, $\mathbb{D}$ (not shown in Fig. 2) constrains the effects of field updates on invariants. Thus, assignments to a and b affect at most $\mathbb{D}$.

Techniques expressed through these seven components carry numerous advantages over descriptions using words [35, 14, 19, 31] or typing rules[26]:

- The aspects in which verification techniques differ are distilled in terms of these components (*e.g.*, invariant semantics using $\mathbb{D}$, $\mathbb{U}$, $\mathbb{C}$, proof obligations using $\mathbb{B}$, $\mathbb{E}$.)
- Relationships between components can be expressed at an abstract level (*e.g.*, well-structured criteria in Def. 5.), and the interpretations of components as areas and regions allow formal comparisons of techniques in terms of set operations.

Developing our framework was challenging because different verification techniques (1) use different type systems to restrict programs and invariants, (2) use different specification languages to express invariants, and (3) use different verification logics. To deal with this diversity within one unified framework:

1. Instead of describing one particular type system, we state requirements on the type systems used with our framework.
2. We assume a judgement that describes that an object satisfies the invariant of a class in a heap. We require that a field update preserves the invariant if the invariant does not fall within $\mathbb{D}$.
3. We express proof obligations via a special construct prv $\mathbb{r}$, which throws an exception if the invariants in region $\mathbb{r}$ cannot be proven, and has an empty effect otherwise.

Contributions. The contributions of this paper are:

1. We present a unified formalism for verification techniques for object invariants.
2. We identify conditions on the framework which guarantee soundness of a verification technique.
3. We separate type system concerns from verification strategy concerns.
4. We show how our framework describes all major verification techniques for visible state invariants.
5. We prove soundness for five techniques, and guided by our framework, discovered a (repairable) unsoundness in the sixth.

Outline. Sec. 2 formalises programs and invariant semantics. Sec. 3 describes our framework and defines soundness. Sec. 4

$$
\begin{array}{lllll|ll}
e & ::= & \mathsf{this} & \textit{(this)} & & x & \textit{(variable)} \\
 & | & \mathsf{null} & \textit{(null)} & & \mathsf{new}\,t & \textit{(new object)} \\
 & | & e.f & \textit{(access)} & & e.f = e & \textit{(assignment)} \\
 & | & e.m(e) & \textit{(method call)} & & e\ \mathsf{prv}\ \mathbb{r} & \textit{(proof annotat.)} \\
 & & & & & & \\
e_r & ::= & \ldots & \textit{(as source exprs.)} & & v & \textit{(value)} \\
 & | & \mathsf{verfExc} & \textit{(verif exc.)} & & \mathsf{fatalExc} & \textit{(fatal exc.)} \\
 & | & \sigma \cdot e_r & \textit{(nested call)} & & \mathsf{call}\ e_r & \textit{(launch)} \\
 & | & \mathsf{ret}\ e_r & \textit{(return)} & & &
\end{array}
$$

Figure 3. Source and runtime expression syntax.

instantiates our framework with existing verification techniques. Sec. 5 presents sufficient conditions for a verification technique to be sound, and states a general soundness theorem. Sec. 6 discusses related work. Proofs and more details are presented in a report [8].

2. Invariant Semantics

We formalise invariant semantics through an operational semantics. This semantics defines at which execution points invariants are expected to hold. In order to cater for the different techniques, the semantics is parameterised by regions to express proof obligations and what invariants are expected to hold. In this section, we focus on the main ideas of our semantics and relegate the less interesting definitions to App. A. We assume sets of identifiers for class names CLS, field names FLD, and method names MTHD, and use variables $c \in$ CLS, $f \in$ FLD and $m \in$ MTHD.

Runtime Structures. A *runtime structure* is a tuple consisting of a set of heaps HP, addresses ADR, and values VAL $=$ ADR$\cup\{\mathsf{null}\}$, with the convention that $h \in$ HP and $\iota \in$ ADR. A runtime structure provides the following operations: *dom* represents the domain of a heap; *cls* yields the class of the object at a given address; *fld* yields the value of a field of the object at a given address in a heap; *upd* yields the new heap after a field update; *new* yields the heap and address resulting from an object creation. We do not describe how these operations work, but require properties about their behaviour, for instance, that *upd* only modifies the corresponding field of the object at the given address, and leaves the remaining heap unmodified. See Def. 9 (App. A) for details.

A stack frame $\sigma \in$ STK $=$ ADR$\times$ADR$\times$MTHD$\times$CLS is a tuple of a receiver address, an argument address, a method identifier, and a class. The latter two items indicate the method currently being executed and the class where it is defined.

Area/Region Structures and Types. An *area/region structure* (Def. 10 in App. A) consists of a set A of object-areas and a set R of invariant-regions. An area $\mathbb{a} \in$ A is a syntactic representation for a set of objects; a region $\mathbb{r} \in$ R is a syntactic representation for a set of object-class pairs.

Several verification techniques specify the invariants that may be assumed or have to be proven relative to a given *viewpoint* object. For instance, verification techniques using ownership [1, 26, 31] typically allow a method to assume the invariants of the viewpoint **this** and of all objects owned by **this**. To capture viewpoints, area/region structures provide the viewpoint adaptation operator $\triangleright$ [6], which adapts a region to the viewpoint described by an area.

We define a type, $t \in$ TYP, as a tuple of an area and a class, $\mathbb{a}\,c$. The area allows us to cater for types that express the topology of the heap, without being specific about the underlying type system.

Expressions. In Fig. 3, we define source expressions $e \in$ EXPR. Besides the usual basic object-oriented constructs, we include proof annotations, $e\ \mathsf{prv}\ \mathbb{r}$, for an expression e. As we will see later, such a proof annotation executes the expression e and then proves the invariants characterised by the region $\mathbb{r}$. It is crucial that our syntax is parametric with the specific area/region structure; we use different structures to model different verification techniques.

In Fig. 3, we also define runtime expressions $e_r \in$ REXPR. A runtime expression is a source expression, a value, a nested call with its stack frame σ, an exception, or a decorated runtime expression. A verification exception verfExc indicates that a proof obligation failed. A fatal exception fatalExc indicates that an expected invariant does not hold. Runtime expressions can be decorated with call e_r and ret e_r to mark the beginning and end of a method call, respectively.

In App. A (Def. 11), we define evaluation contexts, $E[\cdot]$, which describe contexts within one activation record and extend these to runtime contexts, $F[\cdot]$, which also describe nested calls.

Programming Languages. We define a programming language as a tuple consisting of a set PRG of programs, a runtime structure, and an area/region structure (see Def. 12 in App. A). Each $P \in$ PRG comes equipped with the following operations. $\mathcal{F}(c, f)$ yields the type of field f in class c as well as the class in which f is declared (c or a superclass of c). $\mathcal{M}(c, m)$ yields the type of the (single) parameter and the result type of method m in class c. $\mathcal{B}(c, m)$ yields the expression constituting the body of method m in class c as well as the class in which m is declared. Moreover, there are operators to denote subclasses and subtypes ($<:$), inclusion of areas ($\sqsubseteq$), and projection ($[\![\cdot]\!]$) of areas and regions to sets of objects and sets of object-class pairs, respectively. The projections also take an address to interpret areas and regions that are specified relatively to the current object as it is often the case in ownership systems.

We require that $\triangleright$ represents viewpoint adaptation. That is, the projection of a viewpoint-adapted region $\mathbb{a} \triangleright \mathbb{r}$ *wrt.* an address ι is equal to the union of the projections of $\mathbb{r}$ *wrt.* each object in the projection of $\mathbb{a}$:

$$[\![\mathbb{a} \triangleright \mathbb{r}]\!]_{h,\iota} \;=\; \bigcup_{\iota' \in [\![\mathbb{a}]\!]_{h,\iota}} [\![\mathbb{r}]\!]_{h,\iota'}$$

Each program also comes with typing judgements $\Gamma \vdash e : t$ and $h \vdash e_r : t$ for source and runtime expressions, respectively. An environment, $\Gamma \in$ ENV, is a tuple of the class containing the current method, the method identifier, and the type of the sole argument.

Finally, the judgement $h \models \iota, c$ expresses that in heap h, the object at address ι satisfies the invariant declared in class c. The judgement trivially holds if the object is not allocated ($\iota \notin \mathit{dom}(h)$) or is not an instance of c ($\mathit{cls}(h, \iota) \not<: c$). We say that the region $\mathbb{r}$ is *valid* in heap h *wrt.* address ι if all invariants in $[\![\mathbb{r}]\!]_{h,\iota}$ are satisfied. We denote validity of regions by $h \models \mathbb{r},\ \iota$:

$$h \models \mathbb{r},\ \iota \;\Leftrightarrow\; \forall(\iota', c) \in [\![\mathbb{r}]\!]_{h,\iota}.\;\; h \models \iota', c$$

Operational Semantics. Given a program P and a region $\mathbb{X}_{c,m}$ characterising the invariants that are expected to hold in the visible states of a method m of class c, the *runtime semantics* is the following relation defined in Fig. 4:

$$\longrightarrow\; \subseteq\; \text{REXPR} \times \text{HP} \times \text{REXPR} \times \text{HP}$$

The first seven rules are standard for imperative object-oriented languages. Note that in rNew, a new object is created using the function *new*, which takes a type as parameter rather than a class, thereby making the semantics parametric *wrt.* the type system: different type systems may use different areas, and function *new* to describe heap topological information. Similarly, through the use of *upd* and *fld* we can afford to be agnostic about the representation of a heap. Rule rCall describes method calls; it stores the class where the method body is defined in the new stack frame σ, and introduces the "marker" call e_r at the beginning of the method body.

Our reduction rules abstract away from program verification and describe only its effect. Thus, rLaunch, rLaunchExc, rFrame, and rFrameExc check whether $\mathbb{X}_{c,m}$ is valid at the beginning and end of any method execution m defined in class c, and throw a fatal exception, fatalExc, if the check fails. This represents the *visible state* semantics discussed in the introduction. Proof obligations

(rVarThis)
$$\frac{\sigma = (\iota, v, \_, \_)}{\begin{array}{c}\sigma\cdot\mathsf{this},\ h \longrightarrow \sigma\cdot\iota,\ h\\ \sigma\cdot x,\ h \longrightarrow \sigma\cdot v,\ h\end{array}}$$

(rNew)
$$\frac{\begin{array}{l}\sigma = (\iota, \_, \_, \_)\\ h', \iota' = \mathit{new}(h, \iota, t)\end{array}}{\sigma\cdot\mathsf{new}\,t,\ h \longrightarrow \sigma\cdot\iota',\ h'}$$

(rDer)
$$\frac{v = \mathit{fld}(h, \iota, f)}{\iota.f,\ h \longrightarrow v,\ h}$$

(rAss)
$$\frac{h' = \mathit{upd}(h, \iota, f, v)}{\iota.f = v,\ h \longrightarrow v,\ h'}$$

(rCall)
$$\frac{\mathcal{B}(m, \mathit{cls}(h, \iota)) = e, c \quad \sigma = (\iota, v, c, m)}{\iota.m(v),\ h \longrightarrow \sigma\cdot\mathsf{call}\,e,\ h}$$

(rCxtEval)
$$\frac{\sigma\cdot\mathrm{e}_r,\ h \longrightarrow \sigma\cdot\mathrm{e}'_r,\ h'}{\sigma\cdot E[\mathrm{e}_r],\ h \longrightarrow \sigma\cdot E[\mathrm{e}'_r],\ h'}$$

(rCxtFrame)
$$\frac{\mathrm{e}_r,\ h \longrightarrow \mathrm{e}'_r,\ h'}{\sigma\cdot\mathrm{e}_r,\ h \longrightarrow \sigma\cdot\mathrm{e}'_r,\ h'}$$

(rLaunch)
$$\frac{\begin{array}{l}\sigma = (\iota, \_, c, m)\\ h \models \mathbb{X}_{c,m},\ \iota\end{array}}{\sigma\cdot\mathsf{call}\,e,\ h \longrightarrow \sigma\cdot\mathsf{ret}\,e,\ h}$$

(rLaunchExc)
$$\frac{\begin{array}{l}\sigma = (\iota, \_, c, m)\\ h \not\models \mathbb{X}_{c,m},\ \iota\end{array}}{\sigma\cdot\mathsf{call}\,e,\ h \longrightarrow \sigma\cdot\mathsf{fatalExc},\ h}$$

(rFrame)
$$\frac{\begin{array}{l}\sigma = (\iota, \_, c, m)\\ h \models \mathbb{X}_{c,m},\ \iota\end{array}}{\sigma\cdot\mathsf{ret}\,v,\ h \longrightarrow v,\ h}$$

(rFrameExc)
$$\frac{\begin{array}{l}\sigma = (\iota, \_, c, m)\\ h \not\models \mathbb{X}_{c,m},\ \iota\end{array}}{\sigma\cdot\mathsf{ret}\,v,\ h \longrightarrow \mathsf{fatalExc},\ h}$$

(rPrf)
$$\frac{\sigma = (\iota, \_, \_, \_) \quad h \models \mathbb{r},\ \iota}{\sigma\cdot v\ \mathsf{prv}\ \mathbb{r},\ h \longrightarrow \sigma\cdot v,\ h}$$

(rPrfExc)
$$\frac{\sigma = (\iota, \_, \_, \_) \quad h \not\models \mathbb{r},\ \iota}{\sigma\cdot v\ \mathsf{prv}\ \mathbb{r},\ h \longrightarrow \sigma\cdot\mathsf{verfExc},\ h}$$

Figure 4. Reduction rules of operational semantics.

e prv $\mathbb{r}$ are verified once e reduces to a value (rPrf and rPrfExc); if $\mathbb{r}$ is not valid, a verification exception verfExc is thrown.

¡¡¡¡¡¡¡ .mine =======

A visible state invariant may be assumed to hold if it held in the previous visible state and is unaffected by the code executed since that state, or if there is a proof obligation establishing the invariant. Static verification amounts to showing all proof obligations in a program logic, based on the assumption that expected invariants hold in visible states. A verification technique is therefore sound if it does not make any false assumptions, that is, if it guarantees that fatalExc is never thrown. Soundness of a verification technique does allow verfExc to be thrown, but this will never happen in a statically verified program. ¿¿¿¿¿¿¿ .r11218

Visible state invariant satisfaction is determined from unaffected invariants that held in the previous visible state, and from proofs establishing that potentially affected invariants hold once again. Static verification amounts to showing all proof obligations in a program logic, based on the assumption that expected invariants hold in visible states. A verification technique is therefore sound if it does not make any false assumptions, that is, if it guarantees that fatalExc is never thrown. Verification technique soundness does allow verfExc to be thrown, but this will never happen in a statically verified program.

3. Verification Techniques

We now formalise verification techniques and their connection to programs, and define soundness.

A verification technique is essentially a 7-tuple, where the *components* of the tuple provide instantiations for the seven parameters of our framework. These instantiations are expressed in terms of the areas and regions provided by the programming language. To allow the instantiations to refer to the program, for instance, to look up field declarations, we define a verification technique as a mapping from programs to 7-tuples.

Definition 1. *A* verification technique $\mathcal{V}$ for a programming language *is a mapping from programs into a tuple:*

$$\mathcal{V} \ :\ \mathrm{PRG} \rightarrow \mathrm{EXP}\times\mathrm{VUL}\times\mathrm{DEP}\times\mathrm{PRE}\times\mathrm{END}\times\mathrm{UPD}\times\mathrm{CLL}$$

where

$$\begin{array}{lcl cl}
\mathbb{X} & \in & \mathrm{EXP} & = & \mathrm{CLS}\times\mathrm{MTHD}\rightarrow\mathrm{R}\\
\mathbb{V} & \in & \mathrm{VUL} & = & \mathrm{CLS}\times\mathrm{MTHD}\rightarrow\mathrm{R}\\
\mathbb{D} & \in & \mathrm{DEP} & = & \mathrm{CLS}\rightarrow\mathrm{R}\\
\mathbb{B} & \in & \mathrm{PRE} & = & \mathrm{CLS}\times\mathrm{MTHD}\times\mathrm{A}\rightarrow\mathrm{R}\\
\mathbb{E} & \in & \mathrm{END} & = & \mathrm{CLS}\times\mathrm{MTHD}\rightarrow\mathrm{R}\\
\mathbb{U} & \in & \mathrm{UPD} & = & \mathrm{CLS}\times\mathrm{MTHD}\times\mathrm{CLS}\times\mathrm{MTHD}\rightarrow\mathrm{A}\\
\mathbb{C} & \in & \mathrm{CLL} & = & \mathrm{CLS}\times\mathrm{MTHD}\times\mathrm{CLS}\rightarrow\mathrm{A}
\end{array}$$

For a verification technique applied to a program, and the application of the components to class, method names, *etc.*, we use $\mathbb{X}_{c,m}$, $\mathbb{V}_{c,m}$, $\mathbb{D}_c$, $\mathbb{B}_{c,m,\mathbb{a}}$, $\mathbb{E}_{c,m}$, $\mathbb{U}_{c,m,c'}$, $\mathbb{C}_{c,m,c',m'}$. The meaning of these components is:

$\mathbb{X}_{c,m}$**:** the region expected to be valid at the beginning and end of the body of method m in class c. The parameters c and m allow a verification technique to expect different invariants in the visible states of different methods. For instance, JML's helper methods [19, 20] do not expect any invariants to hold.

$\mathbb{V}_{c,m}$**:** the region vulnerable to method m of class c, that is, the region whose validity may be broken while control is inside m. Method m can break an invariant by updating a field or by calling a method that breaks, but does not re-establish the invariant (for instance, a helper method). The parameters c and m allow a verification technique to require that invariants of certain classes (for instance, c's subclasses) are not vulnerable.

$\mathbb{D}_c$**:** the region that depends on the fields declared in class c. The parameter c is used, for instance, to prevent invariants from depending on fields declared in c's superclasses [19, 31].

$\mathbb{B}_{c,m,\mathbb{a}}$**:** the region whose validity has to be proven before calling a method on a receiver in area $\mathbb{a}$ from the execution of a method m in class c. The parameters allow a verification technique to impose proof obligations depending on the calling method and the ownership relation between caller and callee.

$\mathbb{E}_{c,m}$**:** the region whose validity has to be proven at the end of method m in class c. The parameters allow a verification technique to require different proofs for different methods to exclude subclass invariants or helper methods.

$\mathbb{U}_{c,m,c'}$**:** the area of allowed receivers for an update of a field in class c', within the body of method m in class c. The parameters allow a verification technique, for instance, to prevent field updates within pure methods.

$\mathbb{C}_{c,m,c',m'}$**:** the area of allowed receivers for a call to method m' of class c', within the body of method m of class c. The parameters allow a technique to permit calls depending on attributes (*e.g.*, purity or effect specifications) of the caller and the callee.

Role of the Seven Components. $\mathbb{X}$ and $\mathbb{V}$ express precisely which object invariants hold at each step of program execution. The operational semantics uses $\mathbb{X}$ to describe what is expected to hold at visible states; soundness requires that $\mathbb{X} \setminus \mathbb{V}$ holds during a method activation; $\mathbb{V}$ may also affect $\mathbb{X}$ of the calling methods. Well-verification below describes proof obligations using $\mathbb{B}$ and $\mathbb{E}$ and program restrictions through $\mathbb{U}$ and $\mathbb{C}$. Finally, $\mathbb{D}$ restricts invariants. Stated otherwise, $\mathbb{U}$, $\mathbb{C}$, and $\mathbb{D}$ characterise the expressiveness of programs and invariants.

It might be initially surprising that we need as many as seven components. This number is justified by the variety of concepts used by modern verification techniques, such as accessibility of fields, purity, helper methods, ownership, and effect specifications. Note, for instance, that $\mathbb{V}$ would be redundant if all methods were (vs-null) $\dfrac{}{\Gamma \vdash_{\mathcal{V}} \mathsf{null}}$ (vs-Var) $\dfrac{}{\Gamma \vdash_{\mathcal{V}} x}$ (vs-this) $\dfrac{}{\Gamma \vdash_{\mathcal{V}} \mathsf{this}}$ (vs-new) $\dfrac{}{\Gamma \vdash_{\mathcal{V}} \mathsf{new}\, t}$ (vs-fld) $\dfrac{\Gamma \vdash_{\mathcal{V}} e}{\Gamma \vdash_{\mathcal{V}} e.f}$

(vs-ass)

$$\frac{\Gamma \vdash e : \mathbb{a}\, c' \quad \mathcal{F}(c', f) = \_, c \quad \mathbb{a} \sqsubseteq \mathbb{U}_{\Gamma,c} \quad \Gamma \vdash_{\mathcal{V}} e \quad \Gamma \vdash_{\mathcal{V}} e'}{\Gamma \vdash_{\mathcal{V}} e.f = e'}$$

(vs-call)

$$\frac{\Gamma \vdash e : \mathbb{a}\, c' \quad \mathcal{B}(c', m) = \_,\, c \quad \mathbb{a} \sqsubseteq \mathbb{C}_{\Gamma,c,m} \quad \Gamma \vdash_{\mathcal{V}} e \quad \Gamma \vdash_{\mathcal{V}} e'}{\Gamma \vdash_{\mathcal{V}} e.m(e' \,\mathsf{prv}\, \mathbb{B}_{\Gamma,\mathbb{a}})}$$

(vs-class)

$$\frac{\left.\begin{array}{l} \mathcal{B}(c,m) = e, c \\ \mathcal{M}(c,m) = t, t' \end{array}\right\} \Rightarrow \begin{cases} e = e' \,\mathsf{prv}\, \mathbb{E}_{c,m} \\ c, m, t \vdash_{\mathcal{V}} e' \end{cases}}{\vdash_{\mathcal{V}} c}$$

Figure 5. Well-Verified source expressions and classes.

to re-establish the invariants they break; in such a setting, a method could break invariants only through field updates, and $\mathbb{V}$ could be derived from $\mathbb{U}$ and $\mathbb{D}$. However, in the presence of helper methods and ownership, methods may break but not re-establish invariants.

We found the selection of components to be stable after we applied the framework to a couple of techniques. On the other hand, the signatures of the components needed adaptations, which, however did not cause any deep changes in the formalism.

Class and method identifiers can be extracted from an environment Γ or a stack frame σ in the obvious way. Thus, for $\Gamma = c, m, \_$, $\sigma = (\iota, \_, c, m)$, we use $\mathbb{X}_\Gamma$, $\mathbb{X}_\sigma$ as shorthands for $\mathbb{X}_{c,m}$; we also use $\mathbb{B}_{\Gamma,\mathbb{a}}$ and $\mathbb{B}_{\sigma,\mathbb{a}}$ as shorthands for $\mathbb{B}_{c,m,\mathbb{a}}$.

Well-Verified Programs. The judgement $\Gamma \vdash_{\mathcal{V}} e$ expresses that expression e is well-verified according to verification technique $\mathcal{V}$. The rules for this *well-verification judgement* are shown in Fig. 5.

The first five rules express that literals, variable lookup, object creation, and field read do not require proofs. The receiver of a field update must fall into $\mathbb{U}$ (vs-ass). The receiver of a call must fall into $\mathbb{C}$ (vs-call). Moreover, we require the proof of $\mathbb{B}$ before a call. Finally, a class is well-verified if the body of each of its methods is well-verified and ends with a proof obligation for $\mathbb{E}$ (vs-class). Note that we use the type judgement $\Gamma \vdash e : t$ without defining it; the definition is given by the underlying programming language, not by our framework.

A program *P* is well-verified *wrt.* $\mathcal{V}$, denoted as $\vdash_{\mathcal{V}} P$, if *(W1)* all classes are well-verified and *(W2)* all class invariants respect the dependency restrictions dictated by $\mathbb{D}$, *i.e.*, the invariant of an object ι' declared in a class c' will be preserved by an update of a field of a class c if it is not within $\mathbb{D}_c$.

Definition 2. $\vdash_{\mathcal{V}} P \;\Leftrightarrow$

$$\begin{array}{ll} \textit{(W1)} & \forall c \in P.\ \vdash_{\mathcal{V}} c \\ \textit{(W2)} & \left.\begin{array}{l} \mathcal{F}(\mathit{cls}(h,\iota), f) = \_, c \\ (\iota', c') \notin [\![\mathbb{D}_c]\!]_{h,\iota}, \\ h \models \iota', c' \end{array}\right\} \Rightarrow\ \mathit{upd}(h,\iota,f,v) \models \iota', c' \end{array}$$

Fig. 10 in App. A defines the judgement $h \vdash_{\mathcal{V}} \mathrm{e}_r$ for verified runtime expressions. Most of the rules correspond to those from Fig. 5. The others deal with values and nested calls.

Valid States. The regions $\mathbb{X}$ and $\mathbb{X} \setminus \mathbb{V}$ characterise the invariants that are expected to hold in the visible states and between visible states of the current method execution, respectively. That is, they reflect the local knowledge of the current method, but do not describe globally all the invariants that need to hold in a given state.

For any state with heap h and execution stack $\overline{\sigma}$, the function $\mathit{vi}(\overline{\sigma}, h)$ yields the set of *valid invariants*, that is, invariants that are expected to hold :

$$\mathit{vi}(\overline{\sigma}, h) = \begin{cases} \emptyset & \text{if } \overline{\sigma} = \epsilon \\ (\mathit{vi}(\overline{\sigma_1}, h) \cup [\![\mathbb{X}_\sigma]\!]_{h,\sigma}) \backslash [\![\mathbb{V}_\sigma]\!]_{h,\sigma} & \text{if } \overline{\sigma} = \overline{\sigma_1} \cdot \sigma \end{cases}$$

The call stack is empty at the beginning of program execution, at which point we expect the heap to be empty. For each additional stack frame σ, the corresponding method m establishes $\mathbb{X}_\sigma$ at the beginning of the call, and may break $\mathbb{V}_\sigma$ during the call. Therefore, we add $[\![\mathbb{X}_\sigma]\!]_{h,\sigma} \backslash [\![\mathbb{V}_\sigma]\!]_{h,\sigma}$ to the valid invariants.

A state with heap h and stack $\overline{\sigma}$ is *valid* iff:

(V1) $\overline{\sigma}$ is a valid stack, denoted by $h \vdash_{\mathcal{V}} \overline{\sigma}$ (see Def. 13 in App. A), and meaning that the receivers of consecutive method calls are within the respective $\mathbb{C}$ areas.

(V2) The valid invariants $\mathit{vi}(\overline{\sigma}, h)$ hold.

(V3) If execution is in a visible state with σ is the topmost frame of $\overline{\sigma}$, then the expected invariants $\mathbb{X}_\sigma$ hold additionally.

These properties are formalised in Def. 3. A state is determined by a heap h and a runtime expression e_r; the stack is extracted from e_r using function *stack*, see Def. 14 in App. A.

Definition 3. *A state with heap h and runtime expression* e_r *is valid for a verification technique* $\mathcal{V}$*,* $\mathrm{e}_r \models_{\mathcal{V}} h$*, iff:*

$$\begin{array}{ll} \textit{(V1)} & h \vdash_{\mathcal{V}} \mathit{stack}(\mathrm{e}_r) \\ \textit{(V2)} & h \models \mathit{vi}(\mathit{stack}(\mathrm{e}_r), h) \\ \textit{(V3)} & \mathrm{e}_r \in \{F[\sigma \cdot \mathsf{call}\, e], F[\sigma \cdot \mathsf{ret}\, v]\} \;\Rightarrow\; h \models \mathbb{X}_\sigma,\, \sigma \end{array}$$

Soundness. Intuitively, a verification technique is *sound* if verified programs only produce valid states and do not throw fatal exceptions. More precisely, a verification technique $\mathcal{V}$ is sound for a programming language *PL* iff for all well-formed and verified programs $P \in \mathit{PL}$, any well-typed and verified runtime expression e_r executed in a valid state reduces to another verified expression e'_r with a resulting valid state. Note that a verified e'_r contains no fatalExc (see Fig. 10).

Well-formedness of a program *P* is denoted by $\vdash_{\mathbf{wf}}$ *P* (see Def. 15 in App. A). Well-typedness of a runtime expression e_r is denoted by $h, \sigma \vdash \mathrm{e}_r : t$ (see Def. 12 in App. A). Type soundness is a requirement on the type system and is assumed here (see Def. 16 in App. A).

Definition 4. *A verification technique* $\mathcal{V}$ *is* sound *for a programming language PL iff for all programs P* $\in$ *PL:*

$$\left.\begin{array}{l} \vdash_{\boldsymbol{wf}} P, \;\; h, \_ \vdash \mathrm{e}_r : \_, \\ \vdash_{\mathcal{V}} P, \;\; \mathrm{e}_r \models_{\mathcal{V}} h, \;\; h \vdash_{\mathcal{V}} \mathrm{e}_r, \\ \mathrm{e}_r, h \longrightarrow \mathrm{e}'_r, h' \end{array}\right\} \;\Rightarrow\; \mathrm{e}'_r \models_{\mathcal{V}} h', \;\; h' \vdash_{\mathcal{V}} \mathrm{e}'_r$$

4. Instantiations

We now discuss how six verification techniques from the literature, which can be seen as instances of our framework. These techniques had originally been described without explicit concepts for area, region,[1] and viewpoint adaptation; heap topological aspects of the type were intertwined with the verification concerns.

In our description of the six techniques, we were able to disentangle the topological aspects from the verification concerns, and found the concepts of areas and regions to be natural abstractions.

Obviously, an "optimal" verification technique would allow maximal expressiveness of the invariants (*i.e.*, large $\mathbb{D}$), impose as few program restrictions as possible (*i.e.*, large $\mathbb{U}$ and $\mathbb{C}$), and require as few proof obligations as possible (*i.e.*, small $\mathbb{B}$ and $\mathbb{E}$). These are contradictory goals, and some trade-offs need to be struck.

[1] However, the I and E from [26] are related to our $\mathbb{X}$ and $\mathbb{V}$.

	Poetzsch-Heffter	Huizing & Kuiper	Leavens & Müller
$\mathbb{X}_c$	any	any	any
$\mathbb{V}_{c,m}$	any	$\mathsf{vul}\langle c\rangle$	$\mathsf{any\_vs}\langle c\rangle$
$\mathbb{D}_c$	any	$\mathsf{vul}\langle c\rangle$	$\mathsf{self\_vs}\langle c\rangle$
$\mathbb{B}_{c,m,\mathrm{a}}$	any	$\mathsf{vul}\langle c\rangle$	$\mathsf{any\_vs}\langle c\rangle$
$\mathbb{E}_{c,m,c'}$	any	$\mathsf{vul}\langle c\rangle$	$\mathsf{any\_vs}\langle c\rangle$
$\mathbb{U}_{c,m,c'}$	any	self	any if $\mathsf{visF}(c', c)$ emp otherwise
$\mathbb{C}_{c,m,c',m'}$	any	any	any

Figure 6. Verification techniques for unstructured heaps.

The first three techniques use information about classes to improve the tradeoff, whereas the latter three also use information about the topology of the heap. We call them *unstructured heap* and *structured heap* techniques, respectively.

4.1 Verification Techniques for Unstructured Heaps

Unstructured heap techniques use information about classes, visibility, and access paths used in definitions of invariants. The instantiations are summarised in Fig. 6.

Poetzsch-Heffter [35] devised the first verification technique that is sound for call-backs and multi-object invariants. His technique neither restricts programs nor invariants. To deal with this generality, it requires extremely strong proof obligations.

In Fig. 6, any is overloaded, and stands for the region/area of all object/object-class pairs respectively, see App. B for details. The technique requires all invariants to hold in visible states. It does not restrict invariants; $\mathbb{D}$ allows field updates to affect any invariant. $\mathbb{U}$ and $\mathbb{C}$ permit arbitrary receivers for field updates and method calls. Consequently, any invariant is vulnerable to each method. This requires proof obligations for all invariants before calls (to handle call-backs) and at the end of method bodies ($\mathbb{B}$ and $\mathbb{E}$).

Huizing and Kuiper [14] suggest a technique almost as liberal as Poetzsch-Heffter's, but impose fewer proof obligations. They achieve this by determining syntactically for each field the set of invariants that are potentially invalidated by updating the field, and imposing proof obligations only for those vulnerable invariants.

In Fig. 6, the area self denotes the current object; its use in $\mathbb{U}_{c,m,c'}$ restricts the receivers of field updates to **this**. The concept of vulnerability is captured by $\mathsf{vul}\langle c\rangle$, which is the set of all invariants (*i.e.,* object-class pairs) of the current object, as well as those of all *client* objects ι' that refer to a field f of c via a *run-time* access path. Thus, $\mathbb{D}_c = \mathsf{vul}\langle c\rangle$. Again, see details in App. B.

All invariants are expected to hold at visible states ($\mathbb{X}_c = \mathsf{any}$), and methods are allowed to modify only the receiver's fields ($\mathbb{U}_{c,m,c'} = \mathsf{self}$). Therefore, the vulnerable set ($\mathbb{V}_{c,m} = \mathsf{vul}\langle c\rangle$) and the proof obligations at the beginning/end of method calls ($\mathbb{B}_{c,m,\mathrm{a}} = \mathsf{vul}\langle c\rangle = \mathbb{E}_{c,m,c'}$) are smaller than those for Poetzsch-Heffter's technique.

Leavens & Müller [19] use visibility restrictions introduced in programming languages (*e.g.,* Java's private fields are visible only within their class), to derive information hiding for interface specifications. They allow classes to declare several invariants and to specify the visibility of these invariants, and forbid invariants depending on fields with different visibility.

Since our formalisation does not cover field visibility and assumes exactly one invariant per class, we model a special case of their technique: We assume that all fields of a class have the same visibility, and that each class declares exactly one invariant and specifies its visibility. The predicate $\mathsf{visF}(c', c)$ states whether the fields declared in class c' are visible in class c, and $\mathsf{visI}(c', c)$ states whether the invariant declared in class c' is visible in class c. A generalisation is possible but would not provide additional insights.

The technique allows field updates on arbitrary receivers as long as the field is visible in the method performing the update ($\mathbb{U}_{c,m,c'} = \mathsf{any}$ if $\mathsf{visF}(c', c)$ in Fig. 6), and does not restrict method calls ($\mathbb{C}_{c,m,c',m'} = \mathsf{any}$).

$\mathbb{D}$ allows invariants to depend only on fields of the same object declared in the same class, provided that the invariant is visible wherever the field is ($\mathsf{self\_vs}\langle c\rangle$ stands for the pair of current object and c, provided that for all classes $\mathsf{visF}(c, c') \Leftrightarrow \mathsf{visI}(c, c')$, see details in App. B.). This requirement enforces that any method that potentially breaks an invariant can see it and, thus, re-establish it.

The technique guarantees that only visible invariants are vulnerable; therefore, only visible invariants need to be proven at the beginning and end of method bodies ($\mathbb{B}_{c,m,\mathrm{a}} = \mathsf{any\_vs}\langle c\rangle = \mathbb{E}_{c,m,c'}$), where $\mathsf{any\_vs}\langle c\rangle$ stands for all object-class pairs, (ι', c'), where $\mathsf{visI}(c', c)$.) The technique supports helper methods, which we omit here for brevity (but see Sec. 6).

4.2 Comparison of Unstructured Heap Techniques

Invariant Restrictions ($\mathbb{D}$). Poetzsch-Heffter allows invariants to depend on arbitrary locations, in particular, his technique supports multi-object invariants. Huizing and Kuiper require for multi-object invariants the existence of an access path from the object containing the invariant to the object it depends on. This excludes, for instance, universal quantifications over objects. Leavens and Müller focus on invariants of single objects, and address the subclass challenge by disallowing dependencies on inherited fields.

Program Restrictions ($\mathbb{U}$ and $\mathbb{C}$). All three techniques permit arbitrary method calls. Huizing and Kuiper restrict field updates to the receiver **this**. Leavens and Müller require the updated field to be visible, a requirement enforced by the type system anyway. Thus, they are not limiting expressiveness.

Proof Obligations ($\mathbb{B}$ and $\mathbb{E}$). Poetzsch-Heffter as well as Huizing and Kuiper impose proof obligations for invariants of essentially all classes of a program (even though Huizing and Kuiper use a syntactic analysis to exclude invariants that are not vulnerable), which make them non-modular. Leavens and Müller's technique is modular as it requires proof obligations only for visible invariants.

4.3 Verification Techniques for Structured Heaps

We consider three techniques which strike a better trade-off by using the heap topology enforced by ownership types, and summarise them in Fig. 7. The rest of this section introduces the techniques and summarises their formulation in terms of our model—more in [8]. The less specialist reader may skip to section 4.4, where we compare the techniques in terms of an example.

Müller et al. [31] present two techniques for multi-object invariants, called ownership technique and visibility technique (*OT* and *VT* for short), which utilise the hierarchical heap topology enforced by Universe types [7, 30]. Universe types associate reference types with ownership modifiers, which specify ownership relative to the current object. The modifier rep expresses that an object is owned by the current object; peer expresses that an object has the same owner as the current object; any expresses that an object may have any owner.

OT and *VT* forbid rep fields f and g declared in different classes c_f and c_g, of the same object o to reference the same object. This *subclass separation* can be formalised in an ownership model where each object is owned by an object-class pair [21]. In this model, the object referenced from $o.f$ is owned by (o, c_f), whereas the object referenced from $o.g$ is owned by (o, c_g). Since they have different owners, these objects must be different.

	Müller *et al.* (*OT*)	Müller *et al.* (*VT*)	Lu *et al.*
$\mathbb{X}_{c,m}$	own ; rep$^+$	own ; rep$^+$	I ; rep*
$\mathbb{V}_{c,m}$	emp if $pure$ super$\langle c\rangle \sqcup$ own$^+$ otherwise	emp if $pure$ peer$\langle c\rangle \sqcup$ own$^+$ otherwise	E ; own*
$\mathbb{D}_{c}$	self$\langle c\rangle \sqcup$ own$^+$	peer$\langle c\rangle \sqcup$ own$^+$	self ; own*
$\mathbb{B}_{c,m,\mathbb{a}}$	super$\langle c\rangle$ if $\mathbb{a} =$ peer, $\neg pure$ emp otherwise	peer$\langle c\rangle$ if $\mathbb{a} =$ peer, $\neg pure$ emp otherwise	emp
$\mathbb{E}_{c,m}$	emp if $pure$ super$\langle c\rangle$ otherwise	emp if $pure$ peer$\langle c\rangle$ otherwise	self if I $=$ E emp otherwise
$\mathbb{U}_{c,m,c'}$	self if $\neg pure$ emp otherwise	peer if vis$(c', c), \neg pure$ emp otherwise	self if I $=$ E emp otherwise
$\mathbb{C}_{c,m,c',m'}$	emp, if $pure, \neg pure'$ rep$\langle c\rangle \sqcup$ peer otherwise	emp, if $pure, \neg pure'$ rep$\langle c\rangle \sqcup$ peer otherwise	$\bigsqcup_{\mathbb{a},\text{ with SC}(\mathsf{I},\mathsf{E},\mathsf{I}',\mathsf{E}',\mathcal{O}_{\mathbb{a},c})} \mathbb{a}$
where	$pure \equiv c::m$ is pure method $pure' \equiv c'::m'$ is pure method	$pure \equiv c::m$ is pure method $pure' \equiv c'::m'$ is pure method	I $=$ I(c, m) E $=$ E(c, m) I$' = \mathbb{a}$; I(c', m') E$' = \mathbb{a}$; E(c', m')

Figure 7. Verification techniques for structured heaps.

Areas and regions are defined as

$$\begin{array}{lll} \mathbb{a} \in \mathrm{A} & ::= & \mathsf{emp} \mid \mathsf{self} \mid \mathsf{rep}\langle c\rangle \mid \mathsf{peer} \mid \mathsf{any} \mid \mathbb{a} \sqcup \mathbb{a} \\ \mathbb{r} \in \mathrm{R} & ::= & \mathsf{emp} \mid \mathsf{self}\langle c\rangle \mid \mathsf{super}\langle c\rangle \mid \mathsf{peer}\langle c\rangle \mid \mathsf{rep} \mid \mathsf{own} \\ & \mid & \mathsf{rep}^+ \mid \mathsf{own}^+ \mid \mathbb{r}; \mathbb{r} \end{array}$$

We give their interpretation in App. B and here, an intuition for the more interesting cases: For areas, rep$\langle c\rangle$ describes all objects owned by current object and class c; peer describes all objects which share the owner object-class pair with the current object. For regions, super$\langle c\rangle$ is the set of pairs of the current object with all its superclasses, peer$\langle c\rangle$ is all the objects which share owner with the current object, provided their class is visible in c.

Ownership Technique. As shown in Fig. 7, *OT* requires that in visible states, all objects owned by the owner of **this** must satisfy their invariants ($\mathbb{X}$).

Invariants are allowed to depend on fields of the object itself (at the current class) and all its rep objects. Therefore, a field update potentially affects the invariant of the modified object and of all its (transitive) owners ($\mathbb{D}$). Dependencies on inherited fields are disallowed to address the subclass challenge.

To guarantee that pure methods are side-effect free, they must not update fields ($\mathbb{U}$) and may only call pure methods ($\mathbb{C}$). Therefore, pure methods cannot break any invariants ($\mathbb{V}$ is empty) and do not require proof obligations ($\mathbb{B}$ and $\mathbb{E}$ are empty).

A nonpure method may update fields of **this** ($\mathbb{U}$). Type correctness guarantees that the updated field is declared in the enclosing class or a superclass. Therefore, potentially affected by the update are the invariants of **this** for the enclosing class and its superclasses as well as the invariants of the (transitive) owners of **this** ($\mathbb{V}$).

OT handles multi-object invariants by allowing invariants to depend on fields of owned objects ($\mathbb{D}$); thus, methods may break the invariants of the transitive owners of **this** ($\mathbb{V}$). *E.g.,* the invariant of Client (Fig. 1) is admissible only if c is a rep field. In this case, C's method m need not preserve Client's invariant. This is reflected by the definition of $\mathbb{E}$: Only the invariants of **this** are proven at the end of the method, while those of the transitive owners may remain broken; it is the responsibility of the owners to re-establish them. *E.g.,* Client has to re-establish its invariant after a call to c.m().

Since the invariants of the owners of **this** might not hold, *OT* disallows calls on any references, as expressed by $\mathbb{C}$.

The proof obligations for method calls ($\mathbb{B}$) must cover those invariants expected by the callee that are vulnerable to the caller. This intersection contains the invariant of the caller, if caller and callee are peers, and is empty otherwise.

Visibility Technique. *VT* relaxes the restrictions of *OT* in two ways. First, it permits invariants of a class c to depend on fields of peer objects, provided that these invariants are visible in c ($\mathbb{D}$). Visibility is transitive, thus the invariant is also visible wherever fields of c are updated. Second, *VT* permits field updates on peers of **this** ($\mathbb{U}$).

These relaxations make more invariants vulnerable. Therefore, $\mathbb{V}$ includes additionally the invariants of the peers of **this**. This addition is also reflected in the proof obligations before peer calls ($\mathbb{B}$) and before the end of a non-pure method ($\mathbb{E}$).

Lu et al. Lu, Potter, and Xue define Oval, a verification technique based on ownership types [26]. Ownership types support owner parameters for classes[5], and thus a more precise description of the heap topology. The distinctive features of this technique are: (1) Expected and vulnerable invariants are described using the ownership parameters and are specific to each method. (2) Calls require no proofs. (3) Calls and method overriding require "subcontracting".

We discuss a slightly simplified version of Oval, where we omit the existential owner parameter "*", and *non-rep* fields, a refinement whereby only the current object's owners depend on such fields. Both enhance the expressiveness of the language, but are not central to our analysis. Here, we give an overview of Oval; more details are presented in our companion report [8].

In Oval, expected and vulnerable invariants are described using ¡¡¡¡¡¡¡ .mine ownership parameters and are specific to every method in a class. Every Oval program defines a function returing a contract $\langle$I, E$\rangle$ for every method defined: the expected invariants at visible states ($\mathbb{X}$) are the invariants of the object characterised by I and all objects owned by this object; the vulnerable invariants ($\mathbb{V}$) are the object at E and its transitive owners ($\mathbb{D}$). These regions are syntactically characterised by the *L*'s and *K*'s respectively, where ======= ownership parameters and are specific to each method in a class. Every Oval program defines a function returing a contract $\langle$I, E$\rangle$ for each method defined: the expected invariants at visible states ($\mathbb{X}$) are the invariants of the object characterised by I and all objects transitively owned by this object; the vulnerable invariants ($\mathbb{V}$) are the object at E and its transitive owners ($\mathbb{D}$). These regions are syntactically characterised by the *L*'s and *K*'s respectively, where

¿¿¿¿¿¿¿ .r11218 $\quad L ::=$ top | bot | this | $X \quad K ::= \; L \mid K$; rep

and where X stands for the class's owner parameters.

To adapt Oval to our framework, we define areas and regions:

$$\begin{array}{lll} \mathbb{a} \in \mathrm{A} & ::= & \mathsf{emp} \mid \mathsf{self} \mid c\langle \overline{K}\rangle \mid \mathbb{a} \sqcup \mathbb{a} \\ \mathbb{r} \in \mathrm{R} & ::= & \mathsf{emp} \mid \mathsf{self} \mid K \mid K; \mathsf{rep}^* \mid K; \mathsf{own}^* \end{array}$$

	Müller *et al.* (*OT*)	Müller *et al.* (*VT*)	Lu *et al.*
1. $\llbracket\mathbb{X}_{\mathsf{C,m}}\rrbracket_{h_0,3}$	$\{$ (4, D), (4, D'), (3, C), (3, C'), (5, E), (5, E') $\}$	$\{$ (4, D), (4, D'), (3, C), (3, C'), (5, E), (5, E') $\}$	$\{$ (3, C), (3, C'), (5, E), (5, E') $\}$
2. $\llbracket\mathbb{V}_{\mathsf{C,m}}\rrbracket_{h_0,3}$	$\{$ (3, C), (2, B), (1, A') $\}$	$\{$ (3, C), (2, B), (1, A'), (4, D) $\}$	$\{$ (2, B), (2, B'), (1, A), (1, A') $\}$
3. $\llbracket\mathbb{D}_{\mathsf{C}}\rrbracket_{h_0,3}$	$\{$ (3, C), (2, B), (1, A') $\}$	$\{$ (3, C), (2, B), (1, A'), (4, D) $\}$	$\{$ (3, C), (3, C'), (2, B), (2, B'), (1, A), (1, A') $\}$
4. $\llbracket\mathbb{B}_{\mathsf{C,m},\mathbb{a}}\rrbracket_{h_0,3}$	$\emptyset$ if $\mathbb{a} = \mathsf{rep}\langle\mathsf{C}\rangle$ $\{$ (3, C) $\}$ if $\mathbb{a} = \mathsf{peer}$	$\emptyset$ if $\mathbb{a} = \mathsf{rep}\langle\mathsf{C}\rangle$ $\{$ (3, C), (4, D) $\}$ if $\mathbb{a} = \mathsf{peer}$	$\emptyset$
5a. $\llbracket\mathbb{E}_{\mathsf{C,m}}\rrbracket_{h_0,3}$	$\{$ (3, C) $\}$	$\{$ (3, C), (4, D) $\}$	$\emptyset$
5b. $\llbracket\mathbb{E}_{\mathsf{C,m1}}\rrbracket_{h_0,3}$	$\{$ (3, C) $\}$	$\{$ (3, C), (4, D) $\}$	$\{$ (3, C), (3,C') $\}$
6a. $\llbracket\mathbb{U}_{\mathsf{C,m,Objct}}\rrbracket_{h_0,3}$	$\{ 3 \}$	$\{ 3, 4 \}$	$\emptyset$
6b. $\llbracket\mathbb{U}_{\mathsf{C,m1,Objct}}\rrbracket_{h_0,3}$	$\{ 3 \}$	$\{ 3, 4 \}$	$\{ 3 \}$
7. $\llbracket\mathbb{C}_{\mathsf{C,m,Objct,m2}}\rrbracket_{h_0,3}$	$\{ 3, 4, 5 \}$	$\{ 3, 4, 5 \}$	$\{ 1, 2, 3, 4, 5, 6 \}$
assuming that	C::m not pure $ownr(h_0, 5) = 3, \mathsf{C}'$, $ownr(h_0, 3) = 2, \mathsf{B}$, $ownr(h_0, 4) = 2, \mathsf{B}'$, $ownr(h_0, 2) = 1, \mathsf{A}$	C::m not pure $ownr(h_0, 5) = 3, \mathsf{C}'$, $ownr(h_0, 3) = 2, \mathsf{B}$, $ownr(h_0, 4) = 2, \mathsf{B}'$, $ownr(h_0, 2) = 1, \mathsf{A}$ $\mathsf{vis(C,\ D)}, \neg\mathsf{vis(C, D')}$	$\mathsf{I(C, m) = this}$ $\mathsf{E(C, m) = X}$, and X maps to 2 $\mathsf{I(C, m1) = E(C, m1) = this}$ $\mathsf{I(Obj, m2) = bot}$ $\mathsf{E(Obj, m2) = top}$

Figure 8. Comparison of techniques for structured heaps; differences are highlighted in grey.

and give their interpretation in App. B.

Methods expect the invariants of the object characterised by I and all objects owned by this object ($\mathbb{X}$ in Fig. 7).

Oval requires that the expected and vulnerable invariants of every method are disjoint or intersect at this. Consequently, at the end of a method, one has to prove the invariant of the current receiver, if I and E are equal, and nothing otherwise ($\mathbb{E}$). In the former case, the method is allowed to update fields of its receiver; no updates are allowed otherwise ($\mathbb{U}$).

Oval does not impose proof obligations on method calls ($\mathbb{B}$ is empty). Call-backs are handled via *subcontracting*, which is defined using the order $L \preceq L'$, which guarantees that at runtime the object denoted by L will be transitively owned by the object denoted by L'.[2] Oval's subcontracting is adapted here to $\mathsf{SC}(\mathsf{I}, \mathsf{E}, \mathsf{I}', \mathsf{E}', K)$, which holds iff:

$$\begin{array}{ll} \mathsf{I} \prec \mathsf{E} \ \Rightarrow\ \mathsf{I}' \preceq \mathsf{I} & \mathsf{I} = \mathsf{E} \ \Rightarrow\ \mathsf{I}' \prec \mathsf{I} \\ \mathsf{I}' \prec \mathsf{E}' \ \Rightarrow\ \mathsf{E} \preceq \mathsf{E}' & \mathsf{I} = \mathsf{E} = \mathsf{this} \ \Rightarrow\ \mathsf{E} \preceq K \end{array}$$

where I, E characterise the caller, I', E' characterise the callee, and K stands for the owner of the callee. Subcontracting is used to restrict possible method calls ($\mathbb{C}$), where $\mathcal{O}$ (defined in App. B) extracts the owner of $\mathbb{a}$.

Oval also requires $\mathsf{SC}(\mathsf{I}, \mathsf{E}, \mathsf{I}', \mathsf{E}', K)$ to hold between a method and a method overriding it. As we discuss later, this is too weak to guarantee soundness [25], and we have found a counterexample [8, 25]. Therefore, we use the following stronger requirement for methods m of a class c and its subclass c':

$$\mathsf{I}(c', m) \preceq \mathsf{I}(c, m) \preceq \mathsf{E}(c, m) \preceq \mathsf{E}(c', m)$$

which guarantees *(S5)* from Def. 5 in the next section. We refer to the verification technique with this stronger requirement as Oval'.

4.4 Comparison of Structured Heaps Techniques

We compare the techniques for structured heaps using the heap h_0 in Fig. 9, and show in Fig. 8 the values of the components of the three techniques for object 3 and class C. In the last row, we show the class-object pairs which own further objects (information required by OT and VT), and the I and E (information required by Oval).

Invariant Restrictions ($\mathbb{D}$). Both *OT* and Oval support multi-object invariants by permitting the invariant of an object *o* to depend on fields of *o* and of objects (transitively) owned by *o*. However, *OT* requires that fields of *o* are declared in the same class as the invariant to address the subclass challenge. For instance, $\mathbb{D}$ for *OT* does not include (3,C'), whereas $\mathbb{D}$ for Oval does.

In addition, *VT* allows dependencies on peers (therefore, $\mathbb{D}$ includes (4,D)) and thus can handle multi-object structures that are not organised hierarchically.

Program Restrictions ($\mathbb{U}$ and $\mathbb{C}$). In *OT* and Oval, an object may only modify its own fields, whereas *VT* also allows modifications of peers; thus, object 4 is part of $\mathbb{U}$ for *VT*. In Oval, an object may only modify its own fields if the I, E annotations are this; this is why $\mathbb{U}$ is empty for m but contains 3 for m1.

Method calls in *OT* and *VT* are restricted to the peers and reps of an object; thus, a call on a rep object *o* cannot call back into one of *o*'s (transitive) owners, whose invariants might not hold.

In Oval, the receiver of a method call may be *anywhere* within the owners of the current receiver, provided that the I and E annotations of the called method satisfy the subcontract requirement. Therefore, $\mathbb{C}$ for Oval includes for instance object 2, which is not permitted in *OT* and *VT*.

Proof Obligations ($\mathbb{B}$ and $\mathbb{E}$). Since *OT* uses rather restricted invariants, it has a small vulnerable set $\mathbb{V}$ and, thus, few proof obligations. The dependencies on peers permitted by *VT* lead to a larger vulnerable set and more proof obligations. For instance, (4,D) is part of the vulnerable set $\mathbb{V}$ (because executions on 3 might break 4's D-invariant). Hence, of the proof obligations $\mathbb{B}$ and $\mathbb{E}$.

Oval imposes end-of-body proof obligation only when I and E are the same (*i.e.*, m1). Since it permits invariants to depend on inherited fields, it requires proof obligations for subclass invariants. For instance, (3,C') is part of $\mathbb{E}$ for m1, which means that verification of a class requires knowledge of a subclass[3]. *OT* and

[2] The relation is extended for *K* in the obvious way [26].

[3] The Oval developers plan to solve this modularity problem by requiring that any inherited method has to be re-verified in the subclass [25].

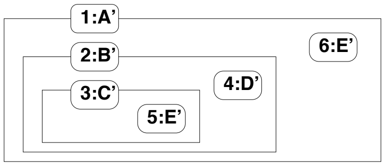


Figure 9. Heap h_0, with objects at addresses 1–6 belonging to indicated classes. Objects atop a box own those inside it. We assume that A' is a subclass of A and analogously for the other classes.

VT avoid this problem by forbidding such dependencies; thus their proof obligations do not include (3,C').

4.5 Conclusions

Exact knowledge of which class's invariant is broken or has to be validated is crucial in ensuring modularity; this was a central point in the works of Huizing & Kuipper, Leavens & Müller, and Müller et al.

All techniques except Poetzsch-Heffter enforce restrictions on which objects may be updated ($\mathbb{U}$) or receive calls ($\mathbb{C}$), and thus reduce the number of objects whose invariant need to be verified ($\mathbb{B}$ and $\mathbb{E}$). Structured heaps techniques refine this not only on the basis of the class of objects, but also on the basis of their location.

Oval's more powerful type system is more precise about the location of objects, and allows methods to declare individually their requirements (E) and their effects (I). The method's effects together with the location of the callee allow Oval to be flexible about further method calls ($\mathbb{C}$). On the other hand, Oval is, in our view, very restrictive as to the dependencies of invariants ($\mathbb{D}$); thus *e.g.*, it cannot describe the subject-observer pattern. *VT* is most liberal in that respect, as it allows dependencies on peers.

Perhaps the most liberal approach would generalise Oval, and allow dependencies to be declared per class; this would require sophisticated analysis to ensure that proof obligations are sufficient.

5. Well-Structured Verification Techniques

We now identify conditions on the components of a verification technique that are sufficient for soundness, state a general soundness theorem, and discuss soundness of the techniques presented in Sec. 4.

Definition 5. *A verification technique is* well-structured *if, for all programs in the programming language:*

$$\begin{array}{ll}
\textit{(S1)} & \mathbb{a} \sqsubseteq \mathbb{C}_{c,m,c'm'} \Rightarrow (\mathbb{a} \triangleright \mathbb{X}_{c',m'}) \setminus (\mathbb{X}_{c,m} \setminus \mathbb{V}_{c,m}) \ \subseteq \ \mathbb{B}_{c,m,\mathbb{a}} \\
\textit{(S2)} & \mathbb{V}_{c,m} \cap \mathbb{X}_{c,m} \subseteq \mathbb{E}_{c,m} \\
\textit{(S3)} & \mathbb{C}_{c,m,c',m'} \triangleright (\mathbb{V}_{c',m'} \setminus \mathbb{X}_{c',m'}) \ \subseteq \ \mathbb{V}_{c,m} \\
\textit{(S4)} & \mathbb{U}_{c,m,c'} \triangleright \mathbb{D}_{c'} \ \subseteq \ \mathbb{V}_{c,m} \\
\textit{(S5)} & c <: c' \ \Rightarrow \begin{cases} \mathbb{X}_{c,m} \subseteq \mathbb{X}_{c',m}, \\ \mathbb{V}_{c,m} \backslash \mathbb{X}_{c,m} \subseteq \mathbb{V}_{c',m} \setminus \mathbb{X}_{c',m} \end{cases}
\end{array}$$

In the above, the set theoretic symbols have the obvious interpretation in the domain of regions. For example *(S2)* is short for $\forall h, \iota \ : \ [\![\mathbb{V}_{c,m}]\!]_{h,\iota} \cap ([\![\mathbb{X}_c]\!]_{h,\iota} \subseteq [\![\mathbb{E}_{c,m}]\!]_{h,\iota}$.

The first two conditions relate proof obligations with expected invariants. *(S1)* ensures for a call within the permitted area that the expected invariants of the callee ($\mathbb{a} \triangleright \mathbb{X}_{c',m'}$) minus the invariants that hold throughout the calling method ($\mathbb{X}_{c,m} \setminus \mathbb{V}_{c,m}$) are included in the proof obligation for the call ($\mathbb{B}_{c,m,\mathbb{a}}$). *(S2)* ensures that the invariants that were broken during the execution of a method, but which are required to hold again at the end of the method ($\mathbb{V}_{c,m} \cap \mathbb{X}_{c,m}$), are included in the proof obligation at the end of the method ($\mathbb{E}_{c,m}$).

The third and fourth condition ensure that invariants that are broken by a method m of class c are actually in its vulnerable set. Condition *(S3)* deals with calls and therefore uses viewpoint adaptation for call regions ($\mathbb{C}_{c,m,c',m'} \triangleright \ldots$). It restricts the invariants that may be broken by the callee method m', but are not re-established by the callee through $\mathbb{E}$. These invariants must be included in the vulnerable invariants of the caller. Condition *(S4)* ensures for field updates within the permitted area that the invariants broken by updating a field of class c' are included in the vulnerable invariants of the enclosing method, m.

Finally, *(S5)* establishes conditions for subclasses. An overriding method m in a subclass c may expect fewer invariants than the overridden m in superclass c'. Moreover, the subclass method must leave less invariants broken than the superclass method.

Soundness Results. The five conditions from Def. 5 guarantee Verification Technique soundness (Def. 4), provided that the programming language has a sound type system (see Def. 16 in App. A).

Theorem 6. *A well-structured verification technique, built on top of a programming language with a sound type system, is sound.*

This theorem is one of the main results of our work. It reduces the complex task of proving verification technique soundness to checking five fairly simple conditions.

Theorem 7. *The verification techniques by Poetzsch-Heffter, by Huizing & Kuiper, by Leavens & Müller, by Müller* et al. *(OT), by Müller* et al. *(VT), and Oval' are well-structured.*

Corollary 8. *The verification techniques by Poetzsch-Heffter, by Huizing & Kuiper, by Leavens & Müller, by Müller* et al. *(OT), by Müller* et al. *(VT), and Oval' are sound.*

We relegate proofs of the theorems to the companion report [8].

Soundness of Oval. The original Oval proposal as in [26] is unsound because it requires subcontracting for method overriding, which gives, in our terminology $\mathbb{V}_{c,m} \backslash \mathbb{X}_{c,m} \ \sqsubseteq \ \mathbb{V}_{c',m}$. This is clearly weaker than what we require in *(S5)*; this alerted us, and using the $\mathbb{X}$ and $\mathbb{V}$ components (no type system properties, nor any other component) we constructed a counterexample showing the unsoundness (*cf.* [8]), which we communicated to the authors [25], who confirmed our findings.

Soundness of the remaining techniques. Our formal proof confirmed soundness for the verification techniques by Poetzsch-Heffter, by Huizing & Kuiper, by Leavens & Müller, by Müller *et al.* (*OT*), by Müller *et al.* (*VT*). Nevertheless, we found that the semi-formal arguments supporting the original soundness claims at times missed crucial steps. For instance, the soundness proofs for *OT* and *VT* [31] do not mention any condition relating to *(S3)* of Def. 5; in our formal proof, *(S3)* was vital to determine what invariants still hold after a method returns (see [8] for details).

6. Related Work

In this section, we discuss related work other than the verification techniques covered in Sec. 4.

The idea of areas and regions is inspired from type and effects systems [37], which have been extremely widely applied, *e.g.*, to support race-free programs and atomicity [10].

Object invariants trace back to Hoare's implementation invariants [12] and monitor invariants [13]. They were popularised in object-oriented programming by Meyer [27]. Their work, as well as other early work on object invariants [23, 24] did not address the three challenges described in the introduction. Since they were not formalised, it is difficult to understand the exact requirements and soundness arguments (see [31] for a detailed discussion). However, once the requirements are clear, a formalisation within our framework seems straightforward.

The verification techniques based on the Boogie methodology [1, 3, 21, 22] do not use a visible state semantics. Instead, each method specifies in its precondition which invariants it requires. Extending our framework to Spec# requires two changes. First, even though Spec# permits methods to specify explicitly which invariants they require, the default is to require the invariants of its arguments and all their peer objects. These defaults can be modelled in our framework by allowing method-specific regions $\mathbb{X}$. Second, Spec# checks invariants at the end of expose blocks instead of the end of method bodies. Expose blocks can easily be added to our formalism.

We only know of one verification technique based on visible states, which cannot be expressed in our framework. The work by Middelkoop *et al.* [29] uses proof obligations that refer to the heap of the pre-state of a method execution. To formalise this technique, we have to generalise our proof obligations to take two invariant-regions, one for the pre-state heap and one for the post-state heap. Since this generality is not needed for any of the other techniques, we omitted a formal treatment in this paper.

Some verification techniques exclude the pre- and post-states of so-called helper methods from the visible states [19, 20]. Helper methods can easily be expressed in our framework by choosing different parameters for helper and non-helper methods. For instance in JML, $\mathbb{X}$, $\mathbb{B}$, and $\mathbb{E}$ are empty for helper methods, because they neither assume nor have to preserve any invariants.

Once established, strong invariants [11] hold throughout program execution. They are especially useful to reason about concurrency and security properties. Our framework can model strong invariants, essentially by preventing them from occurring in $\mathbb{V}$.

Existing techniques for visible state invariants have only limited support for object initialisation. Constructors are prevented from calling methods because the callee method in general requires all invariants to hold, but the invariant of the new object is not yet established. Fähndrich and Xia developed delayed types [9] to control call-backs into objects that are being initialised. Delayed types support strong invariants. Modelling delayed types in our framework is future work.

Even though separation logic [15, 36] has been used to reason about invariants of modules with one instance [32], object invariants are not as important as in other verification techniques. Instead, verifiers are encouraged to write predicates to express consistency criteria [33]. Abstract predicate families [34] allow one to do so without violating abstraction and information hiding. The general predicates of separation logic provide more flexibility than can be expressed by our framework.

7. Conclusions

We presented a framework that describes verification techniques for object invariants in terms of seven parameters, and separates verification concerns from those of the underlying type system. Our formalism is parametric *wrt.* the type system of the programming language, the regions used to describe assumptions and proof obligations, and the meaning of validity of an invariant. We illustrated the generality of our framework by instantiating it to describe six existing verification techniques. We identified sufficient conditions on the framework parameters that guarantee soundness and we proved a universal soundness theorem.

A unified framework with the above separation of concerns, offers, in our opinion, three important advantages. First, it allows a simpler understanding of the verification concerns. Second, it facilitates comparisons. Third, we found checking the soundness conditions significantly simpler than developing soundness proofs from scratch.

We hope that our framework will be used in future development of further verification techniques.

Acknowledgments

We are grateful for feedback from Rustan Leino, Matthew Parkinson, Ronald Middelkoop, and the anonymous POPL and FOOL referees.

This work was funded in part by the Information Society Technologies program of the European Commission, Future and Emerging Technologies under the IST-2005-015905 MOBIUS project. Müller's work was partly carried out at ETH Zurich.

References

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

[2] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart devices (CASSIS)*, LNCS, pages 49–69. Springer-Verlag, 2005.

[3] M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared State. In D. Kozen, editor, *Mathematics of Program Construction*, volume 3125 of *LNCS*, pages 54–84. Springer-Verlag, 2004.

[4] Y. Cheon and G. T. Leavens. The Larch/Smalltalk interface specification language. *ACM Transactions on Software Engineering and Methodology*, 3(3):221–253, 1994.

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

[6] W. Dietl, S. Drossopoulou, and P. Müller. Generic Universe Types. In E. Ernst, editor, *European Conference on Object-Oriented Programming (ECOOP)*, LNCS. Springer-Verlag, 2007. To appear.

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

[8] S. Drossopoulou, A. Francalanza, and P. Müller. A unified framework for verification techniques for object invariants — full paper. Available from `research.microsoft.com/~mueller/publications.html`, 2007.

[9] M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *Object-oriented programming, systems, languages, and applications (OOPSLA)*. ACM Press, 2007. To appear.

[10] Cormac Flanaggan and Shaz Qadeer. A Type and Effect System for Atomicity. In *PLDI'03*, 2003.

[11] R. Hähnle and W. Mostowski. Verification of safety properties in the presence of transactions. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices (CASSIS)*, volume 3362 of *LNCS*, pages 151–171. Springer-Verlag, 2005.

[12] C. A. R. Hoare. Proofs of correctness of data representation. *Acta Informatica*, 1:271–281, 1972.

[13] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[14] K. Huizing and R. Kuiper. Verification of object-oriented programs using class invariants. In T. Maibaum, editor, *Fundamental Approaches to Software Engineering (FASE)*, volume 1783 of *LNCS*, pages 208–221. Springer-Verlag, 2000.

[15] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *Principles of programming languages (POPL)*, pages 14–26. ACM Press, 2001.

[16] K. D. Jones. LM3: A Larch interface language for Modula-3: A Definition and introduction. Technical Report 72, Digital Equipment

Corporation, Systems Research Center, 1991.

[17] G. T. Leavens. An overview of Larch/C++: Behavioral specifications for C++ modules. In H. Kilov and W. Harvey, editors, *Specification of Behavioral Semantics in Object-Oriented Information Modeling*, chapter 8, pages 121–142. Kluwer Academic Publishers, 1996.

[18] G. T. Leavens, A. L. Baker, and C. Ruby. Preliminary design of JML: a behavioral interface specification language for Java. *SIGSOFT Softw. Eng. Notes*, 31(3):1–38, 2006.

[19] G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *International Conference on Software Engineering (ICSE)*, pages 385–395. IEEE, 2007.

[20] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. R. Cok, P. Müller, J. Kiniry, and P. Chalin. JML Reference Manual. Department of Computer Science, Iowa State University. Available from `http://www.jmlspecs.org`, February 2007.

[21] K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In M. Odersky, editor, *European Conference on Object-Oriented Programming (ECOOP)*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.

[22] K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In R. de Nicola, editor, *European Symposium on Programming (ESOP)*, volume 4421 of *LNCS*, pages 316–330. Springer-Verlag, 2007.

[23] B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[24] B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[25] Y. Lu and J. Potter. Soundness of Oval. Priv. Commun., June 2007.

[26] Y. Lu, J. Potter, and J. Xue. Object Invariants and Effects. In *European Conference on Object-Oriented Programming (ECOOP)*, LNCS. Springer-Verlag, 2007. To appear.

[27] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[28] B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[29] R. Middelkoop, C. Huizing, R. Kuiper, and E. J. Luit. Invariants for non-hierarchical object structures. In L. Ribeiro and A. Martins Moreira, editors, *Brazilian Symposium on Formal Methods (SBMF)*, To appear.

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

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

[32] P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and information hiding. In *Principles of programming languages (POPL)*, pages 268–280. ACM Press, 2004.

[33] M. Parkinson. Class invariants: the end of the road? In *International Workshop on Aliasing, Confinement and Ownership*, 2007.

[34] M. Parkinson and G. Bierman. Separation logic and abstraction. In *Principles of programming languages (POPL)*, pages 247–258. ACM Press, 2005.

[35] A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.

[36] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, pages 55–74. IEEE Computer Society, 2002.

[37] J. P. Talpin and P.Jouvelot. The Type and Effect Discipline. In *LICS'92*, 1992.

A. Appendix - The Framework

Definition 9. *A runtime structure is a tuple*

$$\textsc{RStruct} \;=\; (\textsc{Hp}, \textsc{Adr}, \simeq, \preceq, \mathit{dom}, \mathit{cls}, \mathit{fld}, \mathit{upd}, \mathit{new})$$

where $\textsc{Hp}$*, and* $\textsc{Adr}$ *are sets, and where*

$$\begin{array}{lll}
\simeq & \subseteq\ \textsc{Hp} \times \textsc{Hp} & \qquad \preceq\ \subseteq\ \textsc{Hp} \times \textsc{Hp} \\
\mathit{dom} & :\ \textsc{Hp} \longrightarrow \mathcal{P}(\textsc{Adr}) & \\
\mathit{cls} & :\ \textsc{Hp} \times \textsc{Adr} \rightharpoonup \textsc{Cls} & \\
\mathit{fld} & :\ \textsc{Hp} \times \textsc{Adr} \times \textsc{Fld} \rightharpoonup \textsc{Val} & \\
\mathit{upd} & :\ \textsc{Hp} \times \textsc{Adr} \times \textsc{Fld} \times \textsc{Val} \longrightarrow \textsc{Hp} & \\
\mathit{new} & :\ \textsc{Hp} \times \textsc{Adr} \times \textsc{Typ} \longrightarrow \textsc{Hp} \times \textsc{Adr} &
\end{array}$$

where $\textsc{Val} = \textsc{Adr} \cup \{\mathsf{null}\}$ *for some element* $\mathsf{null} \notin \textsc{Adr}$. *For all* $h \in \textsc{Hp},\ \iota, \iota' \in \textsc{Adr},\ v \in \textsc{Val}$*, we require:*

$$\text{(H1)} \qquad \iota \in \mathit{dom}(h) \Rightarrow \exists c.\mathit{cls}(h, \iota) = c$$

$$\text{(H2)} \qquad h \simeq h' \Rightarrow \begin{cases} \mathit{dom}(h) = \mathit{dom}(h'), \\ \mathit{cls}(h, \iota) = \mathit{cls}(h', \iota) \end{cases}$$

$$\text{(H3)} \qquad h \preceq h' \Rightarrow \begin{cases} \mathit{dom}(h) \subseteq \mathit{dom}(h'), \\ \forall \iota \in \mathit{dom}(h). \\ \quad \mathit{cls}(h, \iota) = \mathit{cls}(h', \iota) \end{cases}$$

$$\text{(H4)} \qquad \mathit{upd}(h, \iota, f, v) = h' \Rightarrow \begin{cases} h \simeq h' \quad \mathit{fld}(h', \iota, f) = v, \\ \iota \neq \iota' \text{ or } f \neq f' \Rightarrow \\ \quad \mathit{fld}(h', \iota', f') = \mathit{fld}(h, \iota', f') \end{cases}$$

$$\text{(H5)} \qquad \mathit{new}(h, \iota, t) = h', \iota' \Rightarrow \begin{cases} h \preceq h' \\ \iota' \in \mathit{dom}(h') \backslash \mathit{dom}(h) \end{cases}$$

Definition 10. *An* area/region structure *is a tuple*

$$\textsc{AStruct} \;=\; (\mathrm{A}, \mathrm{R}, \triangleright)$$

where A *and* R *are sets, and* $\triangleright$ *is an operation with signature:*

$$\triangleright \;:\; \mathrm{A} \times \mathrm{R} \rightarrow \mathrm{R}$$

Definition 11. $E[\cdot]$ *and* $F[\cdot]$ *are defined as follows:*

$$\begin{array}{lcl}
E[\cdot] & ::= & [\cdot] \mid E[\cdot].f \mid E[\cdot].f = e \mid \iota.f = E[\cdot] \mid E[\cdot].m(e) \\
 & \mid & \iota.m(E[\cdot]) \mid E[\cdot]\, \mathsf{prv}\, \mathbb{r} \mid \mathsf{ret}\, E[\cdot] \\
F[\cdot] & ::= & [\cdot] \mid F[\cdot].f \mid F[\cdot].f = e \mid \iota.f = F[\cdot] \mid F[\cdot].m(e) \\
 & \mid & \iota.m(F[\cdot]) \mid F[\cdot]\, \mathsf{prv}\, \mathbb{r} \mid \sigma \cdot F[\cdot] \mid \mathsf{call}\, F[\cdot] \mid \mathsf{ret}\, F[\cdot]
\end{array}$$

Definition 12. *A programming language is a tuple*

$$\mathit{PL} \;=\; (\textsc{Prg}, \textsc{RStruct}, \textsc{AStruct})$$

where $\textsc{Prg}$ *is a set where every* $P \in \textsc{Prg}$ *is a tuple*

$$P \;=\; \left(\begin{array}{ll} \mathcal{F}, \mathcal{M}, \mathcal{B}, <: & \textit{(class definitions)} \\ \sqsubseteq, [\![\cdot]\!] & \textit{(inclusion and projections)} \\ \models, \vdash & \textit{(invariant and type satisfaction)} \end{array} \right)$$

with signatures:

$$\begin{array}{lll}
\mathcal{F} & : & \textsc{Cls} \times \textsc{Fld} \rightharpoonup \textsc{Typ} \times \textsc{Cls} \\
\mathcal{M} & : & \textsc{Cls} \times \textsc{Mthd} \rightharpoonup \textsc{Typ} \times \textsc{Typ} \\
\mathcal{B} & : & \textsc{Cls} \times \textsc{Mthd} \rightharpoonup \textsc{Expr} \times \textsc{Cls} \\
<: & \subseteq & \textsc{Cls} \times \textsc{Cls} \cup \textsc{Typ} \times \textsc{Typ} \\
\sqsubseteq & \subseteq & \mathrm{A} \times \mathrm{A} \\
[\![\cdot]\!] & : & \mathrm{A} \times \textsc{Hp} \times \textsc{Adr} \rightarrow \mathcal{P}(\textsc{Adr}) \\
[\![\cdot]\!] & : & \mathrm{R} \times \textsc{Hp} \times \textsc{Adr} \rightarrow \mathcal{P}(\textsc{Adr} \times \textsc{Cls}) \\
\models & \subseteq & \textsc{Hp} \times \textsc{Adr} \times \textsc{Cls} \\
\vdash & \subseteq & (\textsc{Env} \times \textsc{Expr} \cup \textsc{Hp} \times \textsc{RExpr}) \times \textsc{Typ}
\end{array}$$

where every $P \in \textsc{Prg}$ *must satisfy the constraints:*

$$\begin{array}{lrl}
\text{(P1)} & \mathcal{F}(c, f) = t, c' & \Rightarrow c <: c' \\
\text{(P2)} & \mathcal{B}(c, m) = e, c' & \Rightarrow c <: c' \\
\text{(P3)} & \mathcal{F}(\mathit{cls}(h, \iota), f) = t, \_ & \Rightarrow \exists v.\mathit{fld}(h, \iota, f) = v \\
\text{(P4)} & \mathbb{a}_1 \sqsubseteq \mathbb{a}_2 & \Rightarrow [\![\mathbb{a}_1]\!]_{h,\iota} \subseteq [\![\mathbb{a}_2]\!]_{h,\iota} \\
\text{(P5)} & [\![\mathbb{a} \triangleright \mathbb{r}]\!]_{h,\iota} & = \bigcup_{\iota' \in [\![\mathbb{a}]\!]_{h,\iota}} [\![\mathbb{r}]\!]_{h,\iota'} \\
\text{(P6)} & [\![\mathbb{a}]\!]_{h,\iota} & \subseteq \mathit{dom}(h) \\
\text{(P7)} & h \preceq h' & \Rightarrow [\![\mathbb{r}]\!]_{h,\iota} \subseteq [\![\mathbb{r}]\!]_{h',\iota} \\
\text{(P8)} & \mathbb{a}\, c <: \mathbb{a}'\, c' & \Rightarrow \mathbb{a} \sqsubseteq \mathbb{a}',\ c <: c'
\end{array}$$

Definition 13. *Stack* $\overline{\sigma}$ *is* valid *wrt. heap* h *in a verification technique* $\mathcal{V}$*, denoted by* $h \vdash_{\mathcal{V}} \overline{\sigma}$*, iff:*

(ad-null)
$$\frac{}{h \vdash_{\mathcal{V}} \sigma\cdot\textsf{null}}$$

(ad-addr)
$$\frac{\iota \in \mathit{dom}(h)}{h \vdash_{\mathcal{V}} \sigma\cdot\iota}$$

(ad-new)
$$\frac{}{h \vdash_{\mathcal{V}} \sigma\cdot\textsf{new}\, t}$$

(ad-Var)
$$\frac{}{h \vdash_{\mathcal{V}} \sigma\cdot x}$$

(ad-this)
$$\frac{}{h \vdash_{\mathcal{V}} \sigma\cdot\textsf{this}}$$

(ad-verEx)
$$\frac{}{h \vdash_{\mathcal{V}} F[\textsf{verfExc}]}$$

(ad-ass)
$$\frac{\begin{array}{l} h,\sigma \vdash \mathrm{e}_r : \mathbb{a}\, c' \\ \mathcal{F}(c', f) = \_, c \\ \mathbb{a} \sqsubseteq \mathbb{U}_{\sigma,c} \\ h \vdash_{\mathcal{V}} \sigma\cdot\mathrm{e}_r \\ h \vdash_{\mathcal{V}} \sigma\cdot\mathrm{e}'_r \end{array}}{h \vdash_{\mathcal{V}} \sigma\cdot\mathrm{e}_r.f = \mathrm{e}'_r}$$

(ad-fld)
$$\frac{h \vdash_{\mathcal{V}} \sigma\cdot\mathrm{e}_r}{h \vdash_{\mathcal{V}} \sigma\cdot\mathrm{e}_r.f}$$

(ad-end)
$$\frac{h \vdash_{\mathcal{V}} \sigma'\cdot v}{h \vdash_{\mathcal{V}} \sigma\cdot\sigma'\cdot\textsf{ret}\, v}$$

(ad-call)
$$\frac{\begin{array}{l} h,\sigma \vdash \mathrm{e}_r : \mathbb{a}\, c' \\ \mathcal{B}(c', m) = \_,\ c \\ \mathbb{a} \sqsubseteq \mathbb{C}_{\sigma,c,m} \\ h \vdash_{\mathcal{V}} \sigma\cdot\mathrm{e}_r \\ h \vdash_{\mathcal{V}} \sigma\cdot\mathrm{e}'_r \end{array}}{h \vdash_{\mathcal{V}} \sigma\cdot\mathrm{e}_r.m(\mathrm{e}'_r\ \textsf{prv}\ \mathbb{B}_{\sigma,\mathbb{a}})}$$

(ad-call-2)
$$\frac{\begin{array}{l} h,\sigma \vdash v : \mathbb{a}\, c' \\ \mathcal{B}(c', m) = \_,\ c \\ h \models \mathbb{B}_{\sigma,\mathbb{a}},\ \sigma \\ \mathbb{a} \sqsubseteq \mathbb{C}_{\sigma,c,m} \\ h \vdash_{\mathcal{V}} \sigma\cdot v \\ h \vdash_{\mathcal{V}} \sigma\cdot v' \end{array}}{h \vdash_{\mathcal{V}} \sigma\cdot v.m(v')}$$

(ad-start)
$$\frac{h \vdash_{\mathcal{V}} \sigma'\cdot e}{h \vdash_{\mathcal{V}} \sigma\cdot\sigma'\cdot\textsf{call}\, e\ \textsf{prv}\ \mathbb{E}_{\sigma'}}$$

(ad-frame)
$$\frac{h \vdash_{\mathcal{V}} \sigma'\cdot\mathrm{e}_r}{h \vdash_{\mathcal{V}} \sigma\cdot\sigma'\cdot\textsf{ret}\ \mathrm{e}_r\ \textsf{prv}\ \mathbb{E}_{\sigma'}}$$

Figure 10. Verified runtime expressions.

$$\overline{\sigma} = \overline{\sigma_1}\cdot\sigma\cdot\sigma'\cdot\overline{\sigma_2} \quad \Rightarrow \begin{cases} \sigma' = (\iota, \_, c', m) \\ h,\sigma \vdash \iota : \mathbb{a}\ \_ \\ c' <: c,\ \mathbb{a} \sqsubseteq \mathbb{C}_{\sigma,c,m} \end{cases}$$

Definition 14. *The function stack* : RExpr $\rightarrow$ Stk* *yields the stack of a runtime expression:*

$$\mathit{stack}(E[\mathrm{e}_r]) \quad = \begin{cases} \sigma\cdot\mathit{stack}(\mathrm{e}'_r) & \textit{if } \mathrm{e}_r = \sigma\cdot\mathrm{e}'_r \\ \epsilon & \textit{otherwise} \end{cases}$$

Definition 15. *For every program, the judgement:*
$\vdash_{\mathit{wf}}$: (Hp × Stk × Stk × A) ∪ (Env × Hp × Stk) ∪ Prg
is defined as:

- $\vdash_{\mathit{wf}} P \quad \Leftrightarrow \begin{cases} \textit{(F1)} & \mathcal{M}(c,m) = t,\ t' \ \Rightarrow \\ & \quad \exists e.\ \mathcal{B}(c,m) = e, \_, \quad c, m, t \vdash e : t' \\ \textit{(F2)} & c <: c',\ \mathcal{F}(c', f) = t, c'' \ \Rightarrow \\ & \quad \mathcal{F}(c, f) = t', c'',\ t' = t \\ \textit{(F3)} & c <: c',\ \mathcal{M}(c,m) = t, t', \\ & \quad \mathcal{M}(c', m) = t'', t''' \ \Rightarrow \\ & \quad\quad t = t'',\ t' = t'''' \\ \textit{(F4)} & c <: c',\ \mathcal{B}(c', m) = e', c'' \ \Rightarrow \\ & \quad \exists c'''.\ \mathcal{B}(c,m) = e, c''',\ c''' <: c'' \end{cases}$
- $h, \sigma \vdash_{\mathit{wf}} \sigma' : \mathbb{a} \quad \Leftrightarrow \quad \sigma' = (\iota, \_, \_, \_), \quad h, \sigma \vdash \iota : \mathbb{a}\ \_$
- $\Gamma \vdash_{\mathit{wf}} h, \sigma \quad \Leftrightarrow \begin{cases} \exists c, m, t, \iota, v. \\ \quad \Gamma = c, m, t, \quad \sigma = (\iota, v, c, m), \\ \quad \mathit{cls}(h,\iota) <: c, \quad h, \sigma \vdash v : t \end{cases}$

$h, \sigma \vdash_{\mathbf{wf}} \sigma' : \mathbb{a}$ says that the receiver of σ' is within $\mathbb{a}$ as seen from the point of view of σ. $\Gamma \vdash_{\mathbf{wf}} h, \sigma$ says that h, σ respect the typing environment Γ. $\vdash_{\mathbf{wf}} P$ defines well-formed programs as those where method bodies respect their signatures *(F1)*, fields are not overridden *(F2)*, overridden methods preserve typing *(F3)*, and do not "skip superclasses" *(F4)*.

Definition 16. *A programming language* PL *has a* sound type system *if all programs* $P \in$ PL *satisfy the constraints:*

$$\begin{array}{ll} \textit{(T1)} & \Gamma \vdash e : t,\ \ t <: t' \Rightarrow \Gamma \vdash e : t' \\ \textit{(T2)} & h \vdash \mathrm{e}_r : t,\ \ t <: t' \Rightarrow h \vdash \mathrm{e}_r : t' \\ \textit{(T3)} & h \vdash \mathrm{e}_r : t,\ \ h \simeq h' \Rightarrow h' \vdash \mathrm{e}_r : t \\ \textit{(T4)} & h \vdash \sigma\cdot\iota : \_\, c \Rightarrow \mathit{cls}(h,\iota) <: c \\ \textit{(T5)} & h \vdash \sigma\cdot\iota.m(v) : t \Rightarrow \begin{cases} h \vdash \sigma\cdot\iota : \mathbb{a}\, c \\ \mathcal{M}(c,m) = t', t \\ h \vdash \sigma\cdot v : t' \end{cases} \\ \textit{(T6)} & \sigma = (\iota, \_, \_, \_),\ h \vdash \sigma\cdot\iota' : \mathbb{a}\ \_ \Rightarrow \iota' \in \llbracket \mathbb{a} \rrbracket_{h,\iota} \\ \textit{(T7)} & \Gamma \vdash e : \mathbb{a}\, c,\ \ \Gamma \vdash h, \sigma \Rightarrow h \vdash \sigma\cdot e : \mathbb{a}\, c \\ \textit{(T8)} & \left.\begin{array}{l} \vdash_{\mathit{wf}} P, \quad h, \sigma \vdash \mathrm{e}_r : t \\ \mathrm{e}_r,\ h \longrightarrow \mathrm{e}'_r,\ h' \end{array}\right\} \Rightarrow h', \sigma \vdash \mathrm{e}'_r : t \end{array}$$

(T1) and *(T2)* express subsumption. *(T3)*: runtime expression typing does not depend on the field values in the heap. *(T4)*: addresses are typed according to their class in the heap. *(T5)*: method call typing implies that the parameter type and return type set by $\mathcal{M}$ for that method are respected. *(T6)*: the area component of a type assigned to an address respects the projection given for that area with respect to the same viewpoint of the typing. *(T7)* is the correspondence between typing source expressions and runtime expressions for heaps and stack frames that respect the typing environment; *(T8)*: for all well-formed programs, reduction preserves typing.

B. Appendix - The Instantiations

We present more details about the formalization of the six verification techniques.

Poetzsch-Heffter Define A $=$ {any} and R $=$ {any} with interpretations $\llbracket \textsf{any} \rrbracket_{h,\iota} = \mathit{dom}(h)$ and $\llbracket \textsf{any} \rrbracket_{h,\iota} = \mathit{all}(h)$.

Huizing and Kuiper Define A $=$ {self, any} with interpretation $\llbracket \textsf{self} \rrbracket_{h,\iota} = \{\iota\}$ and $\llbracket \textsf{any} \rrbracket_{h,\iota} = \mathit{dom}(h)$, and with

$$\begin{array}{ll} \llbracket \textsf{vul}\langle c\rangle \rrbracket_{h,\iota} = & \{ \mid \text{the invariant of } c' \text{ contains an expression} \\ & \textbf{this}\,.g_1 \ldots g_n.f\ (n \geq 0) \text{ where } \mathcal{F}(c,f) = \_, \_ \wedge \\ & \mathit{fld}(h, \mathit{fld}(h, \mathit{fld}(h, \iota', g_1), \ldots), g_n) = \iota\} \cup \\ & \{(\iota, c') \mid \mathit{cls}(h,\iota) <: c'\} \\ \llbracket \textsf{any} \rrbracket_{h,\iota} = & \mathit{all}(h) \end{array} \quad .$$

Leavens & Müller Define the area set A $=$ {emp, any} with interpretation $\llbracket \textsf{emp} \rrbracket_{h,\iota} = \emptyset$ and $\llbracket \textsf{any} \rrbracket_{h,\iota} = \mathit{dom}(h)$, and the region set R $= \{\textsf{any}, \textsf{self}\langle c\rangle, \textsf{any\_vs}\langle c\rangle\}$ with interpretation:

$$\llbracket \textsf{any} \rrbracket_{h,\iota} = \mathit{all}(h) \qquad \llbracket \textsf{any\_vs}\langle c\rangle \rrbracket_{h,\iota} = \{(\iota', c') \mid \textsf{visI}(c', c)\}$$
$$\llbracket \textsf{self}\langle c\rangle \rrbracket_{h,\iota} = \{(\iota, c) \mid \forall c'.\textsf{visF}(c, c') \Leftrightarrow \textsf{visI}(c, c')\}$$

Müller er al. Assume heap operation giving an object's owner

$$\mathit{ownr} \ :\ \text{Hp} \times \text{Adr} \ \rightarrow\ \text{Adr} \times \text{Cls}$$

The interpretation of areas is defined as

$$\begin{array}{l} \llbracket \textsf{self} \rrbracket_{h,\iota} = \{\iota\} \qquad \llbracket \textsf{any} \rrbracket_{h,\iota} = \mathit{dom}(h) \qquad \llbracket \textsf{emp} \rrbracket_{h,\iota} \quad = \emptyset \\ \llbracket \textsf{rep}\langle c\rangle \rrbracket_{h,\iota} = \{\iota' \mid \ \mathit{ownr}(h, \iota') = \iota\, c\} \\ \llbracket \textsf{peer} \rrbracket_{h,\iota} = \{\iota' \mid \ \mathit{ownr}(h, \iota') = \mathit{ownr}(h, \iota)\} \\ \llbracket \mathbb{a}_1 \sqcup \mathbb{a}_2 \rrbracket_{h,\iota} = \llbracket \mathbb{a}_2 \rrbracket_{h,\iota} \cup \llbracket \mathbb{a}_2 \rrbracket_{h,\iota} \end{array}$$

The interpretation of regions is defined as

$$\llbracket \mathsf{emp} \rrbracket_{h,\iota} = \emptyset \qquad \llbracket \mathsf{self}\langle c \rangle \rrbracket_{h,\iota} = \{(\iota, c) \mid \ \mathit{cls}(h,\iota) <: c\}$$

$$\llbracket \mathsf{super}\langle c \rangle \rrbracket_{h,\iota} = \big\{(\iota, c') \mid \ c <: c'\big\}$$

$$\llbracket \mathsf{peer}\langle c \rangle \rrbracket_{h,\iota} = \big\{(\iota', c') \mid \mathit{ownr}(h,\iota') = \mathit{ownr}(h,\iota) \wedge \mathsf{vis}(c', c)\big\}$$

$$\llbracket \mathsf{rep} \rrbracket_{h,\iota} = \big\{(\iota', c') \mid \mathit{ownr}(h,\iota') = \iota \ \_\big\} \qquad \llbracket \mathsf{own} \rrbracket_{h,\iota} = \{\mathit{ownr}(h,\iota)\}$$

$$\llbracket \mathbb{r}_1 ; \mathbb{r}_2 \rrbracket_{h,\iota} = \bigcup_{(\iota',c) \in \llbracket \mathbb{r}_1 \rrbracket_{h,\iota}} \llbracket \mathbb{r}_2 \rrbracket_{h,\iota'}$$

$$\llbracket \mathsf{rep}^+ \rrbracket_{h,\iota} = \llbracket \mathsf{rep} \rrbracket_{h,\iota} \cup \llbracket \mathsf{rep}; \mathsf{rep}^+ \rrbracket_{h,\iota}$$

$$\llbracket \mathsf{own}^+ \rrbracket_{h,\iota} = \llbracket \mathsf{own} \rrbracket_{h,\iota} \cup \llbracket \mathsf{own}; \mathsf{own}^+ \rrbracket_{h,\iota}$$

Lu et al. We interpret areas and regions as follows:

$$\llbracket \mathsf{emp} \rrbracket_{h,\iota} = \emptyset \qquad \llbracket \mathsf{self} \rrbracket_{h,\iota} = \{\iota\} \qquad \llbracket \mathbb{a} \sqcup \mathbb{a}' \rrbracket_{h,\iota} = \llbracket \mathbb{a} \rrbracket_{h,\iota} \cup \llbracket \mathbb{a}' \rrbracket_{h,\iota}$$

$$\llbracket c\langle \overline{K} \rangle \rrbracket_{h,\iota} = \big\{\iota' \mid h \vdash \iota' : c\langle \overline{\iota} \rangle, \forall i.\, \iota_i \in \{\!\!\{ K_i \}\!\!\}_{h,\iota}\big\}$$

$$\llbracket \mathsf{emp} \rrbracket_{h,\iota} = \llbracket \mathsf{top} \rrbracket_{h,\iota} = \llbracket \mathsf{bot} \rrbracket_{h,\iota} = \emptyset \quad \llbracket \mathsf{self} \rrbracket_{h,\iota} = \{(\iota, c) \mid ...\}$$

$$\llbracket K \rrbracket_{h,\iota} = \big\{(\iota', c) \mid \iota' \in \{\!\!\{ K \}\!\!\}_{h.\iota},\ \mathit{cls}(h,\iota') <: c\big\}$$

$$\llbracket K; \mathbb{r} \rrbracket_{h,\iota} = \begin{cases} \mathit{all}(h) & K = \mathsf{top}, \mathbb{r} = \mathsf{rep}^* \ \vee K = \mathsf{bot}, \mathbb{r} = \mathsf{own}^* \\ \bigcup_{(\iota',c) \in \llbracket K \rrbracket_{h,\iota}} \llbracket \mathbb{r} \rrbracket_{h,\iota'} & \mathbb{r} \in \{\mathsf{rep}^*, \mathsf{own}^*\} \end{cases}$$

$$\llbracket \mathsf{rep}^* \rrbracket_{h,\iota} = \big\{\iota' \mid h \vdash \iota' \preceq^* \iota\big\} \qquad \llbracket \mathsf{own}^* \rrbracket_{h,\iota} = \big\{\iota' \mid h \vdash \iota \preceq^* \iota'\big\}$$

$$\{\!\!\{ X \}\!\!\}_{h,\iota} = \big\{\iota_i \mid h \vdash \iota : c\langle \overline{\iota} \rangle,\ c \text{ has formal parameters } \bar{X},\ X = X_i\big\}$$

As usual in ownership systems, $h \vdash \iota : c\langle \overline{\iota} \rangle$ describes that ι points to an object of a subclass of $c\langle \overline{\iota} \rangle$, while $h \vdash \iota' \preceq \iota$ expresses that ι' is owned by ι, and $h \vdash \iota' \preceq^* \iota$ is the transitive closure.

The owner extraction function $\mathcal{O}$ is defined as:

$$\mathcal{O}_{\mathbb{a},c} = \begin{cases} K_1, & \text{if } \mathbb{a} = c\langle \overline{K} \rangle \\ X_1, & \text{if } \mathbb{a} = \mathsf{self}, \text{ class } c \text{ has formal parameters } \bar{X}. \\ \bot & \text{otherwise} \end{cases}$$