draft copy

A Unified Framework for Verification Techniques for Object Invariants (Full Paper)

date: 30th August 2007
please, check this site for updates and extensions of this paper

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 which 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 (e.g., which fields can be updated), restrictions on invariants (what an invariant may refer to), use of advanced type systems (such as Universe types or ownership), 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 understand whether/why these techniques are sound, whether/why they are modular, and to compare their expressiveness. This general lack of understanding also hampers the development of new approaches.

In this paper, we develop and formalise a unified framework to describe verification techniques for object invariants. We distil seven parameters, which characterise a verification technique, and identify sufficient conditions on these parameters under which a verification technique is sound. We also define what it means for a technique to be modular. 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, modularity, 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 [31], the Larch languages [4, 17, 18], the Java Modeling Language JML [19], and Spec# [2]. Object invariants express consistency criteria for objects, which guarantee their correct

[Copyright notice will appear here once 'preprint' option is removed.]


```
class C {
  int a, b;
  invariant 0 <= a < b;

  C() { a := 0; b := 3; }

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

class Client {
  C c;
  invariant c.a <= 10;

  /∗ methods omitted ∗/
}

class D extends C {
  invariant a <= 10;

  /∗ methods omitted ∗/
}
```

Figure 1. An example (adapted from [23]) illustrating the three main challenges for the verification of object invariants.

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* [34]. Invariants may be violated 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 violated 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.

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 violates 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 violate the invariant of p. In our example, a call o.m might break the invariant of a Client object p where $p.\mathsf{c} = o$. 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 violate 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.

A number of verification techniques have been suggested to address some or all of these problems [1, 3, 14, 20, 23, 29, 32, 33, 34, 38]. These techniques 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:* What is required to be proven? 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.

These differences, together with the fact that the verification techniques are at times not formally specified or otherwise intertwined with the type system, make it hard to understand exactly why verification techniques satisfy claimed properties such as soundness and modularity, and complicate direct comparisons. More importantly though, this general lack of understanding makes it hard to propose new approaches and to choose which techniques to adopt in specification languages.

In this paper, we present a unified framework that formalises the proposed verification techniques, abstracts away from their differences, and allows direct comparisons. We also formalise soundness and modularity for such verification techniques. Our work concentrates on those techniques that require invariants to hold in visible states. These techniques constitute the vast majority of the techniques described in the literature.

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 is important to handle 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.
3. *Proof obligations:* The regions $\mathbb{P}$ and $\mathbb{E}$ describe the invariants that have to be proven to hold in the pre-state of 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.

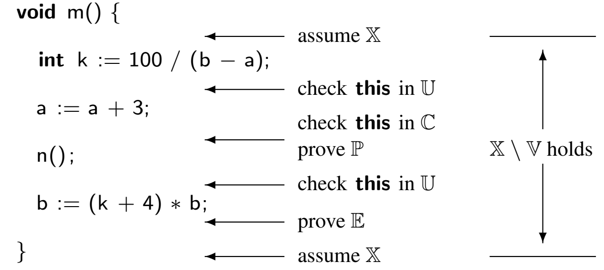

Figure 2. Role of framework parameters for method m (Fig. 1).

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, $\mathbb{P}$ 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 $\mathbb{D}$.

Developing our framework was challenging because different verification techniques (1) use different type systems to restrict programs and invariants, and do not make a clear distinction between the type system and the verification technique, (2) use different specification languages to express invariants, and (3) use different verification logics. To deal with this diversity within one unified framework, we take the following approach:

1. We make a clear delineation between the framework and the type system and instead of describing one particular type system, we state requirements on the type systems used with our framework.
2. We assume a judgment that describes that an object satisfies the invariant of a class in a heap. We require that a field update preserves the invariant if it 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. These techniques are described in terms of seven framework parameters.
2. We formalise soundness for verification techniques.
3. We identify conditions on the framework parameters that guarantee soundness of a verification technique.
4. We identify conditions on the framework parameters that guarantee modularity of a verification technique.

5. We show that our framework indeed describes all major verification techniques for visible state invariants. We use our framework to compare these techniques.

Outline. The paper is organised as follows. Sec. 2 formalises programs and invariant semantics. Sec. 3 describes our framework and defines soundness and modularity. Sec. 4 presents sufficient conditions for a verification technique to be sound, and states a general soundness theorem. Sec. 5 instantiates our framework with existing verification techniques. We discuss related work in Sec. 6.

2. Invariant Semantics

We formalise invariant semantics through an operational semantics. This semantics defines visible states, execution points at which invariants are expected to hold. In order to cater for the different techniques, the semantics is parameterised by regions; these regions are used 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 the appendix. Our language description aims at being as generic as possible, since we intend to apply our analysis to different languages. Thus, instead of describing precisely what the language entities are, we group them as *structures* and state operations and properties on such structures. The actual entities of a language are then instantiations of these structures, implementing the necessary operations and satisfying the required properties.

We assume sets of identifiers for class names $\textsc{Cls}$, field names $\textsc{Fld}$, and method names $\textsc{Mthd}$, and use variables $c \in \textsc{Cls}$, $f \in \textsc{Fld}$ and $m \in \textsc{Mthd}$.

Runtime Structures. A *runtime structure* is a tuple consisting of a set of heaps $\textsc{Hp}$, addresses $\textsc{Adr}$, and values $\textsc{Val} = \textsc{Adr} \cup \{\mathsf{null}\}$, with the convention that $h \in \textsc{Hp}$ and $\iota \in \textsc{Adr}$. 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. To express such requirements, we also define relations $\simeq$ and $\preceq$, denoting heap equivalence and heap extension, respectively. See Def. 32 in the appendix for details.

A stack frame $\sigma \in \textsc{Stk} = \textsc{Adr} \times \textsc{Adr} \times \textsc{Mthd} \times \textsc{Cls}$ 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. 33 in the appendix) consists of a set $\mathbf{A}$ of object-areas and a set $\mathbf{R}$ of invariant-regions. An area $\mathbb{a} \in \mathbf{A}$ is a syntactic representation for a set of objects; a region $\mathbb{r} \in \mathbf{R}$ 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, 29, 34] 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 $\triangleright$ [7], which adapts a region to the viewpoint described by an area.

We define a type, $t \in \textsc{Typ}$, as a tuple of an area and a class. 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 the source expressions $e \in \textsc{Expr}$. 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 first 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.

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

Figure 3. Source and runtime expression syntax.

In Fig. 3, we also define runtime expressions $e_r \in \textsc{RExpr}$. 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 the appendix (Def. 34), 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 $\textsc{Prg}$ of programs, a runtime structure, and an area/region structure (see Def. 35 in the appendix). Each $P \in \textsc{Prg}$ 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 judgments $\Gamma \vdash e : t$ and $h \vdash e_r : t$ for source and runtime expressions, respectively. An environment, $\Gamma \in \textsc{Env}$, is a tuple of the class containing the current method, the method identifier, and the type of the sole argument.

Finally, the judgment $h \models \iota, c$ expresses that in heap h, the object at address ι satisfies the invariant declared in class c. The judgment trivially holds if the object is not allocated ($\iota \notin dom(h)$) or is not an instance of c ($cls(h, \iota) \not<: c$). 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 $h \models \mathbb{r},\ \iota$ defined as

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

We sometimes find it convenient to write $h \models [\![\mathbb{r}]\!]_{h,\iota}$ for $h \models \mathbb{r},\ \iota$.

Operational Semantics. Given a program *P* and a region $\mathbb{X}_{c,m}$ that characterises the invariants that are expected to hold in the

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

(rNew)
$$\frac{\sigma = (\iota, _, _, _) \quad h', \iota' = new(h, \iota, t)}{\sigma{\cdot}\mathsf{new}\, t,\ h \longrightarrow \sigma{\cdot}\iota',\ h'}$$

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

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

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

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

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

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

(rLaunchEx)
$$\frac{\sigma = (\iota, _, c, m) \quad h \not\models \mathbb{X}_{c,m},\ \iota}{\sigma{\cdot}\mathsf{call}\, e,\ h \longrightarrow \sigma{\cdot}\mathsf{fatalExc},\ h}$$

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

(rFrameEx)
$$\frac{\sigma = (\iota, _, c, m) \quad h \not\models \mathbb{X}_{c,m},\ \iota}{\sigma{\cdot}\mathsf{ret}\, v,\ h \longrightarrow \mathsf{fatalExc},\ h}$$

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

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

Figure 4. Reduction rules of operational semantics.

visible states of a method m of class c, the *runtime semantics* is the relation with the following signature, defined in Fig. 4:

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

The first seven rules are rather standard for an imperative object-oriented language. Note that in rNew, a new object is created using the function *new*, which takes a type as third parameter rather than a class, thereby making the semantics parametric *wrt.* the type system: different type systems may use different extensions, expressed here as areas, 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. The 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 the 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 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.

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 does allow verfExc to be thrown, but this will never happen in a verified program.

3. Verification Techniques

In this section, we formalise verification techniques and their connection to programs. Moreover, we define what it means for a verification technique to be sound and modular.

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 (Verification Technique)**.** *A* verification technique $\mathcal{V}$ for a programming language *is a mapping from programs into a tuple:*

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

where

$$\begin{aligned} \text{EXP} &= \text{CLS} \times \text{MTHD} \rightarrow \mathbf{R} \\ \text{VUL} &= \text{CLS} \times \text{MTHD} \rightarrow \mathbf{R} \\ \text{DEP} &= \text{CLS} \rightarrow \mathbf{R} \\ \text{PRE} &= \text{CLS} \times \text{MTHD} \times \mathbf{A} \rightarrow \mathbf{R} \\ \text{END} &= \text{CLS} \times \text{MTHD} \rightarrow \mathbf{R} \\ \text{CLL} &= \text{CLS} \times \text{MTHD} \times \text{CLS} \rightarrow \mathbf{A} \\ \text{ASS} &= \text{CLS} \times \text{MTHD} \times \text{CLS} \times \text{MTHD} \rightarrow \mathbf{A} \end{aligned}$$

For a the verification technique applied to a program, we use $\mathbb{X}_{c,m}$, $\mathbb{V}_{c,m}$, $\mathbb{D}_c$, $\mathbb{P}_{c,m,\mathbb{a}}$, $\mathbb{E}_{c,m}$, $\mathbb{C}_{c,m,c',m'}$, and $\mathbb{U}_{c,m,c'}$ for the application of the first component to a class and method name, *etc.* 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 [20, 21] 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 a field declared in class c. The parameter c is used, for instance, to prevent invariants from depending on fields declared in c's superclasses [20, 34].

$\mathbb{P}_{c,m,\mathbb{a}}$**:** 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 verification technique to permit calls depending on attributes (such as purity or effect specifications) of the caller and the callee.

(vs-null) $$\frac{}{\Gamma \vdash_{\mathcal{V}} \textsf{null}}$$ (vs-Var) $$\frac{}{\Gamma \vdash_{\mathcal{V}} x}$$ (vs-this) $$\frac{}{\Gamma \vdash_{\mathcal{V}} \textsf{this}}$$ (vs-new) $$\frac{}{\Gamma \vdash_{\mathcal{V}} \textsf{new}\, t}$$ (vs-fld) $$\frac{\Gamma \vdash_{\mathcal{V}} e}{\Gamma \vdash_{\mathcal{V}} e.f}$$

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

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

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

Figure 5. Well-Verified source expressions and classes.

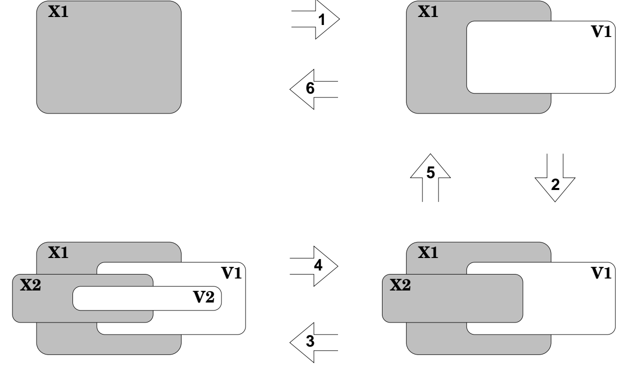

Figure 6. Open calls and valid invariants in a heap

Role of the Seven Components. The operational semantics uses a verification technique to specify the invariants expected in visible states, whereas the static analysis imposed by the verification technique describes program restrictions and proof obligations. More precisely, the operational semantics uses $\mathbb{X}$, to be checked at visible states; soundness requires that $\mathbb{X} \setminus \mathbb{V}$ holds during a method activation. Well-verification static analysis describes proof obligations using $\mathbb{P}$ and $\mathbb{E}$, and ensures program restrictions, through $\mathbb{U}$ and $\mathbb{C}$, are respected. Finally, $\mathbb{D}$ restricts invariants for well-verified programs (*cf.* Def. 2). Sec. 4 gives five conditions on these components that guarantee soundness.

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 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.

The seven components depend on class and method identifiers; these can be extracted from an environment Γ or a stack frame σ in the obvious way. Thus, for $\Gamma = c, m, _$, $\sigma = (\iota, _, c, m)$, and $\sigma' = (_, _, c', m')$, we use $\mathbb{X}_\Gamma$, $\mathbb{X}_\sigma$ as shorthands for $\mathbb{X}_{c,m}$; we also use $\mathbb{P}_{\Gamma,\mathrm{a}}$ and $\mathbb{P}_{\sigma,\mathrm{a}}$ as shorthands for $\mathbb{P}_{c,m,\mathrm{a}}$.

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, and 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 $\mathbb{P}$ before a call. Finally, a class is well-annotated if the body of each of its methods is well-annotated and ends with a proof obligation for $\mathbb{E}$ (vs-class).

A program *P* is well-verified wrt. $\mathcal{V}$, denoted as $\vdash_{\mathcal{V}} P$, if all classes are well-verified and all class invariants respect the dependency restrictions dictated by $\mathbb{D}$. More precisely, *(W2)* below states that the invariant of an object ι' declared in a class c' may be affected by an update of a field of a class c only if the invariant is within $\mathbb{D}_c$.

Definition 2 (Well-Verified Programs)**.**

$\vdash_{\mathcal{V}} P \Leftrightarrow$

$$\textit{(W1)} \quad \forall c \in P.\ \vdash_{\mathcal{V}} c$$

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

Fig. 13 in the appendix defines the judgment $h \vdash_{\mathcal{V}} e_r$ for well-annotated 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 known 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 known to hold (emp denotes the emptyset of object invariants):

Definition 3 (Valid Invariants in a Heap)**.**

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

An empty stack occurs in the initial program state where no invariants hold as there are no allocated objects. For each additional stack frame σ, we know that the corresponding method m requires $\mathbb{X}_\sigma$. Thus, we add these invariants to the valid invariants. Method m may break its vulnerable invariants even if they are not vulnerable to its callers, provided that m re-establishes these invariants before it terminates. Consequently, we subtract $\mathbb{V}_\sigma$ from the set of valid invariants.

Fig. 6 depicts this mechanism of invariant violation and reestablishing for the execution of two consecutive calls. The regions $\mathbb{X}_1, \mathbb{V}_1$ denote the expected and vulnerable regions of the outer call and $\mathbb{X}_2, \mathbb{V}_2$ are the expected and vulnerable regions of the subcall. The first call violates $\mathbb{V}_1$ but $\mathbb{X}_1 \setminus \mathbb{V}_1$ hold throughout the call (1). Before making the subcall, it establishes all of $\mathbb{X}_2$ (2). The subcall violates $\mathbb{V}_2$ (3) but reestablishes all of $\mathbb{X}_2$ before returning (4); similarly, after the first call resumes control (5), it re-establishes $\mathbb{X}_1$ at the end of its execution (6).

Not all stacks are valid with respect to well-verification. The verification component $\mathbb{C}$ prohibits certain sequences of stack frames because the receiver of a stack frame must be in the callable area of the preceding stack frame.

Definition 4 (Valid Stacks)**.** *Stack* $\overline{\sigma}$ *is* valid *wrt. heap* h *and a verification technique* $\mathcal{V}$*, denoted as* $h \vdash_{\mathcal{V}} \overline{\sigma}$ *iff:*

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

Assuming a function *stack* : REXPR $\rightarrow$ STK*, which extracts a stack from a runtime expression, defined as

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

we can define the notion of a valid state for an execution stack. A state with heap h and stack $\overline{\sigma}$ is *valid* iff:

(V1) $\overline{\sigma}$ is a valid stack, $h \vdash_{\mathcal{V}} \overline{\sigma}$, meaning that the receivers of consecutive method calls are within the respective $\mathbb{C}$ areas.

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

(V3) If the state is a visible state, additionally the expected invariants $\mathbb{X}_\sigma$ hold (σ is the topmost frame, *i.e.*, $\overline{\sigma} = \overline{\sigma'} \cdot \sigma$).

These properties are formalised in Def. 5. A state is determined by a heap h and a runtime expression e_r.

Definition 5 (Valid State)**.** *A state with heap h and runtime expression* e_r *is valid for a verification technique* $\mathcal{V}$*, denoted by* $\mathrm{e}_r \models_{\mathcal{V}} h$*, iff:*

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

Soundness. Intuitively, a verification technique is *sound* if verified programs only produce valid states and do not throw fatal exceptions (*cf.* rLaunchEx and rFrameEx in Fig. 4). More precisely, a verification technique $\mathcal{V}$ is sound for a programming language *PL* iff for all well-formed and well-annotated programs $P \in PL$, any well-typed and well-annotated runtime expression e_r executed in a valid state reduces to another well-annotated expression e'_r with a resulting valid state. Note that a well-annotated e'_r contains no fatalExc (see Fig. 13).

We stress the separation of concerns, *i.e.*, the distinction between verification technique soundness, as defined below, and type system soundness. At the same time we note that the definition of the former relies on properties of the latter. Well-formedness of a program *P* is denoted by $\vdash_{\mathbf{wf}} P$ (see Def. 36 in the appendix). Well-typedness of a runtime expression e_r is denoted by $h \vdash \mathrm{e}_r : t$ (see Def. 35 in the appendix). Type soundness is a requirement on the type system and is assumed here.

Definition 6 (Sound Verification Technique)**.** *A verification technique* $\mathcal{V}$ *is sound for a programming language PL iff for all programs* $P \in PL$*:*

$$\left.\begin{array}{l} \vdash_{\mathit{wf}} P,\ \ h \vdash \mathrm{e}_r : t, \\ \vdash_{\mathcal{V}} P,\ \ \mathrm{e}_r \models_{\mathcal{V}} h,\ \ h \vdash_{\mathcal{V}} \mathrm{e}_r, \\ \mathrm{e}_r,\ h \longrightarrow \mathrm{e}'_r,\ h' \end{array}\right\} \ \Rightarrow\ \ \mathrm{e}'_r \models_{\mathcal{V}} h',\ \ h' \vdash_{\mathcal{V}} \mathrm{e}'_r$$

Modularity. A verification technique is modular if all proof obligations for a class c can be shown using the code for c, c's superclasses, and the classes used by c [22]. We call this set of classes the classes that are *visible* in c. Subclasses and client classes of c are in general not visible in c. We assume a reflexive, transitive predicate $\mathsf{vis}(c, c')$ that expresses that class c is visible in class c'. A verification technique can define vis in various ways, for instance, using an import-relation among modules [33] or the accessibility of class members [20].

Definition 7. *A verification technique is* modular *if it only imposes proof obligations for invariants of visible classes:*

$$\begin{array}{llll} (M1) & (\iota', c') \in [\![\mathbb{P}_{c,m,\mathbb{a}}]\!]_{h,\iota} & \Rightarrow & \textit{vis}(c', c) \\ (M2) & (\iota', c') \in [\![\mathbb{E}_{c,m}]\!]_{h,\iota} & \Rightarrow & \textit{vis}(c', c) \end{array}$$

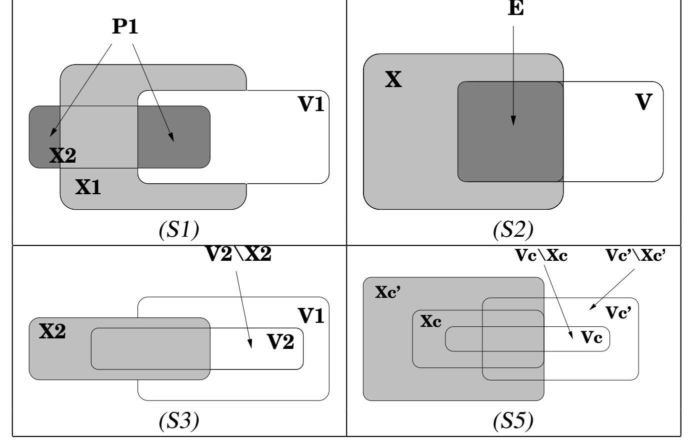

Figure 7. Well Structured Conditions

4. Well-Structured Verification Techniques

In this section, we identify conditions on the components of a verification technique that are sufficient for soundness.

Definition 8 (Well-Structured Verification Methodology)**.** *A verification technique is* well-structured *if, for all programs in the programming language:*

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

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 constraints, *(S1)* and *(S2)*, relate proof obligations with the expected invariants at visible states. *(S1)* ensures that the invariants that are expected to hold at the beginning of a subcall, which do not presently hold in the current call, need to be included in the proof obligation prescribed before the subcall. *(S2)* ensures that the invariants that held at the beginning of the call, where subsequently violated during the call and are required to hold again at the end of the call are included in the prescribed proof obligation at the end of the method body. The top two quadrants of Fig. 7 informally depict region scenarios that would result from *(S1)* and *(S2)*; as in previous examples, regions appended by 1 denote regions of the caller, whereas regions appended by 2 denote regions of the callee. The darkest shades denote regions that must be included in the proof obligations.

Constraint *(S3)* restricts the invariants that can be violated during a subcall but not reestablished at the end through $\mathbb{E}$, *i.e.*, $\mathbb{V} \setminus \mathbb{X}$. More precisely, these residue violated invariants must be included in the vulnerable invariants of the caller (bottom-left quadrant of Fig. 7).

Constraint *(S4)* ensures that the invariants violated *directly* in an active method through the area it is allowed to update, $\mathbb{U}$, is bounded by the vulnerable region of that method.

Finally, *(S5)* establishes constraints for subclasses in a well-structured Verification Methodology: An overridden method in a

subclass may assume less invariants to hold at visible states than that same method in its superclass. Moreover, the residue violated invariants (those broken but not reestablished) of an overridden method in a subclass , denoted by the region $\mathbb{V}c \setminus \mathbb{X}c$ in the bottom right quadrant of Fig. 7, have to be included in the residue violated invariants of that same method in its superclass, *i.e.,* $\mathbb{V}c' \setminus \mathbb{X}c'$ in Fig. 7.

To further motivate the well-structured requirements of Def. 8, let us refer back to Fig. 6. *(S1)* ensures that by proving $\mathbb{P}$ before making the subcall, we can safely make move (2) and reach a valid visible state at the beginning of the subcall. *(S2)* ensures that by proving $\mathbb{E}$ at the end of both method bodies we can make moves (4) and (6) and reach a valid visible states at the end of both calls. Constraint *(S3)* ensures that when the caller resumes control after the subcall, we can make move (5) and reach a state where $\mathbb{X}1 \setminus \mathbb{V}1$ holds in the heap. *(S4)* guarantees that $\mathbb{X}\setminus\mathbb{V}$ always hold for any call by ensuring that $\mathbb{V}$ is an adequate upper limit for the effect of a call. Finally, but crucially, *(S5)* permits static analysis of the relationship between $\mathbb{X}$ and $\mathbb{V}$ in the presence of dynamic dispatch of subclass methods because static region information of what is expected to hold at the visible states and what are the residue vulnerable invariants of the superclass imply the corresponding regions for the same method in the subclass.

The five conditions from Def. 8 guarantee soundness, as stated in Def. 6.

Theorem 9 (Soundness For Visible-State Verification Techniques)**.** *A well-structured verification technique built on top of a PL with a sound type system is sound.*

4.1 Proof of Soundness Theorem

The proof of Theorem 9 uses a number of lemmas we briefly discuss here. For a start, we require to show the correspondence between well-verified source expressions (Fig. 5) and well-verified runtime expressions (Fig. 13).

Lemma 10 (Substitution/Instantiation)**.**

$$\Gamma \vdash_{\mathcal{V}} e,\ \ \Gamma \vdash h, \sigma \ \Rightarrow\ h \vdash_{\mathcal{V}} \sigma \cdot e$$

Proof. By induction on the derivation of $\Gamma \vdash_{\mathcal{V}} e$. □

We also require the following lemma stating that the adaptation operation is adequate and monotonic.

Lemma 11 (Adaptation Correspondence)**.**

1. $h \vdash \sigma \cdot \iota : \mathbb{a}, _ \ \Rightarrow\ [\![\mathbb{r}]\!]_{h,\iota} \subseteq [\![\mathbb{a} \triangleright \mathbb{r}]\!]_{h,\sigma}$
2. $\mathbb{a}_1 \sqsubseteq \mathbb{a}_2 \ \Rightarrow\ \mathbb{a}_1 \triangleright \mathbb{r} \subseteq \mathbb{a}_2 \triangleright \mathbb{r}$
3. $\mathbb{r}_1 \sqsubseteq \mathbb{r}_2 \ \Rightarrow\ \mathbb{a} \triangleright \mathbb{r}_1 \subseteq \mathbb{a} \triangleright \mathbb{r}_2$

Proof. The first clause is straightforward from (T6) of Def. 37 and (P5) of Def. 35. The second and third clauses are also immediate as a result of (P5) of Def. 35. □

We also require a number of lemmas dealing with the reduction rules and heap validity. For instance, the following lemma states that heaps can only grow as a result of a reduction.

Lemma 12. $\mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h' \ \Rightarrow\ h \preceq h'$

Proof. By induction on the derivation of $\mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h'$ and Definition 32. □

The following lemma states that a well-verified runtime expression remains well-verified in an extended heap and also that well-verified values are independent of any guarding stack frame. The latter property is useful when we consider a return from a subcall in the main proof.

Lemma 13.

1. $h \vdash_{\mathcal{V}} \mathrm{e}_r,\ h \preceq h' \ \Rightarrow\ h' \vdash_{\mathcal{V}} \mathrm{e}_r$
2. $h \vdash_{\mathcal{V}} \sigma \cdot v \ \Rightarrow\ h \vdash_{\mathcal{V}} \sigma' \cdot v$

Heap validity depends solely on the stack frames of a runtime expression, thus evaluation contexts are non-influential.

Lemma 14 (Valid States)**.** *If* $stack(\mathrm{e}_r) = \sigma_1 \cdot \ldots \cdot \sigma_n$ *then*

1. $\sigma' \cdot \mathrm{e}_r \models_{\mathcal{V}} h \ \Leftrightarrow\ \sigma' \cdot E[\mathrm{e}_r] \models_{\mathcal{V}} h$
2. $\sigma' \cdot \mathrm{e}_r \models_{\mathcal{V}} h \ \Leftrightarrow\ \begin{cases} h \models \mathbb{X}_{\sigma'} \setminus \mathbb{V}_{\sigma'} \setminus \mathbb{V}_{\sigma_1} \setminus \ldots \setminus \mathbb{V}_{\sigma_n} \\ h \vdash_{\mathcal{V}} \sigma' \cdot \sigma_1 \cdot \ldots \cdot \sigma_n \\ \mathrm{e}_r \models_{\mathcal{V}} h \end{cases}$

Proof. Immediate from Definition 5. □

In order to determine the effect of updates on valid invariants in a heap we require the following lemma.

Lemma 15 (Invariant Satisfaction Effect)**.**

$$\left.\begin{array}{l} \vdash_{\mathcal{V}} P \\ h \models [\![\mathbb{r}]\!]_{h,\iota'} \\ cls(h,\iota) <: c' \\ \mathcal{F}(c', f, =)_, c \\ h' = upd(h, \iota, f, v) \end{array}\right\} \Rightarrow\ h' \models [\![\mathbb{r}]\!]_{h',\iota'} \setminus [\![\mathbb{D}_c]\!]_{h',\iota}$$

Proof. Immediate from (W2) of Def. 2 □

Thus for a well structured verification technique, we are guaranteed that certain invariants are unaffected by reductions.

Lemma 16 (Computation effects)**.** *For arbitrary invariant set* $s = \{(\iota_1, c_1), \ldots, (\iota_n, c_n)\}$*, if* $\mathcal{V}$ *is well-structured then:*

$$\left.\begin{array}{l} h \models s \\ \mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h' \\ stack(\mathrm{e}'_r) = \sigma_1 \cdot \ldots \cdot \sigma_n \end{array}\right\} \Rightarrow\ h' \models s \setminus \mathbb{V}_{\sigma_1} \setminus \ldots \setminus \mathbb{V}_{\sigma_n}$$

Proof. By induction on the derivation of $\mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h'$, Lemma 15 and $(S4)$ of Definition 8. □

Finally, we restate the soundness theorem, Theorem 9, in full and prove the main cases.

Theorem 9 Soundness for Visible-State Verification Techniques *If* $\mathcal{V}$ *is well-structured, then:*

$$\left.\begin{array}{l} \vdash_{wf} P,\ \ h \vdash \mathrm{e}_r : t, \\ \vdash_{\mathcal{V}} P,\ \ \mathrm{e}_r \models_{\mathcal{V}} h,\ \ h \vdash_{\mathcal{V}} \mathrm{e}_r, \\ \mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h' \end{array}\right\} \ \Rightarrow\ \ \mathrm{e}'_r \models_{\mathcal{V}} h',\ \ h' \vdash_{\mathcal{V}} \mathrm{e}'_r$$

Proof. The proof is by induction on the derivation of $\mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h'$. As a shorthand, we find it convenient to write $h \models \mathbb{X}_\sigma$ and $h \models \mathbb{a} \triangleright \mathbb{X}_\sigma$ instead of $h \models [\![\mathbb{X}_\sigma]\!]_{h,\sigma}$ and $h \models [\![\mathbb{a} \triangleright \mathbb{X}_\sigma]\!]_{h,\sigma}$ respectively, and similarly for the framework component $\mathbb{V}_\sigma$. For convenience we also enumerate the premises of the Theorem as

$$\vdash_{\mathbf{wf}} P, \quad (1)$$
$$h \vdash \mathrm{e}_r : t, \quad (2)$$
$$\vdash_{\mathcal{V}} P, \quad (3)$$
$$\mathrm{e}_r \models_{\mathcal{V}} h, \quad (4)$$
$$h \vdash_{\mathcal{V}} \mathrm{e}_r, \quad (5)$$
$$\mathrm{e}_r,\, h \longrightarrow \mathrm{e}'_r,\, h' \quad (6)$$

We here focus on the main cases for the derivation of (6) and leave the remaining simpler cases for the interested reader.

rAss: From the conclusion and the premises of the rule we know

$$\mathrm{e}_r = \sigma\cdot\iota.f := v \tag{7}$$
$$\mathrm{e}'_r = \sigma\cdot v \tag{8}$$
$$h' = \mathit{upd}(h, \iota, f, v) \tag{9}$$

From (7) we know (5) could only have been derived using vd-ass and from the premises of this rule we know

$$h \vdash \sigma\cdot\iota : \mathrm{a}\, c' \tag{10}$$
$$\mathcal{F}(c', f) = _, c \tag{11}$$
$$\mathrm{a} \sqsubseteq \mathbb{U}_{\sigma,c} \tag{12}$$
$$h \vdash_{\mathcal{V}} \sigma\cdot\iota \tag{13}$$
$$h \vdash_{\mathcal{V}} \sigma\cdot v \tag{14}$$

From (9) and Def. 32 *(H4)* we know

$$h \simeq h' \text{ which implies } h \preceq h' \tag{15}$$

Thus by (14), (15) and Lemma 14.1 and then by (8) we derive that the resultant configuration is still well-verified, *i.e.,*

$$h' \vdash_{\mathcal{V}} \mathrm{e}'_r$$

We still need to show that (6) reduces to a valid state. From (4) and Def. 5 we know

$$h \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma \tag{16}$$

Also, from (10) and Def. 37*(T4)* we know

$$\mathit{cls}(h, \iota) <: c' \tag{17}$$

By (3), (16), (17), (11) (9) and Lemma 15 we obtain

$$h' \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma \setminus [\![\mathbb{D}_c]\!]_{h,\iota} \tag{18}$$

By (10) and Lemma 11.1 we get

$$[\![\mathbb{D}_c]\!]_{h,\iota} \subseteq [\![\mathrm{a} \triangleright \mathbb{D}_c]\!]_{h,\sigma} \tag{19}$$

By (12) and Lemma 11.2 we get

$$[\![\mathrm{a} \triangleright \mathbb{D}_c]\!]_{h,\sigma} \subseteq [\![\mathbb{U}_{\sigma,c} \triangleright \mathbb{D}_c]\!]_{h,\sigma} \tag{20}$$

Since we assume that our verification technique $\mathcal{V}$ is well-structured, by 8*(S4)* we also get

$$[\![\mathbb{U}_{\sigma,c} \triangleright \mathbb{D}_c]\!]_{h,\sigma} \subseteq [\![\mathbb{V}_\sigma]\!]_{h,\sigma} \tag{21}$$

Thus, from (18), (19), (20), (21) and set inclusion transitivity we obtain

$$h' \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma$$

which by (8) and Def. 5 means we get the valid state

$$\mathrm{e}'_r \models_{\mathcal{V}} h'$$

rCall: From the conclusion and the premises of the rule we know

$$\mathrm{e}_r = \sigma\cdot\iota.m(v) \tag{22}$$
$$\mathrm{e}'_r = \sigma\cdot\sigma'\cdot\mathsf{call}\, e_b \tag{23}$$
$$\mathcal{B}(\mathit{cls}(h, \iota), m) = e_b,\ c \tag{24}$$
$$\sigma' = (\iota, v, c, m) \tag{25}$$
$$h' = h \tag{26}$$

From (22) we know that (5) could only have been derived using vd-call-2, and thus from the premises of this rule we get

$$h \vdash \sigma\cdot\iota : \mathrm{a}\, c' \tag{27}$$
$$\mathcal{B}(c', m) = _,\ c'' \tag{28}$$
$$h \models \mathbb{P}_{\sigma,\mathrm{a}},\ \sigma \tag{29}$$
$$\mathrm{a} \sqsubseteq \mathbb{C}_{\sigma,c'',m} \tag{30}$$
$$h \vdash_{\mathcal{V}} \sigma\cdot\iota \tag{31}$$
$$h \vdash_{\mathcal{V}} \sigma\cdot v \tag{32}$$

From (3), Def. 2*(W1)* and vs-class we know

$$e_b = e\, \mathsf{prv}\, \mathbb{E}_{c,m} \tag{33}$$
$$(c, m, _) \vdash_{\mathcal{V}} e \tag{34}$$

From (24) and Def. 35*(P2)* we know

$$\mathit{cls}(h, \iota) <: c \tag{35}$$

This allows us to deduce that h, σ' is well-formed *wrt.* the environment $(c, m, _)$ since by (25), (35) and Def. 36 we get

$$(c, m, _) \vdash_{\mathsf{wf}} h, \sigma' \tag{36}$$

By (34), (36) and Lemma 10 we derive

$$h \vdash_{\mathcal{V}} \sigma'\cdot e \tag{37}$$

Hence, by (37) and vd-start we get

$$h \vdash_{\mathcal{V}} \sigma\cdot\sigma'\cdot\mathsf{call}\, e\, \mathsf{prv}\, \mathbb{E}_{\sigma'}$$

and by (26), (23), (33) and (25) we obtain

$$h' \vdash_{\mathcal{V}} \mathrm{e}'_r$$

We still need to show that (6) reduces to a valid state, that is $\mathrm{e}'_r \models_{\mathcal{V}} h'$. From (27), Def. 37*(T4)* we know

$$\mathit{cls}(h, \iota) <: c' \tag{38}$$

and by (38), (28), (24), (1) and Def. 36*(F4)* we deduce

$$c <: c'' \tag{39}$$

Since $\mathcal{V}$ is well-structured, then by Def. 8*(S5)* and (39) we obtain

$$\mathbb{X}_{c,m} \subseteq \mathbb{X}_{c'',m} \tag{40}$$

which, by Lemma 11.2 yields

$$\mathrm{a} \triangleright \mathbb{X}_{c,m} \subseteq \mathrm{a} \triangleright \mathbb{X}_{c'',m} \tag{41}$$

Moreover from Def. 8*(S1)* and (30) we get

$$(\mathrm{a} \triangleright \mathbb{X}_{c'',m}) \setminus (\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \subseteq \mathbb{P}_{\sigma,\mathrm{a}} \tag{42}$$

From (41) and (42) we obtain

$$(\mathrm{a} \triangleright \mathbb{X}_{c,m}) \setminus (\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \subseteq \mathbb{P}_{\sigma,\mathrm{a}} \tag{43}$$

From (4) and Def. 5, and then from (29) we know

$$h \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma\ \cup\ \mathbb{P}_{\sigma,\mathrm{a}} \tag{44}$$

and from (43) and (44) we obtain

$$h \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma\ \cup\ (\mathrm{a} \triangleright \mathbb{X}_{c,m}) \tag{45}$$

From (25), (27) and Lemma 11.1 we know

$$[\![\mathbb{X}_{\sigma'}]\!]_{h,\sigma'} \subseteq [\![\mathrm{a} \triangleright \mathbb{X}_{c,m}]\!]_{h,\sigma} \tag{46}$$

and by (45) and (46) we obtain Def. 5*(V2)* and *(V3)*, *i.e.,*

$$h \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma\ \cup\ \mathbb{X}_{\sigma'} \tag{47}$$

Also by (25), (27), (30), (39) and Def. 4 we deduce Def. 5*(V1)*, *i.e.,*

$$h \vdash_{\mathcal{V}} \sigma\cdot\sigma' \tag{48}$$

and by (48), (47), Def. 5, and by (23) and (26) we get, as required,

$$e'_r \models_{\mathcal{V}} h' \tag{49}$$

rCxtFrame: From the conclusion and the premises of the rule we know

$$e_r = \sigma \cdot e^1_r \tag{50}$$
$$e'_r = \sigma \cdot e^2_r \tag{51}$$
$$e^1_r,\ h \longrightarrow e^2_r,\ h' \tag{52}$$

From (50) and (52)[1] we know that (5) could have been derived using either of the following three subcases:

1. vd-start: From the conclusion and premises of this rule we know

$$e^1_r = \sigma' \cdot \mathsf{call}\ e\ \mathsf{prv}\ \mathbb{E}_{\sigma'} \tag{53}$$
$$h \vdash_{\mathcal{V}} \sigma' \cdot e \tag{54}$$

As a result of (53), we know that (52) could have only been derived using either rLaunch or rLaunchEx. Moreover, because of (4), (50), (53), *i.e.*, $\sigma \cdot \sigma' \cdot \mathsf{call}\ e\ \mathsf{prv}\ \mathbb{E}_{\sigma'} \models_{\mathcal{V}} h$, and 5, we also know

$$h \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma \cup \mathbb{X}_{\sigma'} \tag{55}$$
$$h \vdash_{\mathcal{V}} \sigma \cdot \sigma' \tag{56}$$

Now (55), in particular $h \models \mathbb{X}_{\sigma'}$, rules out the use of rLaunchEx to derive (52). Thus, if (52) was derived using rLaunch, we know

$$e^2_r = \sigma' \cdot \mathsf{ret}\ e\ \mathsf{prv}\ \mathbb{E}_{\sigma'} \quad (\text{where } e'_r = \sigma \cdot e^2_r) \tag{57}$$
$$h' = h \tag{58}$$

By (54), (57), (58) and vd-frame we deduce

$$h' \vdash_{\mathcal{V}} e'_r$$

and by (55), (56), (57), (58) and the fact that $\mathcal{V}$ is well-structured we deduce

$$e'_r \models_{\mathcal{V}} h'$$

2. vd-frame: From the conclusion and premises of this rule we know

$$e^1_r = \sigma' \cdot \mathsf{ret}\ e^3_r\ \mathsf{prv}\ \mathbb{E}_{\sigma'} \tag{59}$$
$$h \vdash_{\mathcal{V}} \sigma' \cdot e^3_r \tag{60}$$

From (59), we know (52) could have been derived using either of the following 3 subcases:

(a) rCxtEval with

$$E[\cdot] = \mathsf{ret}\ [\cdot]\ \mathsf{prv}\ \mathbb{E}_{\sigma'} \tag{61}$$
$$\sigma' \cdot e^3_r,\ h \longrightarrow \sigma' \cdot e^4_r,\ h' \tag{62}$$
$$e^2_r = \sigma' \cdot E[e^4_r] \tag{63}$$

From (4), (50), (59), (61) and Lemma 14.1 and then Lemma 14.2 we obtain

$$\sigma' \cdot e^3_r \models_{\mathcal{V}} h \tag{64}$$
$$h \vdash_{\mathcal{V}} \sigma \cdot \sigma' \tag{65}$$
$$h \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma \setminus \mathbb{V}_{\sigma'} \setminus \mathbb{V}_{\mathit{stack}(e^3_r)} \tag{66}$$

Also, from (2), (50), (59), (61) and 37*(T6)* we know

$$h \vdash \sigma' \cdot e^3_r : t \tag{67}$$

Thus by (1), (67), (3), (64), (60), (62) and inductive hypothesis we infer

$$h' \vdash_{\mathcal{V}} \sigma' \cdot e^4_r \tag{68}$$
$$\sigma' \cdot e^4_r \models_{\mathcal{V}} h' \tag{69}$$

By (68), (61), (63), (51) and vd-frame we derive

$$h' \vdash_{\mathcal{V}} e'_r$$

By (69), (61), (63) and Lemma 14.1 we deduce

$$e^2_r \models_{\mathcal{V}} h' \tag{70}$$

From (66), (62) and Lemma 16 we get

$$h' \models \mathbb{X}_\sigma \setminus \mathbb{V}_\sigma \setminus \mathbb{V}_{\sigma'} \setminus \mathbb{V}_{\mathit{stack}(e^4_r)} \tag{71}$$

and by (69), (71), (63), (51) and Lemma 14.2 we obtain, as required,

$$e'_r \models_{\mathcal{V}} h'$$

(b) rCxtEval, rPrf with

$$E[\cdot] = \mathsf{ret}\ [\cdot] \text{ and } e^3_r = v \tag{72}$$
$$\sigma' \cdot v\ \mathsf{prv}\ \mathbb{E}_{\sigma'},\ h \longrightarrow \sigma' \cdot v,\ h \tag{73}$$
$$e^2_r = \sigma' \cdot v \text{ and } h' = h \tag{74}$$
$$h \models \mathbb{E}_{\sigma'},\ \sigma' \tag{75}$$

From (72), (74), (51), (60) and vd-end we obtain

$$h' \vdash_{\mathcal{V}} e'_r$$

Since e'_r is a visible state, Def. 5 requires us to prove that more invariants hold for the resultant state to be valid. From $\mathcal{V}$being well-structured, 8*(S2)* and (75) we deduce

$$h \models \mathbb{X}_{\sigma'} \cap \mathbb{V}_{\sigma'} \tag{76}$$

and by (74), (72), (50), (51), (59), (4) we know

$$h \models ((\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \cup \mathbb{X}_{\sigma'}) \setminus \mathbb{V}_{\sigma'} \tag{77}$$

And thus by (76), (77) and then Def. 5 we deduce $e'_r \models_{\mathcal{V}} h'$

(c) rCxtEval, rPrfEx with

$$E[\cdot] = \mathsf{ret}\ [\cdot] \text{ and } e^3_r = v$$
$$\sigma' \cdot v\ \mathsf{prv}\ \mathbb{E}_{\sigma'},\ h \longrightarrow \sigma' \cdot \mathsf{verfExc},\ h$$
$$e^2_r = \sigma' \cdot \mathsf{verfExc} \text{ and } h' = h$$
$$h \not\models \mathbb{E}_{\sigma'},\ \sigma'$$

This case is similar to the previous case and is left for the interested reader.

3. vd-end: from the conclusion and the premises of the rule, we obtain

$$e^1_r = \sigma' \cdot \mathsf{ret}\ v \tag{78}$$
$$h \vdash_{\mathcal{V}} \sigma' \cdot v \tag{79}$$

From (78) we know (52) could only have been derived using either rFrame or rFrameEx. However, from (4), (50) and (78) we know that e_r is in a visible state at σ' and thus

$$h \models \mathbb{X}_{\sigma'} \tag{80}$$

which rules out the possibility of using rFrameEx. Thus by rFrame we know

$$e^2_r = v \tag{81}$$
$$h' = h \tag{82}$$

By (51), (81), (82), (79) and Lemma 13.2 we obtain

$$h' \vdash_{\mathcal{V}} e'_r$$

[1] The fact that e^1_r reduces means that e^1_r contains at least one more stack frame, since all reduction rules in Fig. 4 are defined over runtime expressions of the form $\sigma \cdot e_r$.

From (50), (78), (4) and Def. 5, Def. 4 and Def. 3 we know

$$h \models ((\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \cup \mathbb{X}_{\sigma'}) \setminus \mathbb{V}_{\sigma'} \cup \mathbb{X}_{\sigma'} \quad (83)$$

$$\sigma' = (\iota, _, c', m),\ h \vdash \sigma{\cdot}\iota : \mathbb{a}\ _,\ c' <: c,\ \mathbb{a} \sqsubseteq \mathbb{C}_{\sigma,c,m} \quad (84)$$

We can rewrite (83) as

$$h \models (\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \setminus (\mathbb{V}_{\sigma'} \setminus \mathbb{X}_{\sigma'}) \cup \mathbb{X}_{\sigma'} \quad (85)$$

By (84) and Lemma 11.1 we have

$$[\![\mathbb{V}_{\sigma'} \setminus \mathbb{X}_{\sigma'}]\!]_{h,\sigma'} \subseteq [\![\mathbb{a} \triangleright (\mathbb{V}_{\sigma'} \setminus \mathbb{X}_{\sigma'})]\!]_{h,\sigma} \quad (86)$$

Also by (84) and Lemma 11.2 we have

$$\mathbb{a} \triangleright (\mathbb{V}_{\sigma'} \setminus \mathbb{X}_{\sigma'}) \ \subseteq\ \mathbb{C}_{\sigma,c,m} \triangleright (\mathbb{V}_{\sigma'} \setminus \mathbb{X}_{\sigma'}) \quad (87)$$

Since we assume our verification technique $\mathcal{V}$to be well-structured, then by 8*(S3)* we know

$$\mathbb{C}_{\sigma,c,m} \triangleright (\mathbb{V}_{c,m} \setminus \mathbb{X}_{c,m}) \ \subseteq\ \mathbb{V}_\sigma \quad (88)$$

and by *(S5)* and (84) we also know

$$(\mathbb{V}_{\sigma'} \setminus \mathbb{X}_{\sigma'}) \ \subseteq\ (\mathbb{V}_{c,m} \setminus \mathbb{X}_{c,m}) \quad (89)$$

and by (89) and Lemma 11.3 we get

$$\mathbb{C}_{\sigma,c,m} \triangleright (\mathbb{V}_{\sigma'} \setminus \mathbb{X}_{\sigma'}) \ \subseteq\ \mathbb{C}_{\sigma,c,m} \triangleright (\mathbb{V}_{c,m} \setminus \mathbb{X}_{c,m}) \quad (90)$$

By (86), (87), (90) and (88) we can rewrite (85) as

$$h \models (\mathbb{X}_\sigma \setminus \mathbb{V}_\sigma) \cup \mathbb{X}_{\sigma'} \quad (91)$$

and by (91), (82), (81) and (51) we obtain

$$e_r' \models_{\mathcal{V}} h'$$

as required.

□

5. Instantiations

In this section, we instantiate our framework to describe six verification techniques from the literature. We discuss modularity and compare their expressiveness. We also prove their soundness using Def. 8 and Theorem 9 from Sec. 4.

An optimal verification technique would allow maximal expressivity 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 $\mathbb{P}$ and $\mathbb{E}$). Obviously, these are contradictory goals, and some trade-offs need to be struck.

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.

5.1 Verification Techniques for Unstructured Heaps

Unstructured heap techniques make trade-offs by using information about classes, visibility, and access paths used in definitions of invariants. The instantiations are summarised in Fig. 8 whereby the keyword *all* denotes the set of all object invariants.

5.1.1 Poetzsch-Heffter

Poetzsch-Heffter [38] 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.

The absence of restrictions is reflected by the areas and regions needed to model Poetzsch-Heffter's technique. We define a singleton area set $\mathbf{A} = \{\mathsf{any}\}$ and a singleton region set $\mathbf{R} = \{\mathsf{any}\}$ with interpretations $[\![\mathsf{any}]\!]_{h,\iota} = \mathit{dom}(h)$ and $[\![\mathsf{any}]\!]_{h,\iota} = \mathit{all}$.

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

Figure 8. Verification techniques for unstructured heaps.

As shown in Fig. 8, this technique requires all invariants to hold in visible states. It does not restrict invariants; $\mathbb{D}$ allows a field update 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 method calls (to handle call-backs) and at the end of the method. Thus, Poetzsch-Heffter's technique is not modular.

5.1.2 Huizing & Kuiper

Huizing and Kuiper's technique [14] is almost as liberal as Poetzsch-Heffter's, but imposes fewer proof obligations. It achieves this by determining syntactically for each field the set of invariants that are potentially invalidated by updating the field. Proof obligations are imposed only for those vulnerable invariants.

We define the area set $\mathbf{A} = \{\mathsf{self}, \mathsf{any}\}$ with the interpretation $[\![\mathsf{self}]\!]_{h,\iota} = \{\iota\}$ and $[\![\mathsf{any}]\!]_{h,\iota} = \mathit{dom}(h)$. The area self is used to restrict the receivers of field updates to **this** (see Fig. 8). The concept of vulnerability is captured by the region set $\mathbf{R} = \{\mathsf{vul}\langle\mathbf{c}\rangle, \mathsf{any}\}$ with the following interpretation:

$[\![\mathsf{vul}\langle c\rangle]\!]_{h,\iota} =$ $\{(\iota', c') \mid$ the invariant of c' contains an expression **this** $.g_1 \ldots g_n.f$ $(n \geq 0)$ where $\mathcal{F}(c,f) = _,_ \wedge$ $\mathit{fld}(h, \mathit{fld}(h, \mathit{fld}(h, \iota', g_1), \ldots), g_n) = \iota\} \cup$ $\{(\iota, c') \mid \mathit{cls}(h,\iota) <: c'\}$

$[\![\mathsf{any}]\!]_{h,\iota} =$ *all*

Given an address ι and a class c, the set of vulnerable invariants contains the invariants of all client objects ι' of ι that refer to a field f of c via an access path $g_1 \ldots g_n$, as well as all invariants of ι. The interpretation shows that this technique inspects client invariants syntactically to determine whether they are vulnerable or not.

As shown in Fig. 8, this technique requires all invariants to hold in visible states. It does not restrict invariants; therefore, $\mathbb{D}$ describes exactly the set of vulnerable invariants. These invariants are vulnerable to each method and must be proven before method calls and at the end of each method.

Formalizing Huizing and Kuiper's technique in our framework reveals that it is very similar to Poetzsch-Heffter's. The main difference is that the former technique uses a syntactic analysis and restricts field updates to reduce proof obligations. However, this analysis is non-modular.

5.1.3 Leavens & Müller

Leavens and Müller [20] studied information hiding in interface specifications, based on the notion of visibility defined by access control of the programming language. For instance in Java, private field are visible only within their class. Their technique allows classes to declare several invariants and to specify the visibility of these invariants.

Since our formalization does not cover the visibility of fields and assumes exactly one invariant per class, we model a special case of Leavens and Müller's technique. We assume that all fields of a class have the same visibility. The predicate $\mathsf{visF}(c', c)$ yields

whether the fields declared in class c' are visible in class c. We assume that each class declares exactly one invariant and specifies its visibility. The predicate $\mathsf{visI}(c', c)$ yields whether the invariant declared in class c' is visible in class c. A generalization is possible, but does not provide any deeper insights.

We define the area set $\mathbf{A} = \{\mathsf{emp}, \mathsf{any}\}$ with the interpretation $[\![\mathsf{emp}]\!]_{h,\iota} = \emptyset$ and $[\![\mathsf{any}]\!]_{h,\iota} = dom(h)$. This technique permits field updates on arbitrary receivers as long as the field is visible in the method performing the update (see Fig. 8). Method calls are not restricted

The visibility of invariants is captured by the region set $\mathbf{R} = \{\mathsf{any}, \mathsf{self}\langle\mathbf{c}\rangle, \mathsf{any}\langle\mathbf{c}\rangle\}$ with the following interpretation:

$$[\![\mathsf{any}]\!]_{h,\iota} = all \qquad [\![\mathsf{any}\langle c\rangle]\!]_{h,\iota} = \{(\iota', c') \mid \mathsf{visI}(c', c)\}$$
$$[\![\mathsf{self}\langle c\rangle]\!]_{h,\iota} = \{(\iota, c) \mid \forall c'.\mathsf{visF}(c, c') \Leftrightarrow \mathsf{visI}(c, c')\}$$

$\mathbb{D}$ allows invariants to depend on fields of the same object declared in the same class, provided that the invariant is visible wherever the field is. This requirement enforces that any method that potentially breaks an invariant can see it and, thus, re-establish it. This requirement is very restrictive, as it disallows multi-object invariants and prevents invariants from depending on inherited fields.

The technique guarantees that only visible invariants are vulnerable; therefore, only visible invariants need to be proven before method calls and at the end of methods. It also supports helper methods, which we omit here for brevity

The visibility predicate visI is stronger than the usual definitions of the visibility predicate vis in Def. 7. Therefore, this technique is modular.

5.1.4 Comparison

We compare invariant restrictions, program restrictions, and proof obligations.

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 ($\mathbb{P}$ and $\mathbb{E}$). Both Poetzsch-Heffter and 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). This makes both techniques highly non-modular. Leavens and Müller's technique requires proof obligations only for visible invariants, which makes this technique modular.

Lemma 17 (Well-Structuredness of Verification Techniques for Unstructured Heaps)**.** *The Poetzsch-Heffter, Huizing and Kuiper and Leavens and Müller are well-structured.*

Proof. We here outline the proof for Poetzsch-Heffter and leave the proof of the remaining two for the interested reader. According to Def. 8, the components of the Poetzsch-Heffter verification technique, given earlier in Fig. 8, have to satisfy the following 5 criteria:

(S1): From $(\mathbb{a} \triangleright \mathbb{X}_{c',m'}) \setminus (\mathbb{X}_{c,m} \setminus \mathbb{V}_{c,m}) \subseteq \mathbb{P}_{c,m,\mathbb{a}}$ we get:

$$\mathsf{any} \triangleright \mathsf{any} \setminus (\mathsf{any} \setminus \mathsf{any}) \subseteq \mathsf{any}$$
$$\mathsf{any} \triangleright \mathsf{any} \setminus \emptyset \subseteq \mathsf{any}$$
$$\mathsf{any} \subseteq \mathsf{any}$$

(S2): From $\mathbb{V}_{c,m} \cap \mathbb{X}_{c,m} \subseteq \mathbb{E}_{c,m}$ we get:

$$\mathsf{any} \cap \mathsf{any} \subseteq \mathsf{any}$$
$$\mathsf{any} \subset \mathsf{any}$$

(S3): From $\mathbb{C}_{c,m,c',m'} \triangleright (\mathbb{V}_{c',m'} \setminus \mathbb{X}_{c',m'}) \subseteq \mathbb{V}_{c,m}$ we get:

$$\mathsf{any} \triangleright (\mathsf{any} \setminus \mathsf{any}) \subseteq \mathsf{any}$$
$$\mathsf{any} \triangleright \emptyset \subseteq \mathsf{any}$$
$$\emptyset \subseteq \mathsf{any}$$

(S4): From $\mathbb{U}_{c,m,c'} \triangleright \mathbb{D}_{c'} \subseteq \mathbb{V}_{c,m}$ we get:

$$\mathsf{any} \triangleright \mathsf{any} \subseteq \mathsf{any}$$
$$\mathsf{any} \subseteq \mathsf{any}$$

(S5): For $c <: c'$, from $\mathbb{X}_{c,m} \subseteq \mathbb{X}_{c',m}$ we get

$$\mathsf{any} \subseteq \mathsf{any}$$

and from $\mathbb{V}_{c,m} \setminus \mathbb{X}_{c,m} \subseteq \mathbb{V}_{c',m} \setminus \mathbb{X}_{c',m}$ we get

$$\mathsf{any} \setminus \mathsf{any} \subseteq \mathsf{any} \setminus \mathsf{any}$$

□

5.2 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. 9.

5.2.1 Müller *et al.*

Müller, Poetzsch-Heffter, and Leavens [34] present two techniques for multi-object invariants, called ownership technique and visibility technique (*OT* and *VT* for short). Both techniques utilise the hierarchic heap topology enforced by Universe types [8, 33]. 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.

Both *OT* and *VT* forbid 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* is formalised elegantly by using an ownership model where each object is owned by an object-class pair [23]. 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.

We assume a heap operation that yields the owner of an object in a heap: $ownr : \textsc{Hp} \times \textsc{Adr} \rightarrow \textsc{Adr} \times \textsc{Cls}$. The set of areas is:

$$\mathbb{a} \in \mathbf{A} ::= \mathsf{emp} \mid \mathsf{self} \mid \mathsf{rep}\langle\mathbf{c}\rangle \mid \mathsf{peer} \mid \mathsf{any} \mid \mathbb{a} \sqcup \mathbb{a}$$

with the following interpretation:

$$[\![\mathsf{self}]\!]_{h,\iota} = \{\iota\} \qquad [\![\mathsf{any}]\!]_{h,\iota} = dom(h) \qquad [\![\mathsf{emp}]\!]_{h,\iota} = \emptyset$$
$$[\![\mathsf{rep}\langle c\rangle]\!]_{h,\iota} = \{\iota' \mid ownr(h, \iota') = \iota\, c\}$$
$$[\![\mathsf{peer}]\!]_{h,\iota} = \{\iota' \mid ownr(h, \iota') = ownr(h, \iota)\}$$
$$[\![\mathbb{a}_1 \sqcup \mathbb{a}_2]\!]_{h,\iota} = [\![\mathbb{a}_2]\!]_{h,\iota} \cup [\![\mathbb{a}_2]\!]_{h,\iota}$$

These areas essentially reflect the ownership modifiers rep, peer, and any of Universe types. The rep area is parameterised by a class to express ownership by object-class pairs.

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

Figure 9. Verification techniques for structured heaps.

The two techniques require a rather rich set of regions to deal with the various aspects of ownership and subclassing:

$\mathbb{r} \in \mathbf{R} ::= \mathsf{emp} \mid \mathsf{self}\langle c\rangle \mid \mathsf{super}\langle c\rangle \mid \mathsf{peer}\langle c\rangle \mid \mathsf{rep} \mid \mathsf{own} \mid \mathsf{rep}^+ \mid \mathsf{own}^+ \mid \mathbb{r};\mathbb{r}$

with the following interpretations:

$$[\![\mathsf{emp}]\!]_{h,\iota} = \emptyset \qquad [\![\mathsf{self}\langle c\rangle]\!]_{h,\iota} = \{(\iota,c) \mid \ cls(h,\iota) <: c\}$$
$$[\![\mathsf{super}\langle c\rangle]\!]_{h,\iota} = \{(\iota,c') \mid \ c <: c'\}$$
$$[\![\mathsf{peer}\langle c\rangle]\!]_{h,\iota} = \{(\iota',c') \mid ownr(h,\iota') = ownr(h,\iota) \wedge \mathsf{vis}(c',c)\}$$
$$[\![\mathsf{rep}]\!]_{h,\iota} = \{(\iota',c') \mid ownr(h,\iota') = \iota\,_\} \qquad [\![\mathsf{own}]\!]_{h,\iota} = \{ownr(h,\iota)\}$$
$$[\![\mathbb{r}_1;\mathbb{r}_2]\!]_{h,\iota} = \textstyle\bigcup_{(\iota',c)\in[\![\mathbb{r}_1]\!]_{h,\iota}} [\![\mathbb{r}_2]\!]_{h,\iota'}$$
$$[\![\mathsf{rep}^+]\!]_{h,\iota} = [\![\mathsf{rep}]\!]_{h,\iota} \cup [\![\mathsf{rep};\mathsf{rep}^+]\!]_{h,\iota}$$
$$[\![\mathsf{own}^+]\!]_{h,\iota} = [\![\mathsf{own}]\!]_{h,\iota} \cup [\![\mathsf{own};\mathsf{own}^+]\!]_{h,\iota}$$

Here we exploit that owners and object invariants both are object-class pairs. Therefore, we can use the owner (o,c) of an object to denote the object invariant for object o declared in class c.

Ownership Technique. As shown in Fig. 9, *OT* requires that in visible states, all objects owned by the owner of **this** must satisfy their invariants, as expressed by $\mathbb{X}$.

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 ($\mathbb{P}$ and $\mathbb{E}$ are empty).

A non-pure 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, both *OT* and *VT* disallows calls on any references, as expressed by $\mathbb{C}$.

The proof obligations for method calls ($\mathbb{P}$) 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 to depend on fields of peer objects of a class c, provided that these invariants are visible in class c ($\mathbb{D}$). We denote the visibility of class c' from c as $\mathsf{vis}(c',c)$ and assume that visibility is reflexive, symmetric and 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, $\mathbb{V}$, $\mathbb{P}$ and $\mathbb{E}$ includes additionally the invariants of the peers of **this**. This addition is also reflected in the proof obligations before peer calls ($\mathbb{P}$) and before the end of a non-pure method ($\mathbb{E}$).

Both *OT* and *VT* are modular. In particular, verification of c does not require knowledge of c’s subclasses because subclass invariants must not depend on inherited fields.

Lemma 18. *OT and VT are well-structured.*

Proof. The proof assumes the following definition of area inclusion for the universe type system, defined as the least relation characterised by the rules below. It is not hard to see that this definition satisfies constraint *(P4)* of Def. 35. Other definitions for universe set inclusion are possible.

(u-emp)
$$\frac{}{\mathsf{emp} \sqsubseteq \mathbb{a}}$$

(u-any)
$$\frac{}{\mathbb{a} \sqsubseteq \mathsf{any}}$$

(u-self)
$$\frac{}{\mathsf{self} \sqsubseteq \mathsf{peer}}$$

(u-union)
$$\frac{}{\mathbb{a}_1 \sqsubseteq \mathbb{a}_1 \sqcup \mathbb{a}_2 \quad \mathbb{a}_2 \sqsubseteq \mathbb{a}_1 \sqcup \mathbb{a}_2}$$

(u-relf)
$$\frac{}{\mathbb{a} \sqsubseteq \mathbb{a}}$$

(u-trans)
$$\frac{\mathbb{a}_1 \sqsubseteq \mathbb{a}_2 \quad \mathbb{a}_2 \sqsubseteq \mathbb{a}_3}{\mathbb{a}_1 \sqsubseteq \mathbb{a}_3}$$

We start by showing that *OT* is well-structured from the components given in Fig. 9.

(S1): There are a number of areas that satisfy $\mathbb{a} \sqsubseteq \mathbb{C}_{c,m,c'm'}$. We here give the proof for the main two cases, *i.e.*, peer and $\mathsf{rep}\langle c\rangle$. For $\mathbb{a} = \mathsf{peer}$ we have:

$$(\mathsf{peer} \triangleright \mathsf{own}\,;\mathsf{rep}^+) \setminus (\mathsf{own}\,;\mathsf{rep}^+ \setminus \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \subseteq \mathsf{super}\langle c\rangle$$

When we adapt $\mathsf{own}\,;\mathsf{rep}^+$, *i.e.*, everything beneath the owner of the current receiver, by peer, *i.e.*, $\mathsf{peer} \triangleright \mathsf{own}\,;\mathsf{rep}^+$, we still

get $\mathsf{own}\,;\mathsf{rep}^+$ since peers share the same owner. Thus we get

$$\mathsf{own}\,;\mathsf{rep}^+ \setminus (\mathsf{own}\,;\mathsf{rep}^+ \setminus \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \subseteq \mathsf{super}\langle c\rangle$$

Using the set identity

$$A \setminus (B \setminus C) = (A \cap C) \cup (A \setminus B) \tag{92}$$

on the left hand of the inclusion we get:

$$(\mathsf{own}\,;\mathsf{rep}^+ \cap \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \cup (\mathsf{own}\,;\mathsf{rep}^+ \setminus \mathsf{own}\,;\mathsf{rep}^+) \subseteq \mathsf{super}\langle c\rangle$$

and thus

$$(\mathsf{own}\,;\mathsf{rep}^+ \cap \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \cup \emptyset \subseteq \mathsf{super}\langle c\rangle \tag{93}$$

From the interpretations given, we can show that

$$\mathsf{own}\,;\mathsf{rep}^+ \cap \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+ = \mathsf{super}\langle c\rangle$$

and as a result, from (93) we obtain

$$\mathsf{super}\langle c\rangle \subseteq \mathsf{super}\langle c\rangle$$

For the second case, *i.e.*, $\mathbb{a} = \mathsf{rep}\langle c\rangle$ we have

$$(\mathsf{rep}\langle c\rangle \triangleright \mathsf{own}\,;\mathsf{rep}^+) \setminus (\mathsf{own}\,;\mathsf{rep}^+ \setminus \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \subseteq \mathsf{emp}$$

When we adapt $\mathsf{own}\,;\mathsf{rep}^+$ by $\mathsf{rep}\langle c\rangle$ we get all objects transitively owned by the current receiver, namely rep^+ and thus we get

$$\mathsf{rep}^+ \setminus (\mathsf{own}\,;\mathsf{rep}^+ \setminus \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \subseteq \mathsf{emp}$$

At this point we apply the set identity (92) from the previous case and get

$$(\mathsf{rep}^+ \cap \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \cup (\mathsf{rep}^+ \setminus \mathsf{own}\,;\mathsf{rep}^+) \subseteq \mathsf{emp}$$

Since rep^+ does not include the current receiver, we know $\mathsf{rep}^+ \cap \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+ = \emptyset$. Also since $\mathsf{rep}^+ \subseteq \mathsf{own}\,;\mathsf{rep}^+$, we also know $\mathsf{rep}^+ \setminus \mathsf{own}\,;\mathsf{rep}^+ = \emptyset$ and hence we get

$$\emptyset \cup \emptyset \subseteq \mathsf{emp}$$

(S2): We have two cases, for pure and non-pure methods.

- If the method is pure, then $\mathbb{V}_{c,m} = \mathsf{emp}$ and $\mathbb{E}_{c,m} = \mathsf{emp}$ and thus we obtain

$$\mathsf{emp} \cap \mathsf{own}\,;\mathsf{rep}^+ \subseteq \mathsf{emp}$$

  Since the interpretation of emp is the emptyset, the above becomes trivially true.

- If the method is non-pure we have

$$\mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+ \cap \mathsf{own}\,;\mathsf{rep}^+ \subseteq \mathsf{super}\langle c\rangle \tag{94}$$

  From the interpretations, it is not hard to show directly that

$$\mathsf{super}\langle c\rangle \cap \mathsf{own}\,;\mathsf{rep}^+ = \mathsf{super}\langle c\rangle \tag{95}$$

$$\mathsf{own}^+ \cap \mathsf{own}\,;\mathsf{rep}^+ = \emptyset \tag{96}$$

  from which (94) follows.

(S3): Instantiating the components of Fig. 9 we obtain

$$\mathsf{rep}\langle c'\rangle \sqcup \mathsf{peer} \triangleright ((\mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \setminus \mathsf{own}\,;\mathsf{rep}^+) \subseteq \mathsf{super}\langle c'\rangle \sqcup \mathsf{own}^+$$

where we highlight the different roles played by the classes c and c' in the above statement. From the previous identities (95) and (96) we obtain

$$(\mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+) \setminus \mathsf{own}\,;\mathsf{rep}^+ = \mathsf{own}^+ \tag{97}$$

and thus

$$\mathsf{rep}\langle c'\rangle \sqcup \mathsf{peer} \triangleright (\mathsf{own}^+) \subseteq \mathsf{super}\langle c'\rangle \sqcup \mathsf{own}^+$$

From the interpretations we derive the identities:

$$\mathsf{rep}\langle c'\rangle \triangleright \mathsf{own}^+ = \mathsf{self}\langle c'\rangle \sqcup \mathsf{own}^+ \tag{98}$$

$$\mathsf{peer} \triangleright \mathsf{own}^+ = \mathsf{own}^+ \tag{99}$$

and thus, from the direct interpretation of $\sqcup$ we obtain

$$\mathsf{self}\langle c'\rangle \cup \mathsf{own}^+ \subseteq \mathsf{super}\langle c'\rangle \cup \mathsf{own}^+$$

which is immediately true since, from the interpretations we know $\mathsf{self}\langle c'\rangle \subseteq \mathsf{super}\langle c'\rangle$

(S4): Once again we have two cases.

- For pure methods we have

$$\mathsf{emp} \triangleright \mathsf{self}\langle c\rangle \sqcup \mathsf{own}^+ \subseteq \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+$$

  Since the interpretation of emp is $\emptyset$, anything adapted by the viewpoint emp given emp and thus

$$\mathsf{emp} \subseteq \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+$$

  which is trivially true.

- For non-pure methods we have

$$\mathsf{self} \triangleright \mathsf{self}\langle c\rangle \sqcup \mathsf{own}^+ \subseteq \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+$$

  Any adaptation by self acts as the identity and thus we obtain

$$\mathsf{self}\langle c\rangle \sqcup \mathsf{own}^+ \subseteq \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+$$

  which is true by the same reasons we gave for case **(S3)** above.

(S5): For $c <: c'$ we have to show

1. $\mathsf{own}\,;\mathsf{rep}^+ \subseteq \mathsf{own}\,;\mathsf{rep}^+$
2. $\begin{pmatrix} \mathsf{super}\langle c\rangle \sqcup \mathsf{own}^+ \setminus \\ \mathsf{own}\,;\mathsf{rep}^+ \end{pmatrix} \subseteq \begin{pmatrix} \mathsf{super}\langle c'\rangle \sqcup \mathsf{own}^+ \setminus \\ \mathsf{own}\,;\mathsf{rep}^+ \end{pmatrix}$

The first statement is trivially true whereas the second statement is true because (97) is true for any c, so substituting the right hand side of (97) gives us own^+ on both sides.

The proof of well-structuredness of the *VT* is similar to that of the *OT*:

(S1): Like before, the two cases for $\mathbb{a}$ we consider are peer and $\mathsf{rep}\langle c\rangle$. For peer we have the proof

$$\begin{array}{ll}
\mathsf{peer} \triangleright \mathsf{own}\,;\mathsf{rep}^+ \setminus \begin{pmatrix} \mathsf{own}\,;\mathsf{rep}^+ \setminus \\ \mathsf{peer}c \sqcup \mathsf{own}^+ \end{pmatrix} & \subseteq \mathsf{peer}\langle c\rangle \\
\mathsf{own}\,;\mathsf{rep}^+ \setminus \begin{pmatrix} \mathsf{own}\,;\mathsf{rep}^+ \setminus \\ \mathsf{peer}c \sqcup \mathsf{own}^+ \end{pmatrix} & \subseteq \mathsf{peer}\langle c\rangle \\
\begin{pmatrix} \mathsf{own}\,;\mathsf{rep}^+ \cap \\ \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ \end{pmatrix} \cup \begin{pmatrix} \mathsf{own}\,;\mathsf{rep}^+ \setminus \\ \mathsf{own}\,;\mathsf{rep}^+ \end{pmatrix} & \subseteq \mathsf{peer}\langle c\rangle \\
\begin{pmatrix} \mathsf{own}\,;\mathsf{rep}^+ \cap \\ \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ \end{pmatrix} \cup \emptyset & \subseteq \mathsf{peer}\langle c\rangle
\end{array}$$

Here we use the property

$$\mathsf{own}\,;\mathsf{rep}^+ \cap \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ = \mathsf{peer}\langle c\rangle \tag{100}$$

derived directly from the interpretations and get

$$\mathsf{peer}\langle c\rangle \cup \emptyset \subseteq \mathsf{peer}\langle c\rangle$$

For the case of $\mathsf{rep}\langle c\rangle$ we have the proof:

$$\begin{array}{ll} \mathsf{rep}\langle c\rangle \triangleright \mathsf{own}\,;\mathsf{rep}^+ \setminus \left(\begin{array}{c}\mathsf{own}\,;\mathsf{rep}^+\setminus\\ \mathsf{peer}c \sqcup \mathsf{own}^+\end{array}\right) & \subseteq \mathsf{emp}\\ \mathsf{rep}^+ \setminus \left(\begin{array}{c}\mathsf{own}\,;\mathsf{rep}^+\setminus\\ \mathsf{peer}c \sqcup \mathsf{own}^+\end{array}\right) & \subseteq \mathsf{emp}\\ (\mathsf{rep}^+ \cap \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+) \cup (\mathsf{rep}^+ \setminus \mathsf{own}\,;\mathsf{rep}^+) & \subseteq \mathsf{emp}\\ \emptyset \cup \emptyset & \subseteq \mathsf{emp}\end{array}$$

(S2): There are two cases.

- For pure methods we have

$$\begin{array}{r} \mathsf{emp} \cap \mathsf{own}\,;\mathsf{rep}^+ \sqsubseteq \mathsf{emp}\\ \mathsf{emp} \sqsubseteq \mathsf{emp}\end{array}$$

- For non-pure methods we have

$$\mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ \cap \mathsf{own}\,;\mathsf{rep}^+ \sqsubseteq \mathsf{peer}\langle c\rangle$$

  which is true from (100) earlier.

(S3): There are four cases to consider here, depending on the purity of the two methods. We here give the proof for two cases.

- If m is pure and m' is non-pure we have

$$\begin{array}{ll} \mathsf{emp} \triangleright (\mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+ \setminus \mathsf{own}\,;\mathsf{rep}^+) & \sqsubseteq \mathsf{emp}\\ \mathsf{emp} \sqsubseteq \mathsf{emp} & \end{array}$$

- When both m and m' are non-pure, then by (100) we have

$$\begin{array}{ll} \mathsf{rep}\langle c\rangle \sqcup \mathsf{peer} \triangleright \left(\begin{array}{c}\mathsf{peer}\langle c'\rangle \sqcup \mathsf{own}^+\\ \setminus \mathsf{own}\,;\mathsf{rep}^+\end{array}\right) & \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+\\ \mathsf{rep}\langle c\rangle \sqcup \mathsf{peer} \triangleright \mathsf{own}^+ & \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+\end{array}$$

  At this point we use the identities (98), (99) derived earlier to obtain

$$\mathsf{self}\langle c\rangle \sqcup \mathsf{own}^+ \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+$$

  which is true because we can show $\mathsf{self}\langle c\rangle \sqsubseteq \mathsf{peer}\langle c\rangle$.

(S4): We have two cases.

- If $\mathbb{U}_{c,m,c'} = \mathsf{emp}$ we have two of cases and here we consider the case where m is not pure (the other case is similar).

$$\begin{array}{r} \mathsf{emp} \triangleright \mathsf{peer}\langle c'\rangle \sqcup \mathsf{own}^+ \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+\\ \mathsf{emp} \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+\end{array}$$

- If $\mathbb{U}_{c,m,c'} = \mathsf{peer}$ then we know that m is not pure and that c is visible from c', *i.e.,* $\mathsf{vis}(c', c)$. We therefore obtain the proof

$$\begin{array}{r} \mathsf{peer} \triangleright \mathsf{peer}\langle c'\rangle \sqcup \mathsf{own}^+ \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+\\ \mathsf{peer}\langle c'\rangle \sqcup \mathsf{own}^+ \sqsubseteq \mathsf{peer}\langle c\rangle \sqcup \mathsf{own}^+\end{array}$$

  and by the symmetric property of $\mathsf{vis}(c', c)$ and the interpretation of $\mathsf{peer}\langle c\rangle$ we derive the identity $\mathsf{peer}\langle c\rangle = \mathsf{peer}\langle c'\rangle$ which make the above true.

(S5): This is similar to *(S5)* for *OT*.

□

5.2.2 Lu *et al.*

In [29], Lu, Potter, and Xue define Oval, a verification technique based on ownership types. The distinctive features of this technique are 1) invariant dependencies are restricted to object representations, defined by the ownership structure 2) standard and vulnerable invariants are specific to every methods in a class 3) before subcalls no proof obligations are used 4) the end of a method body proof obligation involves at most the current receiver object 5) subcalls and method overriding are regulated by "subcontracting".

Here we describe Oval', an adaptation of Oval, where *i*) we omit non-rep fields, a refinement whereby the invariant of the current object cannot depend on such fields (but its owners can), and *ii*) we drop the existential class parameter "*" annotation - both features enhance programming expressivity of Oval, but are deemed as non-central to our analysis. Oval' also used different restrictions for method overriding, because the original restrictions defined in Oval lead to unsoundness [28], as we discuss later on. In [29] description of the Oval verification technique is intertwined with that of the ownership type system it is based on. However, for the presentation of Oval', we strive to disentangle the two.

Oval' classes have owner parameters, indicated by X, Y, and the subclass relationship is described through a judgment $c\langle\overline{X}\rangle \lhd c'\langle\overline{X'}\rangle$ defined as:

$$\frac{\texttt{class}\ c\langle\overline{X}\rangle\ \texttt{extends}\ c\langle\overline{X}\rangle \ldots}{\begin{array}{c}c\langle\overline{X}\rangle \lhd c\langle\overline{X}\rangle\\ c\langle\overline{X}\rangle \lhd c'\langle\overline{X'}\rangle\end{array}} \qquad \frac{\begin{array}{c}c\langle\overline{X}\rangle \lhd c'\langle X'\rangle\\ c'\langle\overline{Y}\rangle \lhd c''\langle\overline{Y'}\rangle\end{array}}{c\langle\overline{X}\rangle \lhd c''\langle\{\overline{X'}/\bar{Y}\}\bar{Y}'\rangle}$$

where $\overline{X}$ are the disjoint formal class parameters of c in the program.

For simplicity we require that the formal class parameters are disjoint for every class. This assumption is very powerful, as, in contrast to usual systems, it allows the $\lhd$ relationship to be context independent.

An Oval program also defines an "inside" partial order relation, $\preceq$ for parameters of the same class.

Whenever $\mathcal{B}(c, m)$ is defined, Oval' programs also offer functions I and E from which method specific standard and vulnerable regions can be obtained

$$\begin{array}{ll} \mathsf{I} : \textsc{Cls} \times \textsc{Mthd} \rightharpoonup L & \mathsf{E} : \textsc{Cls} \times \textsc{Mthd} \rightharpoonup L\\ \text{where } L ::= \mathsf{top} \mid \mathsf{bot} \mid \mathsf{this} \mid X & K ::= \ L \mid K\,;\mathsf{rep}\end{array}$$

"Contexts" L, obtained from [29], are syntactic descriptions of the standard and vulnerable regions. As in [29], the type system extends L to K to described context abstraction[29], *i.e.,* objects owned by class parameters, and generalises the partial ordering $\preceq$ to K as a lattice bounded by top and bot, using rules from [29]. As in [29], the type system defines the judgement

$$c\langle\overline{K}\rangle <: c\langle\overline{K}\rangle \ \Leftrightarrow\ c\langle\overline{X}\rangle \lhd c'\langle\overline{X'}\rangle,\ \forall i. K'_i = \{\overline{K}/\bar{X}\}\bar{X}'$$

As in [29], the type system also requires all classes c and methods m to satisfy

$$\mathsf{I}(c,m) \preceq \mathsf{E}(c,m) \qquad \mathsf{I}(c,m) = \mathsf{E}(c,m) \Rightarrow \mathsf{I}(c,m) = \mathsf{this} \tag{101}$$

which guarantees that the expected and the vulnerable invariants of every method can intersect at most at the current object. Central to Oval is *subcontracting*, which we adopt from Oval'(modulo renaming)

Definition 19 (Subcontracting)**.**

$$SC(I, E, I', E', K) \Leftrightarrow \begin{cases} I \prec E \ \Rightarrow\ I' \preceq I\\ I = E \ \Rightarrow\ I' \prec I\\ I' \prec E' \ \Rightarrow\ E \preceq E'\\ I = E = this \ \Rightarrow\ E \preceq K\end{cases}$$

In Oval', subcontracting restricts subcalls, but it does *not* restrict methods overrides, as is the case from Oval, because it leads to unsoundness[28]. We revisit this point in Sec. 5.3. For Oval', we simply assume subclassing is governed by condition *(S5)* of Def. 8.

$$\frac{}{\textsf{emp} \sqsubseteq \mathbb{a}} \quad \frac{c\langle\overline{X}\rangle \lhd c\langle\overline{X'}\rangle}{c\langle\overline{K}\rangle \sqsubseteq c\langle\{\overline{K}/\overline{X}\}\overline{X'}\rangle} \quad \frac{}{\mathbb{a} \sqsubseteq \mathbb{a} \sqcup \mathbb{a}'}$$

$$\frac{}{K \sqsubseteq K;\textsf{rep}^*} \quad \frac{}{K \sqsubseteq K;\textsf{own}^*}$$

$$\frac{K \preceq K'}{K;\textsf{rep}^* \sqsubseteq K';\textsf{rep}^*} \quad \frac{K \preceq K'}{K';\textsf{own}^* \sqsubseteq K;\textsf{own}^*}$$

Figure 10. The $\sqsubseteq$ relation for Oval'

The heap model defines an additional operation *typ* which gives the runtime type of each object, $c\langle\bar{\iota}\rangle$ where:

$$typ(h,\iota) = c\langle\bar{\iota}\rangle \;\Rightarrow\; cls(h,\iota) = c,\; c\langle\overline{X}\rangle \lhd c'\langle\overline{X'}\rangle,\; |\bar{\iota}| = |\overline{X}|$$

The owner of ι above is ι_1. We define address runtime typing and address ownership as:

$$h \vdash \iota : c\langle\bar{\iota}\rangle \;\Leftrightarrow\; \begin{cases} typ(h,\iota) = c'\langle\overline{\iota'}\rangle,\; c'\langle\overline{X'}\rangle \lhd c\langle\overline{X}\rangle, \\ \forall i.\iota_i = \{\overline{\iota'}/\overline{X'}\}X_i \end{cases}$$

$$h \vdash \iota' \preceq \iota \;\Leftrightarrow\; typ(h,\iota') = c\langle\iota,\bar{\iota}\rangle$$

$$h \vdash \iota' \preceq^* \iota \;\Leftrightarrow\; \iota' = \iota \vee \exists\iota''.h \vdash \iota' \preceq \iota'',\, h \vdash \iota'' \preceq^* \iota$$

Definition 20. *Oval' areas and regions are defined as follows:*

$$\mathbb{a} \in \mathbf{A} ::= \textit{emp} \mid \textit{this} \mid c\langle\overline{K}\rangle \mid \mathbb{a} \sqcup \mathbb{a}$$
$$\mathbb{r} \in \mathbf{R} ::= \textit{emp} \mid K \mid K;\textit{rep}^* \mid K;\textit{own}^*$$

Remark 21. *Note, that our definition of areas introduces some redundancy, because a type* $t = \mathbb{a}\ c$ *would have the shape, e.g.,*`C<rep,o2> C`*. This redundancy is harmless.*

The interpretation for areas and regions is based on the interpretation of extended contexts:

$$\{\!\![\textsf{top}]\!\!\}_{h,\iota} = \{\!\![\textsf{bot}]\!\!\}_{h,\iota} = \emptyset \qquad \{\!\![\textsf{this}]\!\!\}_{h,\iota} = \{\iota\}$$
$$\{\!\![X]\!\!\}_{h,\iota} = \{\iota'_i \mid h \vdash \iota : c\langle\bar{\iota}\rangle,\; c\langle\overline{X}\rangle \lhd _,\; X = X_i\}$$
$$\{\!\![K;\textsf{rep}]\!\!\}_{h,\iota} = \{\iota' \mid \iota'' \in \{\!\![oEffK]\!\!\}_{h,\iota},\; h \vdash \iota' \preceq \iota''\}$$

The interpretation of areas is:

$$[\![\textsf{emp}]\!]_{h,\iota} = \emptyset \qquad [\![\textsf{this}]\!]_{h,\iota} = \{\iota\} \qquad [\![\mathbb{a} \sqcup \mathbb{a}']\!]_{h,\iota} = [\![\mathbb{a}]\!]_{h,\iota} \cup [\![\mathbb{a}']\!]_{h,\iota}$$
$$[\![c\langle\overline{K}\rangle]\!]_{h,\iota} = \{\iota' \mid h \vdash \iota' : c\langle\bar{\iota}\rangle, \forall i.\, \iota_i \in \{\!\![K_i]\!\!\}_{h,\iota}\}$$

The interpretation for regions is as follows:

$$[\![\textsf{emp}]\!]_{h,\iota} = [\![\textsf{top}]\!]_{h,\iota} = [\![\textsf{bot}]\!]_{h,\iota} = \emptyset$$
$$[\![K]\!]_{h,\iota} = \{(\iota',c) \mid \iota' \in \{\!\![K]\!\!\}_{h.\iota},\; cls(h,\iota') <: c\}$$
$$[\![K;\mathbb{r}]\!]_{h,\iota} = \begin{cases} \textit{all} & K = \textsf{top}, \mathbb{r} = \textsf{rep}^* \;\vee K = \textsf{bot}, \mathbb{r} = \textsf{own}^* \\ \bigcup_{(\iota',c) \in [\![K]\!]_{h,\iota}} [\![\mathbb{r}]\!]_{h,\iota'} & \mathbb{r} \in \{\textsf{rep}^*, \textsf{own}^*\} \end{cases}$$
$$[\![\textsf{rep}^*]\!]_{h,\iota} = \{\iota' \mid h \vdash \iota' \preceq^* \iota\} \qquad [\![\textsf{own}^*]\!]_{h,\iota} = \{\iota' \mid h \vdash \iota \preceq^* \iota'\}$$

Based on the ordering $\preceq$, we define the reflexive and transitive judgment $\sqsubseteq$ for areas and regions in Fig. 10. Based on the viewpoint type adaptation of the Oval type system[29] we define the "adaptation" operation **;** between areas and contexts *L*, returning extended contexts *K*:

$$\mathbb{a}\,;L = \begin{cases} L & \text{if } \mathbb{a} = \textsf{this}; \\ K_i & \text{if } \mathbb{a} = c\langle\overline{K}\rangle, L = X_i; \\ K_1\,;\textsf{rep} & \text{if } \mathbb{a} = c\langle\overline{K}\rangle, L = \textsf{this} \\ \bot & \text{otherwise.} \end{cases}$$

from which we define the viewpoint adaptation operation

$$\mathbb{a} \triangleright \mathbb{r} = \begin{cases} \textsf{emp} & \mathbb{a} = \textsf{emp} \vee \mathbb{r} = \textsf{emp} \\ \mathbb{r} & \mathbb{a} = \textsf{this} \\ \mathbb{r}_1 \sqcup \mathbb{r}_2 & \mathbb{a} = \mathbb{a}_1 \sqcup \mathbb{a}_2,\, \mathbb{r}_i = \mathbb{a}_i \triangleright \mathbb{r} \\ K_1;\textsf{rep} & \mathbb{a} = c\langle\overline{K}\rangle,\; \mathbb{r} = \textsf{this} \\ K_i & \mathbb{a} = c\langle\overline{K}\rangle,\; \mathbb{r} = X_i \\ (\mathbb{a};K);\mathbb{r}' & \mathbb{a} = c\langle\overline{K}\rangle,\; \mathbb{r} = K;\mathbb{r}',\; \mathbb{r}' \in \{\textsf{rep}, \textsf{rep}^*, \textsf{own}^*\} \end{cases}$$

Lemma 22. *Oval' is a programming language in the sense of definition 35. Also, Oval' has a sound type system in the sense of definition 37.*

Remark 23. *Note also, that usually in ownership type systems, and indeed in most systems with parameterized classes, the field and method lookup functions,* $\mathcal{F}$, $\mathcal{M}$ *and* $\mathcal{B}$ *are defined on types, rather than classes. For instance, one would expect to have* $\mathcal{F}(c\langle o1, o2\rangle, f)$ *rather than* $\mathcal{F}(c, f)$ *as in our framework. In contrast, in our framework, these functions are defined on classes. Namely, as we have requested the owner parameters to be disjoint across different classes, the meaning of. e.g.,,* $\mathcal{F}(c, f)$ *is, implicitly that of* $\mathcal{F}(c\langle c1, c2\rangle, f)$ *where* $c1, c2$ *are the formal ownership parameters of class* c.

Furthermore, in contrast to usual practice in ownership types, and parameterized classes, the type of an inherited field (or method) remains the same (as required in Def. 35, part F2 and F3 of Def. 36. Again, because the owner parameters are disjoint across classes, we can make this simplification. For example, for

```
class C<c1>{  A<c1> f; }
class D<d1> extends C<c1> { }
```

we would have that $\mathcal{F}(C, f) = \mathcal{F}(D, f) =$ `A<c1> A`.

Our framework does not require the underlying type system of the programming language to be expressed in terms of the functions $\mathcal{F}$ *and* $\mathcal{M}$. *Nevertheless, the underlying type system* could *be expressed in terms of these functions. For example, for field access, we would have the underlying type system rule:*

$$\frac{\Gamma \vdash e : \mathbb{a}\ c \qquad \mathcal{F}(c, f) = \mathbb{a}'\ c'}{\Gamma \vdash e.f : (\mathbb{a} \triangleright \mathbb{a}')\ c'}$$

where we define $\mathcal{R}$, *the owner parameter extraction function so that it extracts all owner parameters out of a context sequence, i.e.,* $\mathcal{R}(\textit{top}) = \mathcal{R}(\textit{bot}) = \mathcal{R}(\textit{this}) = \epsilon$, $\mathcal{R}(X) = X$, $\mathcal{R}(K;\textit{rep}) = \mathcal{R}(K)$, *and where* $\mathcal{R}(K, \overline{K}) = \mathcal{R}(K), \mathcal{R}(\overline{K})$, *and where the formal parameters of a class are defined through* $OP(c) = \overline{X}$ *iff class* c *has formal owner parameters* $\overline{X}$, *and where we define the area adaptation operator* $\triangleright$ *as follows:*

$$c\langle\overline{K}\rangle \triangleright c'\langle\overline{K'}\rangle = \begin{cases} c'\langle\overline{K'}\rangle & \textit{if } \mathcal{R}(\overline{K'}) = \epsilon \\ c'\langle[\overline{K}/\overline{X''}]\overline{K'}\rangle & \textit{if } c\langle\overline{X}\rangle \lhd c''\langle\overline{X''}\rangle \\ & \textit{and } \mathcal{R}(\overline{K'}) \subseteq OP(c'') \\ \bot & \textit{otherwise.} \end{cases}$$

For example `D<o3>` $\triangleright$ `A<c1> = A<o3>`.

We define owner extraction function $\mathcal{O}$ as follows

$$\mathcal{O}_{\mathbb{a},c} = \begin{cases} K_1, & \text{if } \mathbb{a} = c\langle\overline{K}\rangle \\ X_1, & \text{if } \mathbb{a} = \textsf{this},\; c\langle\overline{X}\rangle \lhd _ \\ \bot & \text{otherwise} \end{cases}$$

These functions are used to describe the Oval' verification technique, as shown in Fig. 9.

Lemma 24. *Oval' is well-structured.*

Proof. We use the shorthand $\mathsf{I} = \mathsf{I}(c, m)$, $\mathsf{E} = \mathsf{E}(c, m)$, $\mathsf{I}' = \mathsf{I}(c', m')$ and $\mathsf{E}' = \mathsf{E}(c', m')$ where we recall that they all come from the domain of *L*. We also use the following Lemmas:

Lemma 25. $K \prec K' \Rightarrow \begin{cases} K\,;\mathit{rep}^* \subseteq K'\,;\mathit{rep}^* \\ K'\,;\mathit{own}^* \subseteq K\,;\mathit{own}^* \\ K\,;\mathit{rep}^* \cap K'\,;\mathit{own}^* = \emptyset \end{cases}$

Lemma 26. *If* $\mathbb{a}; L \neq \bot$ *then* $\mathbb{a}; L = \mathbb{a} \triangleright L$

Lemma 27. $\mathit{this}\,;\mathit{rep}^* \cap \mathit{this}\,;\mathit{own}^* = \mathit{this}$

Lemma 28. $K \prec \mathit{this} \;\Rightarrow\; K\,;\mathit{rep}^* \subseteq (\mathit{this}\,;\mathit{rep}^* \setminus \mathit{this})$

(S1): We need to show

$$\mathbb{a} \triangleright \mathsf{I}'\,;\mathsf{rep}^* \setminus (\mathsf{I}\,;\mathsf{rep}^* \setminus \mathsf{E}\,;\mathsf{own}^*) \subseteq \mathsf{emp} \qquad (102)$$

If $\mathbb{a} \sqsubseteq \mathbb{P}_{c,m,c',m'}$ then by Fig. 9 we know

$$\mathsf{SC}(\mathsf{I}, \mathsf{E}, \mathbb{a}\,;\mathsf{I}', \mathbb{a}\,;\mathsf{E}', \mathcal{O}_{\mathbb{a},c}) \qquad (103)$$

and from (103) and Def. 19 we obtain two subcases

$\mathsf{I} \prec \mathsf{E}$**:** From this subcase's clause, *i.e.*, $\mathsf{I} \prec \mathsf{E}$, and Def. 19 we also know

$$\mathbb{a}; \mathsf{I}' \preceq \mathsf{I} \qquad (104)$$

and thus, since the ordering $\preceq$ is not defined for $\bot$ values, we conclude

$$\mathbb{a}; \mathsf{I}' \neq \bot \qquad (105)$$

From the subcase clause, $\mathsf{I} \prec \mathsf{E}$, and Lemma 25 we obtain

$$\mathsf{I}\,;\mathsf{rep}^* \setminus \mathsf{E}\,;\mathsf{own}^* = \mathsf{I}\,;\mathsf{rep}^*$$

and thus from (102) we get

$$\mathbb{a} \triangleright \mathsf{I}'\,;\mathsf{rep}^* \setminus \mathsf{I}\,;\mathsf{rep}^* \subseteq \mathsf{emp} \qquad (106)$$

From (105) and Lemma 26 we can rewrite (104) as $\mathbb{a} \triangleright \mathsf{I}' \preceq \mathsf{I}$ and by Lemma 25 we obtain

$$\mathbb{a} \triangleright \mathsf{I}'\,;\mathsf{rep}^* \subseteq \mathsf{I}\,;\mathsf{rep}^*$$

and thus $\mathbb{a} \triangleright \mathsf{I}'\,;\mathsf{rep}^* \setminus \mathsf{I}\,;\mathsf{rep}^* = \mathsf{emp}$ satisfying (106).

$\mathsf{I} = \mathsf{E} = \mathsf{this}$**:** Similar to the case before, from $\mathsf{I} = \mathsf{E}$, Def. 19 and Lemma 26 we get

$$\mathbb{a} \triangleright \mathsf{I}' \prec \mathsf{this} \qquad (107)$$

From the subcase clause, $\mathsf{I} = \mathsf{E} = \mathsf{this}$, and Lemma 27 we can derive

$$\mathsf{this}\,;\mathsf{rep}^* \setminus \mathsf{this}\,;\mathsf{own}^* = \mathsf{this}\,;\mathsf{rep}^* \setminus \mathsf{this}$$

and thus by (102) we obtain

$$\mathbb{a} \triangleright \mathsf{I}'\,;\mathsf{rep}^* \setminus (\mathsf{this}\,;\mathsf{rep}^* \setminus \mathsf{this}) \subseteq \mathsf{emp}$$

Finally, from (107) and Lemma 28 we derive that

$$\mathbb{a} \triangleright \mathsf{I}'\,;\mathsf{rep}^* \setminus (\mathsf{this}\,;\mathsf{rep}^* \setminus \mathsf{this}) = \mathsf{emp}$$

which satisfies the above.

(S2): Immediate from (101), Lemma 25 and Lemma 27.

(S3): We recall that

$$\mathbb{P}_{c,m,c',m'} = \sqcup \mathbb{a}_i \text{ such that } \mathsf{SC}(\mathsf{I}, \mathsf{E}, \mathbb{a}_i\,;\mathsf{I}', \mathbb{a}_i\,;\mathsf{E}', \mathcal{O}_{\mathbb{a}_i,c})$$

We here prove that for every such $\mathbb{a}_i$

$$\mathbb{a}_i \triangleright (\mathbb{V}_{c',m'} \setminus \mathbb{X}_{c',m'}) \subseteq \mathbb{V}_{c,m}$$

From which *(S3)* follows from the monotonicity of $\triangleright$. For this proof we find it convenient to distribute the adaptation in *(S3)* and show

$$\mathbb{a}_i \triangleright \mathbb{V}_{c',m'} \setminus \mathbb{a}_i \triangleright \mathbb{X}_{c',m'} \subseteq \mathbb{V}_{c,m} \qquad (108)$$

From the subcontract definition, we have two subcases:

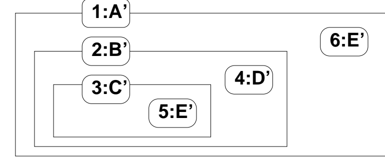

Figure 11. Heap h_0, with objects at addresses 1–6 belonging to indicated classes. Objects atop a box own those inside it. Assume that A' is a subclass of A and analogously for the other classes.

$\mathbb{a}_i\,;\mathsf{I}' \prec \mathbb{a}_i\,;\mathsf{E}'$**:** From the subcase clause $\mathbb{a}_i\,;\mathsf{I}' \prec \mathbb{a}_i\,;\mathsf{E}'$, Lemma 26 and Lemma 25 we deduce

$$\begin{aligned} \mathbb{a}_i \triangleright \mathbb{V}_{c',m'} \setminus \mathbb{a}_i \triangleright \mathbb{X}_{c',m'} & \\ &= \mathbb{a}_i \triangleright \mathsf{E}'\,;\mathsf{own}^* \setminus \mathbb{a}_i \triangleright \mathsf{I}'\,;\mathsf{rep}^* \\ &= \mathbb{a}_i \triangleright \mathsf{E}'\,;\mathsf{own}^* \end{aligned}$$

and thus from (108) we obtain

$$\mathbb{a}_i \triangleright \mathsf{E}'\,;\mathsf{own}^* \subset \mathsf{E}\,;\mathsf{own}^* \qquad (109)$$

From the subcase and Def. 19 we also know $\mathsf{E} \preceq \mathbb{a}_i\,;\mathsf{E}'$, thus by Lemma 26 we have $\mathsf{E} \preceq \mathbb{a}_i \triangleright \mathsf{E}'$ and hence by Lemma 25 we obtain (109) as required.

$\mathbb{a}_i\,;\mathsf{I}' = \mathbb{a}_i\,;\mathsf{E}' = \mathsf{this}$**:** From Lemma 27, Lemma 26 and (108) we obtain

$$\mathbb{a}_i \triangleright \mathsf{E}'\,;\mathsf{own}^* \setminus \mathsf{this} \subseteq \mathsf{E}\,;\mathsf{own}^* \qquad (110)$$

From the subcase and Def. 19 we also know $\mathsf{E} \preceq \mathcal{O}_{\mathbb{a}_i,c}$ which proves (110) as required.

(S4): By (101) we have tow subcases to consider:

$\mathsf{I} \prec \mathsf{E}$**:** From 9 we know $\mathbb{U}_{c,m,c'} = \mathsf{emp}$ thus we have the proof

$$\begin{aligned} \mathsf{emp} \triangleright (\mathsf{this}\,;\mathsf{own}^*) &\subseteq \mathsf{E}\,;\mathsf{own}^* \\ \mathsf{emp} &\subseteq \mathsf{E}\,;\mathsf{own}^* \end{aligned}$$

$\mathsf{I} = \mathsf{E} = \mathsf{this}$**:** From 9 we know $\mathbb{U}_{c,m,c'} = \mathsf{this}$ thus we have the proof

$$\begin{aligned} \mathsf{this} \triangleright (\mathsf{this}\,;\mathsf{own}^*) &\subseteq \mathsf{this}\,;\mathsf{own}^* \\ \mathsf{this}\,;\mathsf{own}^* &\subseteq \mathsf{this}\,;\mathsf{own}^* \end{aligned}$$

(S5): Immediate from the assumptions of Oval'.

□

5.2.3 Comparison

We illustrate differences between the techniques for structured heaps using the heap h_0 in Fig. 11. Fig. 12 shows the values of the components of the three techniques for class C and object 3.

OT and *VT* require knowledge of the class at which on object is owned; this information is shown in the last row of Fig. 12. For Oval, the methods have I and E as given in the last row.

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

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

Figure 12. Comparison of techniques for structured heaps; differences are highlighted in grey.

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* ($\mathbb{P}$ *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 $\mathbb{P}$ and $\mathbb{E}$.

Oval imposes end-of-body proof obligation only when I and E are the same (i.e. m1). Since Oval 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. *OT* and *VT* disallow such dependencies and their proof obligations do not include (3,C'). This restriction is important for modularity.[2] Oval never impose proof obligations before method calls ($\mathbb{P}$ is empty), and prevents potentially dangerous call-backs through the subcontract requirement.

5.3 Soundness of Verification Techniques

Instead of proving soundness for every single verification technique discussed in this section, Theorem 9 reduces this complex task to merely checking that the seven components of every instantiations satisfy the five (fairly simple) well-structured conditions of Def. 8. Assuming that the underlying type system is sound, once we show well-structuredness for a technique, verification technique soundness (Def. 6) follows.

Lemma 29 (Type System Soundness for Universes)**.** *The Universes Type System satisfies Def. 37.*

[2] The Oval developers plan to solve this modularity problem by requiring that any inherited method has to be re-verified in the subclass [28].

Proof. The typing rules together with the soundness proof for the Universes type system has already been given in [6] bar the rules for the (novel) construct $e\,\mathsf{prv}\,\mathbb{r}$ and the exceptions verfExc and fatalExc. The typing of the proof annotation construct however depends exclusively on the typing of the subexpression e; typically this construct would be typechecked using a rule such as the one shown below. Also, the type system should typecheck exceptions related to the verification technique as shown below.

$$\frac{\Gamma \vdash e : t}{\Gamma \vdash e\,\mathsf{prv}\,\mathbb{r} : t} \qquad \frac{}{\begin{array}{c}\Gamma \vdash \mathsf{verfExc} : t\\ \Gamma \vdash \mathsf{fatalExc} : t\end{array}}$$

With these additions, it is not hard to check that the type system satisfies the requirements set out by Def. 37. ☐

Lemma 30 (Type System Soundness for Oval')**.** *The Oval' Type System satisfies Def. 37.*

Proof. From [29]. ☐

Corollary 31. *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.*

Proof. Immediate from Theorem 9, Lemmas 29 and 30, and Lemmas 17, 18, 24. ☐

Soundness of Oval. The original Oval proposal [29] requires subcontracting for method overriding, using the analogy between dynamic dispatch and that of the superclass's overridden method making a subcall to the overridden method in the subclass. This analogy however is misleading because a subcalls contains a proof at the end of the method body of the superclass. This proof does not appear in dynamic dispatch and as a result, the overridden method would be allowed to have more residue vulnerable invariants. More precisely, the Oval subcontract requirement for overriding methods 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)* of Def. 6, and makes the system unsound. The following counter example elucidates this unsoundness.

```
class D<o> {
  C1<this> c = new C2<this>();
  void m() <this,o> { c.mm() }
}
class C1<o>{
  void mm() <this,this> {...}
}
class C2<o> extends C2<o> {
  void mm() <bot,this> {...}
}
```

The above example typechecks and is well-verified according to the rules given in [29][3]. More specifically, according to the rule [EXP-CAL], c.mm() is a valid subcall for m() of D because the contract of mm() in C1, *i.e.*, <**this**, **this**> is a subcontract of <**this** ,o>, the contract of m() in D. Moreover, C2 is a valid subclass of C1 because the rule [METHOD] in [29] ensures that the overridden method mm() of C2 satisfies subcontracting for mm() of C1 (*i.e.*, <bot,**this**> is a subcontract of <**this**, **this**>). At runtime, m() of D calls c.mm() under the assumption that c, owned by **this**, will be valid after the subcall returns since the contract <**this**, **this**> means that **this** may be violated during the call but re-established at the end of the call. However, the dynamically dispatched mm() of C2 is allowed to break the invariant of c without having to reestablish them, which clearly violates the contract of m() in D.

We discovered this unsoundness by noticing the discrepancy with *(S5)*; we contacted the authors [28] who confirmed this, having independently discovered the error, and applied the same fix as in Oval'.

6. Related Work

In this section, we discuss related work other than the verification techniques covered in Sec. 5.

The idea of areas and regions is inspired from type and effects systems[42], which have been extremely widely applied, *e.g.,* to support race-free programs and atomicity [10], or restricted effect of computations on predicates [5, 41, 40].

Object invariants trace back to Hoare's implementation invariants [12] and monitor invariants [13]. They were popularised in object-oriented programming by Meyer [30]. Their work, as well as other early work on object invariants [25, 26] 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 [34] for a detailed discussion). However, once the requirements are clear, a formalization within our framework seems straightforward.

The verification techniques based on the Boogie methodology [1, 3, 23, 24] do not use a visible state semantics. Instead, each method specifies in its precondition which invariants it requires. This fine-grained specification is beyond the expressiveness of our type-based framework. However, the Spec# language [2] encourages a stylised usage of the Boogie methodology which is very similar to the ownership technique [34] and which can be formalised in our framework.

We only know of one verification technique based on visible states, which cannot be expressed in our framework. The work by Middelkoop et al. [32] 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.

[3] We recall that in [29], the type system and the verification technique are given as one set of static rules as opposed to the disentangling advocated by our work.

Some verification techniques exclude the pre- and post-states of so-called helper methods from the visible states [20, 21]. Helper methods can easily be expressed in our framework by choosing different parameters for helper and non-helper methods. For instance in JML, $\mathbb{X}$, $\mathbb{P}$, and $\mathbb{E}$ are empty for helper methods, because they neither assume nor are they required 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 easily model strong invariants, essentially by preventing them from occurring in $\mathbb{V}$.

Existing techniques for visible state invariants have only limited support for object initialization. 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 monotonous invariants, which, once established, are never broken. Modeling delayed types in our framework is future work.

Even though separation logic [15, 39] has been used to reason about invariants of modules with one instance [35], object invariants are not as important as in other verification techniques. Instead, verifiers are encouraged to write predicates to express consistency criteria [36]. Abstract predicate families [37] allow one to do so without violating abstraction and information hiding. As for the Boogie methodology, 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. The framework advocates for the separation between verification concerns from type system concerns. Thus, 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 generic conditions on the framework components that guarantee soundness and modularity, and we proved a universal soundness theorem.

A unified framework with the above separation of concerns, offers, in our opinion, three important advantages over several independent formalisms. First, it allows a simpler understanding of the verification mechanisms (under the assumption that the type system is sound). Second, it facilitates comparisons between such verification mechanisms. Third, we found that checking the well-structured conditions of a verification technique is significantly simpler than developing soundness proofs from scratch. We are confident that our framework will simplify the development of further verification techniques.

As future work, we want to combine the flexibility of heap topology offered by ownership types [27] with the more liberal conditions on regions affecting invariants from [34]. We also plan to extend our framework to techniques that do not use visible state semantics. For instance, verification techniques for concurrent programs [16] check invariants at the end of lock blocks, and the Boogie methodology uses designated expose blocks to delimit code in which an invariant need not hold [1]. The verification of frame properties, in particular in the presence of model fields, is very similar to the verification of invariants [22, 33]. An interesting direction for future work is to extend our framework to frame properties.

Acknowledgments

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.

We want to thank Rustan Leino for suggestions on notation, Ronald Middlekoop for discussion of related work, the Imperial College as well as the Cambridge University programming languages group for suggestions on presentation, and Yi Lu and John Potter for many discussions about Oval.

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 st ate. 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] Dave Clarke and Sophia Drossopoulou. Ownership, Encapsulation and the Disjointness of Types and Effects. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications (OOPSLA'02)*, pages 292–310, Seattle, Washington, USA, November 2002. ACM Press.

[6] Dave Cunningham, Werner Dietl, Sophia Drossopoulou, Adrian Francalanza, and Peter Mueller. UJ: Type Soundness and Encapsulation for Universe Types. Unpublished report available from `www.doc.ic.ac.uk/ scd/framework/longpaper`.

[7] 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.

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

[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 Flanagan and Shaz Qadeer. Types for atomicity. In *TLDI '03: Proceedings of the 2003 ACM SIGPLAN international workshop on Types in languages design and implementation*, pages 1–12, New York, NY, USA, 2003. ACM Press.

[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] B. Jacobs, K. R. M. Leino, F. Piessens, and W. Schulte. Safe concurrency for aggregate objects with invariants. In *Software Engineering and Formal Methods (SEFM)*, pages 137–146. IEEE Computer Society, 2005.

[17] K. D. Jones. LM3: A Larch interface language for Modula-3: A definiti on and introduction. Technical Report 72, Digital Equipment Corporation, Systems Research Center, 1991.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] K. R. M. Leino. *Toward Reliable Modular Programs*. PhD thesis, California Institute of Technology, 1995.

[23] 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.

[24] 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.

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

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

[27] Y. Lu and J. Potter. Protecting Representation with Effect Encapsulation. In *POPL*, pages 359–371, 2006.

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

[29] 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.

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

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

[32] 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.

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

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

[35] 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.

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

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

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

[39] 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.

[40] M. Smith. *A Model of Effects with an Application to Ownership Types*. PhD thesis, Imperial College, 2006.

[41] Matthew Smith and Sophia Drossopoulou. Cheaper Reasoning with Ownership Types. In *ECOOP International Workshop on Aliasing, Confinement and Ownership (IWACO 2003)*, 2003.

[42] M. Tofte and J.P. Talpin. Region-based memory management. *Information and Computation*, 1997.

A. Appendix

A.1 Language Definitions

Definition 32. *A runtime structure is a tuple*

$$\text{RSTRUCT} = (\text{HP}, \text{ADR}, \simeq, \preceq, dom, cls, fld, upd, new)$$

where HP*, and* ADR *are sets, and where*

$$\begin{array}{lll} \simeq & \subseteq \text{HP} \times \text{HP} & \preceq \subseteq \text{HP} \times \text{HP} \\ dom & : \text{HP} \to \mathcal{P}(\text{ADR}) & \\ cls & : \text{HP} \times \text{ADR} \rightharpoonup \text{CLS} & \\ fld & : \text{HP} \times \text{ADR} \times \text{FLD} \rightharpoonup \text{VAL} & \\ upd & : \text{HP} \times \text{ADR} \times \text{FLD} \times \text{VAL} \to \text{HP} & \\ new & : \text{HP} \times \text{ADR} \times \text{TYP} \to \text{HP} \times \text{ADR} & \end{array}$$

where $\text{VAL} = \text{ADR} \cup \{null\}$ *for some element* $null \notin \text{ADR}$*. For all* $h \in \text{HP}$, $\iota, \iota' \in \text{ADR}$, $v \in \text{VAL}$*, we require:*

(H1) $\iota \in dom(h) \Rightarrow \exists c.cls(h, \iota) = c$

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

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

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

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

Definition 33. *An* area/region structure *is a tuple*

$$\text{ASTRUCT} = (\mathbf{A}, \mathbf{R}, \triangleright)$$

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

$$\triangleright : \mathbf{A} \times \mathbf{R} \to \mathbf{R}$$

Definition 34. $E[\cdot]$ *and* $F[\cdot]$ *are defined as follows:*

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

Definition 35. *A programming language is a tuple*

$$PL = (\text{PRG}, \text{RSTRUCT}, \text{ASTRUCT})$$

where PRG *is a set where every* $P \in \text{PRG}$ *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:

$$\begin{array}{lll} \mathcal{F} & : & \text{CLS} \times \text{FLD} \rightharpoonup \text{TYP} \times \text{CLS} \\ \mathcal{M} & : & \text{CLS} \times \text{MTHD} \rightharpoonup \text{TYP} \times \text{TYP} \\ \mathcal{B} & : & \text{CLS} \times \text{MTHD} \rightharpoonup \text{EXPR} \times \text{CLS} \\ <: & \subseteq & \text{CLS} \times \text{CLS} \cup \text{TYP} \times \text{TYP} \\ \sqsubseteq & \subseteq & \mathbf{A} \times \mathbf{A} \\ [\![\cdot]\!] & : & \mathbf{A} \times \text{HP} \times \text{ADR} \to \mathcal{P}(\text{ADR}) \\ [\![\cdot]\!] & : & \mathbf{R} \times \text{HP} \times \text{ADR} \to \mathcal{P}(\text{ADR} \times \text{CLS}) \\ \models & \subseteq & \text{HP} \times \text{ADR} \times \text{CLS} \\ \vdash & \subseteq & (\text{ENV} \times \text{EXPR} \cup \text{HP} \times \text{STK} \times \text{REXPR}) \times \text{TYP} \end{array}$$

$$\text{(vd-null)}\ \frac{}{h \vdash_{\mathcal{V}} \sigma \cdot \mathsf{null}} \qquad \text{(vd-addr)}\ \frac{\iota \in dom(h)}{h \vdash_{\mathcal{V}} \sigma \cdot \iota} \qquad \text{(vd-new)}\ \frac{}{h \vdash_{\mathcal{V}} \sigma \cdot \mathsf{new}\, t}$$

$$\text{(vd-Var)}\ \frac{}{h \vdash_{\mathcal{V}} \sigma \cdot x} \qquad \text{(vd-this)}\ \frac{}{h \vdash_{\mathcal{V}} \sigma \cdot \mathsf{this}} \qquad \text{(vd-verEx)}\ \frac{}{h \vdash_{\mathcal{V}} F[\mathsf{verfExc}]}$$

$$\text{(vd-ass)}\ \frac{\begin{array}{l} h \vdash \sigma \cdot \mathsf{e}_r : \mathbb{a}\, c' \\ \mathcal{F}(c', f) = _, c \\ \mathbb{a} \sqsubseteq \mathbb{U}_{\sigma,c} \\ h \vdash_{\mathcal{V}} \sigma \cdot \mathsf{e}_r \\ h \vdash_{\mathcal{V}} \sigma \cdot \mathsf{e}'_r \end{array}}{h \vdash_{\mathcal{V}} \sigma \cdot \mathsf{e}_r.f := \mathsf{e}'_r} \qquad \text{(vd-fld)}\ \frac{h \vdash_{\mathcal{V}} \sigma \cdot \mathsf{e}_r}{h \vdash_{\mathcal{V}} \sigma \cdot \mathsf{e}_r.f} \qquad \text{(vd-end)}\ \frac{h \vdash_{\mathcal{V}} \sigma' \cdot v}{h \vdash_{\mathcal{V}} \sigma \cdot \sigma' \cdot \mathsf{ret}\, v}$$

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

$$\text{(vd-start)}\ \frac{h \vdash_{\mathcal{V}} \sigma' \cdot e}{h \vdash_{\mathcal{V}} \sigma \cdot \sigma' \cdot \mathsf{call}\, e\ \mathsf{prv}\ \mathbb{E}_{\sigma'}} \qquad \text{(vd-frame)}\ \frac{h \vdash_{\mathcal{V}} \sigma' \cdot \mathsf{e}_r}{h \vdash_{\mathcal{V}} \sigma \cdot \sigma' \cdot \mathsf{ret}\ \mathsf{e}_r\ \mathsf{prv}\ \mathbb{E}_{\sigma'}}$$

Figure 13. Well-annotated runtime expressions.

where every $P \in \text{PRG}$ *must satisfy the constraints:*

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

Definition 36. *For every program, the judgement:*

$$\vdash_{\mathit{wf}} : (\text{HP} \times \text{STK} \times \text{STK} \times \mathbf{A}) \cup (\text{ENV} \times \text{HP} \times \text{STK}) \cup \text{PRG}$$

is defined as:

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

The judgement $h, \sigma \vdash_{\mathbf{wf}} \sigma' : \mathbb{a}$ expresses that the receiver of σ' is within $\mathbb{a}$ as seen from the point of view of σ. $\Gamma \vdash_{\mathbf{wf}} h, \sigma$ expresses 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 37. *A programming language* PL *has a* sound type system *if all programs* $P \in \text{PL}$ *satisfy the constraints:*

(T1) $\Gamma \vdash e : t,\;\; t <: t' \;\Rightarrow \Gamma \vdash e : t'$

(T2) $h \vdash \mathsf{e}_r : t,\;\; t <: t' \;\Rightarrow h \vdash \mathsf{e}_r : t'$

(T3) $h \vdash \mathsf{e}_r : t,\;\; h \preceq h' \;\Rightarrow h' \vdash \mathsf{e}_r : t$

(T4) $h \vdash \sigma{\cdot}\iota : _\, c \;\Rightarrow \mathit{cls}(h, \iota) <: c$

(T5) $h \vdash \sigma{\cdot}\iota.m(v) : t \;\Rightarrow \begin{cases} h \vdash \sigma{\cdot}\iota : \mathbb{a}\, c \\ \mathcal{M}(c, m) = t', t \\ h \vdash \sigma{\cdot}v : t' \end{cases}$

(T6) $h \vdash \sigma{\cdot}\sigma'{\cdot}\,\mathsf{ret}\,\mathsf{e}_r\;\mathsf{prv}\,\mathbb{r} : t \;\;\Rightarrow\;\; h \vdash \sigma'{\cdot}\mathsf{e}_r : t$

(T7) $\sigma = (\iota, _, _, _),\; h \vdash \sigma{\cdot}\iota' : \mathbb{a}\,_ \;\Rightarrow \iota' \in [\![\mathbb{a}]\!]_{h,\iota}$

(T8) $\Gamma \vdash e : \mathbb{a}\,c,\;\; \Gamma \vdash h, \sigma \;\Rightarrow h, \sigma \vdash e : \mathbb{a}\,c$

(T9) $\forall \mathbb{X}.\;\; \left.\begin{array}{l} \vdash P,\;\; h \vdash \mathsf{e}_r : t \\ \mathsf{e}_r,\, h \longrightarrow \mathsf{e}'_r,\, h' \end{array}\right\} \Rightarrow h' \vdash \mathsf{e}'_r : t$

(T1) and *(T2)* express subsumption. *(T3)* states that runtime expression typing does not depend on the field values assigned in the heap. *(T4)* states that addresses are typed according to their class in the heap. *(T5)* and *(T6)* are a technical constraint stating that method call typing implies that the parameter type and return type set by $\mathcal{M}$ for that method are respected and that proof obligations do not interfere with typing. *(T7)* states that 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. The most important constraints are *(T8)* and *(T9)*: *(T8)* states the correspondence between typing source expressions and runtime expressions for heaps and stack frames that respect the typing environment; *(T9)* states that for all well-formed programs, reduction preserves typing.