A Unified Framework for Verification Techniques for Object Invariants

S. Drossopoulou[(1)], A. Francalanza[(2)], P. Müller[(3)], and A. J. Summers[(1)]

[(1)] Imperial College London,
[(2)] University of Southampton,
[(3)] Microsoft Research, Redmond

Abstract. Object invariants define the consistency of objects. They have subtle semantics because of call-backs, multi-object invariants and subclassing. Several visible-state verification techniques for object invariants have been proposed. It is difficult to compare these techniques and ascertain their soundness because of differences in restrictions on programs and invariants, in the use of advanced type systems (*e.g.*, ownership types), in the meaning of invariants, and in proof obligations.
We develop a unified framework for such techniques. We distil seven parameters that characterise a verification technique, and identify sufficient conditions on these parameters which guarantee soundness. We instantiate our framework with three verification techniques from the literature, and use it to assess soundness and compare expressiveness.

1 Introduction

Object invariants play a crucial role in the verification of object-oriented programs, and have been an integral part of all major contract languages such as Eiffel [25], the Java Modeling Language JML [17], and Spec# [2]. Object invariants express consistency criteria for objects, ranging 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).

While the basic idea of object invariants is simple, verification techniques for practical OO-programs face challenges. These challenges are made more daunting by the common expectation that classes should be 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 o in an inconsistent state.

Multi-object invariants: When the invariant of an object p depends on the state of another object o, modifications of o potentially break the invariant of p. In particular, when verifying o, the invariant of p may not be known and, if not, cannot be expected to be preserved.

Subclassing: When the invariant of a subclass D refers to fields declared in a superclass C then methods of C can break D's invariant by assigning to these fields. In particular, when verifying a class, its subclass invariants are not known in general, and so cannot be expected to be preserved.

Several verification techniques address some or all of these challenges [1, 3, 14, 16, 18, 23, 26, 27, 31]. They share many commonalities, but differ in the following important aspects:

1. *Invariant semantics:* Which invariants are expected to hold when?
2. *Invariant restrictions:* Which objects may invariants depend on?
3. *Proof obligations:* What proofs are required, and where?
4. *Program restrictions:* Which objects' methods/fields may be called/updated?
5. *Type systems:* What syntactic information is used for reasoning?
6. *Specification languages:* What syntax is used to express invariants?
7. *Verification logics:* How are invariants proved?

These differences, together with the fact that most verification techniques are not formally specified, complicate the comparison of verification techniques, and hinder the understanding of why these techniques satisfy claimed properties such as soundness. For these reasons, it is hard to decide which technique to adopt, or to develop new sound techniques.

In this paper, we present a unified framework for verification techniques for object invariants. This framework formalises verification techniques in terms of seven parameters, which abstract away from differences pertaining to language features (type system, specification language, and logics) and highlight characteristics intrinsic to the techniques, thereby aiding comparisons. Subsets of these parameters describe aspects applicable to all verification techniques; for example, a generic definition of *soundness* is given in terms of two framework parameters, expressivity is captured by three other parameters.

We concentrate on techniques that require invariants to hold in the pre-state and post-state of a method execution (often referred to as *visible states* [27]) while temporary violations between visible states are permitted. These techniques constitute the vast majority of those described in the literature.

Contributions. The contributions of this paper are:

1. We present a unified formalism for object invariant verification techniques.
2. We identify conditions on the framework that guarantee soundness of a verification technique.
3. We separate type system concerns from verification strategy concerns.
4. We show how our framework describes some advanced verification techniques for visible state invariants.
5. We prove soundness for a number of techniques, and, guided by our framework, discover an unsoundness in one technique.

Our framework allows the extraction of comparable data from techniques that were presented using different concepts, terminology and styles. Comparative value judgements concerning the techniques are beyond the scope of our paper.

Outline. Sec. 2 gives an overview of our work, explaining the important concepts. Sec. 3 formalises program and invariant semantics. Sec. 4 describes our framework and defines soundness. Sec. 5 instantiates our framework with existing verification techniques. Sec. 6 presents sufficient conditions for a verification technique to be sound, and states a general soundness theorem. Sec. 7 discusses related work. Proofs and more details are in the companion report [8]. This paper follows our FOOL paper [7], but provides more explanations and examples.

2 Example and Approach

Example. Consider a scenario, in which a Person holds an Account, and has a salary. An Account has a balance, an interestRate and an associated DebitCard. This example will be used throughout the paper. We give the code in Fig. 1.

```
class Account {
 Person holder;
 DebitCard card;
 int balance, interestRate;

 // invariant I1: balance < 0 ==>
       interestRate == 0;
 // invariant I2: card.acc == this;

 void withdraw(int amount) {
   balance -= amount;
   if (balance < 0) {
     interestRate = 0;
     this.sendReport();
   }
 }

 void sendReport()
   { holder.notify(); }
}

class SavingsAccount
            extends Account {
 // invariant I3: balance >= 0;
}
```

```
class Person {
  Account account;
  int salary;

  // invariant I4:
  //   account.balance + salary > 0;

  void spend(int amount)
    { account.withdraw(amount); }

  void notify()
    { ... }
}

class DebitCard {
  Account acc;
  int dailyCharges;

  // invariant I5:
  //   dailyCharges <= acc.balance;
}
```

Fig. 1. An account example illustrating the main challenges for the verification of object invariants. We assume that fields hold non-null values.

Account's interestRate is required to be zero when the balance is negative (I1). A further invariant (the two can be read as conjuncts of the full invariant for the class) ensures that the DebitCard associated with an account has a consistent reference back to the account (I2). A SavingsAccount is a special kind of Account, whose balance must be positive (I3). Person's invariant (I4) requires that the sum of salary and account's balance is positive. Finally, DebitCard's invariant (I5) requires dailyCharges not to exceed the balance of the associated account. Thus, I2, I4, and I5 are multi-object invariants.

To illustrate the challenges faced by verification techniques, suppose that p is an object of class Person, which holds the Account a with DebitCard d:

Call-backs: When p executes its method spend, this results in a call of withdraw on a, which (via a call to sendReport) eventually calls back notify on p; the call notify might reach p in a state where I4 does not hold.

Multi-object invariants: When a executes its method withdraw, it may temporarily break its invariant I1, since its balance is debited before any corresponding change is made to its interestRate . This violation is not important according to the visible state semantics; the **if** statement immediately afterwards ensures that the invariant is restored before the next visible state. However, by making an unrestricted reduction of the account balance, the method potentially breaks the invariants of other objects as well. In particular, p's invariant I4, and d's invariant I5 may be broken.

Subclassing: Further to the previous point, if a is a SavingsAccount, then calling the method withdraw may break the invariant I3, which was not necessarily known during the verification of class Account.

These points are addressed in the literature by striking various trade-offs between the differing aspects listed in the introduction.

Approach. Our framework uses seven parameters to capture the first four aspects in which verification techniques differ, *i.e.*, invariant semantics, invariant restrictions, proof obligations and program restrictions. To describe these parameters we use two abstract notions, which we call *regions* and *properties*. A *region* (when interpreted semantically) describes a set of objects (*e.g.*, those on which a method may be called), while a property describes a set of invariants (*e.g.*, the invariants that have to be proven before a method call). We deal with the aspects identified in the previous section as follows:

1. *Invariant semantics:* The property $\mathbb{X}$ describes the invariants expected to hold in visible states. The property $\mathbb{V}$ describes the invariants *vulnerable* to a given method, *i.e.*, those which may be broken while the method executes.
2. *Invariant restrictions:* The property $\mathbb{D}$ describes the invariants that may depend on a given heap location. This also characterises indirectly the locations an invariant may depend on.
3. *Proof obligations:* The properties $\mathbb{B}$ and $\mathbb{E}$ describe the invariants that must be proven to hold before a method call and at the end of a method body, respectively.
4. *Program restrictions:* The regions $\mathbb{U}$ and $\mathbb{C}$ describe the permitted receivers for field updates and method calls, respectively.
5. *Type systems:* We parameterise our framework by the type system. We state requirements on the type system, but leave abstract its concrete definition. We require that types are formed of a region-class pair so that we can handle types that express heap topologies (such as ownership types).
6. *Specification languages:* Rather than describing invariants concretely, we assume a judgement that expresses that an object satisfies the invariant of a class in a heap.
7. *Verification logics:* We express proof obligations via a special construct prv $\mathbb{p}$, which throws an exception if the invariants in property $\mathbb{p}$ cannot be proven, and has an empty effect otherwise. We leave abstract how the actual proofs are constructed and checked.

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

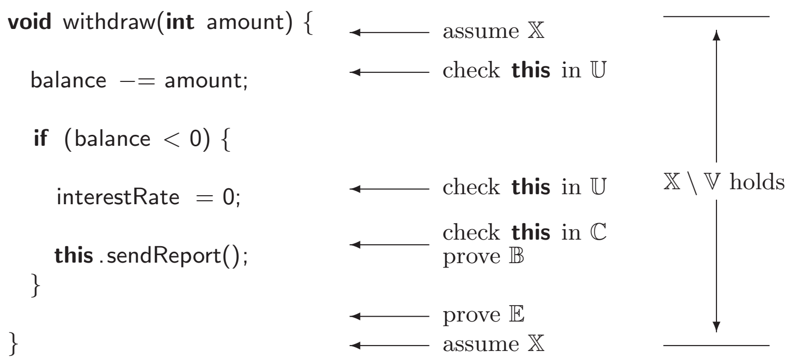

Fig. 2. Role of framework parameters for method withdraw from Fig. 1.

The number of parameters reflects the variety of concepts used by verification techniques, such as accessibility of fields, purity, helper methods, ownership, and effect specifications. For instance, $\mathbb{V}$ 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 general, methods may break but not re-establish invariants.

The seven parameters capture concepts explicitly or implicitly found in all verification techniques, defined either through words [27, 14, 16, 31] or typing rules [23]. For example, $\mathbb{V}$ is implicit in [27], but is crucial for their soundness argument. $\mathbb{X}$ and $\mathbb{V}$ are explicit in [23], while $\mathbb{U}$ and $\mathbb{C}$ are implicitly expressed as constraints in their typing rules. Subsets of these seven parameters characterise verification technique concepts e.g., soundness (through $\mathbb{X}$ and $\mathbb{V}$), expressiveness ($\mathbb{D}$, $\mathbb{X}$ and $\mathbb{V}$), proof obligations ($\mathbb{B}$ and $\mathbb{E}$).

3 Invariant Semantics

We formalise invariant semantics through an operational semantics, defining at which execution points invariants are required to hold. In order to cater for the different techniques, the semantics is parameterised by properties to express proof obligations and which invariants are expected to hold. In this section, we focus on the main ideas of our semantics and relegate the less interesting definitions to App. A. We assume sets of identifiers for class names CLS, field

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

Fig. 3. Source and runtime expression syntax.

names FLD, and method names MTHD, and use variables $c \in$ CLS, $f \in$ FLD and $m \in$ MTHD.

Runtime Structures. A *runtime structure* is a tuple consisting of a set of heaps HP, a set of addresses ADR, and a set of values VAL $=$ ADR $\cup \{\mathsf{null}\}$, using variables $h \in$ HP, $\iota \in$ ADR, and $v \in$ VAL. A runtime structure provides the following operations. The operation $dom(h)$ represents the domain of the heap. $cls(h, \iota)$ yields the class of the object at address ι. The operation $fld(h, \iota, f)$ yields the value of a field f of the object at address ι. Finally, $upd(h, \iota, f, v)$ yields the new heap after a field update, and $new(h, \iota, t)$ yields the heap and address resulting from the creation of a new object of type t. We leave abstract how these operations work, but require properties about their behaviour, for instance that upd only modifies the corresponding field of the object at the given address, and leaves the remaining heap unmodified. See Def. 9 in App. A for details.

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

Regions, Properties and Types. A region $\mathbb{r} \in$ R is a syntactic representation for a set of objects; a property $\mathbb{p} \in$ P is a syntactic representation for a set of assertions about particular objects. It is crucial that our syntax is parametric with the specific regions and properties; we use different regions and properties to model different verification techniques.[1]

We define a type $t \in$ TYP, as a pair of a region and a class. The region allows us to cater for types that express the topology of the heap, without being specific about the underlying type system.

Expressions. In Fig. 3, we define source expressions $e \in$ EXPR. In order to simplify our presentation (but without loss of generality), we restrict methods to always have exactly one argument. Besides the usual basic object-oriented constructs, we include proof annotations $e\ \mathsf{prv}\ \mathbb{p}$. As we will see later, such a proof annotation executes the expression e and then imposes a proof obligation for the invariants characterised by the property $\mathbb{p}$. To maintain generality, we avoid being precise about the actual syntax and checking of proofs.

[1] For example, in Universe types, **rep** and **peer** are regions, while in ownership types, ownership parameters such as X, and also **this**, are regions (more in Sec. 5).

In Fig. 3, we also define runtime expressions $e_r \in \text{RExpr}$. 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 $\mathsf{verfExc}$ indicates that a proof obligation failed. A fatal exception $\mathsf{fatalExc}$ indicates that an expected invariant does not hold. Runtime expressions can be decorated with $\mathsf{call}\, e_r$ and $\mathsf{ret}\, e_r$ to mark the beginning and end of a method call, respectively.

In Def. 10 (App. A), we define evaluation contexts, $E[\cdot]$, which describe contexts within one activation record and extend these to runtime contexts, $F[\cdot]$, which also describe nested calls.

Programming Languages. We define a programming language as a tuple consisting of a set Prg of programs, a runtime structure, a set of regions, and a set of properties (see Def. 11 in App. A). Each $\Pi \in \text{Prg}$ 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 signature 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 regions ($\sqsubseteq$), and interpretation ($[\![\cdot]\!]$) of regions and properties.

The interpretation of a region produces a set of objects. We characterise each invariant by an object-class pair, with the intended meaning that the invariant specified in the class holds for the object.[2] Therefore, the interpretation of a property produces a set of object-class pairs, specifying all the invariants of interest. Regions and properties are interpreted *w.r.t.* a heap, and from the *viewpoint* of a "current object"; therefore, their definitions depend on heap and address parameters: $[\![\ldots]\!]_{h,\iota}$.

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

Finally, the judgement $h \models \iota, c$ expresses that in heap h, the object at address ι satisfies the invariant declared in class c. We define that the judgement 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 property $\mathbb{p}$ is *valid* in heap h *w.r.t.* address ι if all invariants in $[\![\mathbb{p}]\!]_{h,\iota}$ are satisfied. We denote validity of properties by $h \models \mathbb{p},\, \iota$:

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

Operational Semantics. The framework parameter $\mathbb{X}$ describes which invariants are expected to hold at visible states. Given a program Π and a set of properties $\mathbb{X}_{c,m}$, each characterising the property that needs to hold at the beginning and end of a method m of class c, the *runtime semantics* is the relation $\longrightarrow \subseteq (\text{RExpr} \times \text{Hp}) \times (\text{RExpr} \times \text{Hp})$ defined in Fig. 4.

The first eight rules are standard for object-oriented languages. Note that in rNew, a new object is created using the function *new*, which takes a type as

[2] An object may have different invariants for each of the classes it belongs to [18].

(rThis)
$$\frac{\sigma = (\iota, \_, \_, \_)}{\sigma\cdot\mathsf{this},\ h \longrightarrow \sigma\cdot\iota,\ h}$$

(rVar)
$$\frac{\sigma = (\_, v, \_, \_)}{\sigma\cdot x,\ h \longrightarrow \sigma\cdot v,\ h}$$

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

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

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

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

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

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

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

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

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

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

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

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

Fig. 4. Reduction rules of operational semantics.

parameter rather than a class, thereby making the semantics parametric *w.r.t.* the type system: different type systems may use different regions and definitions of *new* to describe heap-topological information. Similarly, *upd* and *fld* do not fix a particular heap representation. Rule rCall describes method calls; it stores the class in which the method body is defined in the new stack frame σ, and introduces the "marker" $\mathsf{call}\,\mathrm{e}_r$ at the beginning of the method body.

Our reduction rules abstract away from program verification and describe only its effect. Thus, rLaunch, rLaunchExc, rFrame, and rFrameExc check whether $\mathbb{X}_{c,m}$ is valid at the beginning and end of any execution of a method m defined in class c, and generate a fatal exception, fatalExc, if the check fails. This represents the visible state semantics discussed in the introduction. Proof obligations $e\ \mathsf{prv}\ \mathbb{p}$ are verified once e reduces to a value (rPrf and rPrfExc); if $\mathbb{p}$ is not found to be valid, a verification exception verfExc is generated.

Verification using visible state semantics amounts to showing all proof obligations in some program logic, based on the assumption that expected invariants hold in visible states. Informally then, a specific verification technique described in our framework is sound if it guarantees that a fatalExc is never encountered. Verification technique soundness does allow verfExc to be generated, but this will never happen in a correctly verified program. We give a formal definition of soundness at the end of the next section.

This semantics allows us to be parametric *w.r.t.* the syntax of invariants and the logic of proofs. We also define properties that permit us to be parametric *w.r.t.* a sound type system (*cf.* Def. 15 in App. A). Thus, we can concentrate entirely on verification concerns.

4 Verification Techniques

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 regions and properties provided by the programming language. To allow the instantiations to refer to the program (for instance, to look up field declarations), we define a verification technique as a mapping from programs to 7-tuples.

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

$$\mathcal{V} \; : \; \textsc{Prg} \rightarrow \textsc{eXp} \times \textsc{Vul} \times \textsc{Dep} \times \textsc{Pre} \times \textsc{End} \times \textsc{Upd} \times \textsc{Cll}$$

where

$$\begin{array}{llll} \mathbb{X} \in & \textsc{eXp} = \textsc{Cls} \times \textsc{Mthd} \rightarrow \mathrm{P} & \mathbb{V} \in & \textsc{Vul} = \textsc{Cls} \times \textsc{Mthd} \rightarrow \mathrm{P} \\ \mathbb{D} \in & \textsc{Dep} = \textsc{Cls} \rightarrow \mathrm{P} & \mathbb{B} \in & \textsc{Pre} = \textsc{Cls} \times \textsc{Mthd} \times \mathrm{R} \rightarrow \mathrm{P} \\ \mathbb{E} \in & \textsc{End} = \textsc{Cls} \times \textsc{Mthd} \rightarrow \mathrm{P} & \mathbb{C} \in & \textsc{Cll} = \textsc{Cls} \times \textsc{Mthd} \times \textsc{Cls} \rightarrow \mathrm{R} \\ \mathbb{U} \in & \textsc{Upd} = \textsc{Cls} \times \textsc{Mthd} \times \textsc{Cls} \times \textsc{Mthd} \rightarrow \mathrm{R} & & \end{array}$$

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

$\mathbb{X}_{c,m}$: the property 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 [17] do not expect any invariants to hold.

$\mathbb{V}_{c,m}$: the property vulnerable to method m of class c, that is, the property whose validity may be broken while control is inside m. 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 property that may depend on fields declared in class c. The parameter c can be used, for instance, to prevent invariants from depending on fields declared in c's superclasses [16, 27].

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

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

$\mathbb{U}_{c,m,c'}$: the region 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 technique, for instance, to prevent field updates within pure methods.

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

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

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

$$\text{(vs-null)}\ \frac{}{\Gamma \vdash_{\mathcal{V}} \mathsf{null}} \quad \text{(vs-var)}\ \frac{}{\Gamma \vdash_{\mathcal{V}} x} \quad \text{(vs-this)}\ \frac{}{\Gamma \vdash_{\mathcal{V}} \mathsf{this}} \quad \text{(vs-new)}\ \frac{}{\Gamma \vdash_{\mathcal{V}} \mathsf{new}\, t} \quad \text{(vs-fld)}\ \frac{\Gamma \vdash_{\mathcal{V}} e}{\Gamma \vdash_{\mathcal{V}} e.f}$$

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

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

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

Fig. 5. Well-verified source expressions and classes.

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

Fig. 9 in App. A defines the judgement $h \vdash_{\mathcal{V}} e_r$ for verified runtime expressions. The rules correspond to those from Fig. 5, with the addition of rules for values and nested calls.

A program Π is well-verified *w.r.t.* $\mathcal{V}$, denoted as $\vdash_{\mathcal{V}} \Pi$, iff *(1)* all classes are well-verified and *(2)* all class invariants respect the dependency restrictions dictated by $\mathbb{D}$. That is, the invariant of an object ι' declared in a class c' will be preserved by an update of a field of an object of class c if it is not within $\mathbb{D}_c$.

Definition 2 $\vdash_{\mathcal{V}} \Pi \;\Leftrightarrow$

(1) $\forall c \in \Pi.\ \vdash_{\mathcal{V}} c$

(2) $\mathcal{F}(\mathit{cls}(h, \iota), f) = \_,\ c,\ (\iota', c') \notin [\![\mathbb{D}_c]\!]_{h,\iota},\ h \models \iota', c' \;\Rightarrow\; \mathit{upd}(h, \iota, f, v) \models \iota', c'$

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

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

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

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

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

(1) $\overline{\sigma}$ is a valid stack, denoted by $h \vdash_{\mathcal{V}} \overline{\sigma}$ (Def. 12 in App. A), and meaning that the receivers of consecutive method calls are within the respective $\mathbb{C}$ regions.
(2) The valid invariants $vi(\overline{\sigma}, h)$ hold.
(3) If execution is in a visible state with σ as the topmost frame of $\overline{\sigma}$, then the expected invariants $\mathbb{X}_\sigma$ hold additionally.

These properties are formalised in Def. 3. A state is determined by a heap h and a runtime expression e_r; the stack is extracted from e_r using function *stack*, given by Def. 13 in App. A.

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

$$\textit{(1)}\ \ h \vdash_{\mathcal{V}} \mathit{stack}(e_r) \qquad \textit{(2)}\ \ h \models vi(\mathit{stack}(e_r), h)$$
$$\textit{(3)}\ \ e_r = F[\sigma \cdot \mathsf{call}\, e] \ \ \textit{or}\ \ e_r = F[\sigma \cdot \mathsf{ret}\, v] \ \ \Rightarrow \ \ h \models \mathbb{X}_\sigma,\, \sigma$$

Soundness. A verification technique is *sound* if verified programs only produce valid states and do not throw fatal exceptions. More precisely, a verification technique $\mathcal{V}$ is sound for a programming language *PL* iff for all well-formed and verified programs $\Pi \in PL$, any well-typed and verified runtime expression e_r executed in a valid state reduces to another verified expression e'_r with a resulting valid state. Note that a verified e'_r contains no $\mathsf{fatalExc}$ (see Fig. 9).

Well-formedness of program Π is denoted by $\vdash_{\mathbf{wf}} \Pi$ (Def. 14, App. A). Well-typedness of runtime expression e_r is denoted by $h \vdash e_r : t$ and required as part of a sound type system in Def. 11, App. A. These requirement permits separation of concerns, whereby we can formally define verification technique soundness *in isolation*, assuming program well-formedness and a sound type system.

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

$$\left.\begin{array}{l} \vdash_{\mathbf{wf}} \Pi, \quad h \vdash e_r : \_, \quad \vdash_{\mathcal{V}} \Pi, \quad e_r \models_{\mathcal{V}} h, \\ h \vdash_{\mathcal{V}} e_r, \quad e_r, h \longrightarrow e'_r, h' \end{array}\right\} \Rightarrow \ e'_r \models_{\mathcal{V}} h', \quad h' \vdash_{\mathcal{V}} e'_r$$

5 Instantiations

In our earlier paper [7], we discuss six verification techniques from the literature in terms of our framework, namely those by Poetzsch-Heffter [31], Huizing &

Kuiper [14], Leavens & Müller [16], Müller *et al.* [27], and Lu *et al.* [23]. In this paper we concentrate on the techniques based on heap topologies [27, 23], because those benefit most from the formalisation in our framework.

Müller *et al.* [27] present two techniques for multi-object invariants, called ownership technique and visibility technique (*OT* and *VT* for short), which use the hierarchic heap topology enforced by Universe types [6]. Their distinctive features are: (1) Expected and vulnerable invariants are specified per class. (2) Invariant restrictions take into account subclassing (thereby addressing the subclass challenge). (3) Proof obligations are required before calls (thereby addressing the call-back challenge) and at the end of calls. (4) Program restrictions are uniform for all methods[3], and are based on the relative object placement in the hierarchy.

Lu *et al.* [23] define *Oval*, a verification technique based on ownership types, which support owner parameters for classes [5], thus permitting a more precise description of the heap topology. The distinctive features of *Oval* are: (1) Expected and vulnerable invariants are specific to every method in every class through the notion of *contracts*. (2) Invariant restrictions do not take subclassing into account. (3) Proof obligations are only imposed at the end of calls. (4) To address the call-back challenge, calls are subject to "subcontracting", a requirement that guarantees that the expected and vulnerable invariants of the callee are within those of the caller.

OT, *VT*, and *Oval* are discussed in more detail in our companion report [8]. In the remainder of this section, we introduce these techniques and summarise them in Fig. 6. We explain the notation from Fig. 6 informally, and define it formally in the appendix. This section (without the appendix) gives an overall intuition, aimed at the reader who is not interested in all of the formal details.

To sharpen our discussion *w.r.t.* structured heaps, we will be adding annotations to the example from Fig. 1, to obtain a topology where the Person p owns the Account a and the DebitCard d.

5.1 Instantiation for *OT* and *VT*

Universe types associate reference types with *Universe 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. Fig. 7 shows the Universe modifiers for our example from Fig. 1, which allow one to apply *OT* and *VT*.

To address the subclass challenge, *OT* and *VT* both forbid **rep** fields f and g declared in different classes c_f and c_g of the same object o to reference the same object. This *subclass separation* can be formalised in an ownership model where each object is owned by an object-class pair (see [18] for details).

[3] However, both *OT* and *VT* have special rules for pure (side-effect free) methods. We ignore pure methods here, but refer the interested reader to [7].

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

Fig. 6. Components of verification techniques. For *Oval*, $\mathcal{O}_{\mathbb{r},c}$ is the owner of $\mathbb{r}$; we use shorthands $\mathsf{I} = \mathsf{I}(c, m)$, and $\mathsf{E} = \mathsf{E}(c, m)$, and $\mathsf{I}' = \mathbb{r}\,;\mathsf{I}(c', m')$, and $\mathsf{E}' = \mathbb{r}\,;\mathsf{E}(c', m')$.

```
class Account {
  peer DebitCard card;
  any Person holder;
  ...
}

class Person {
  rep Account account;
  ...
}

class DebitCard {
  peer Account acc;
  ...
}
```

Fig. 7. Universe modifiers for the Account example from Fig. 1.

Regions and Properties. For *OT* and *VT*, we define the sets of regions and properties to be:

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

In our framework, Universe modifiers intuitively correspond to regions, since they describe areas of the heap. For example, peer describes all objects which share the owner (object-class pair) with the current object. However, because of the subclass separation described above, it is useful to employ richer regions of the form $\mathsf{rep}\langle c\rangle$, describing all objects owned by the current object *and* class c. For regions (and properties) we also include the "union" of two regions (properties). The predicate $\mathsf{intrsPeer}(\mathbb{r})$ checks whether a region intersects the peer region.

For properties, $\mathsf{self}\langle c\rangle$ represents the singleton set containing a pair of the current object with the class c. The property $\mathsf{super}\langle c\rangle$ represents the set of pairs of the current object with all its classes that are superclasses of c. The property $\mathsf{peer}\langle c\rangle$ represents all the objects (paired with their classes) that share the owner with the current object, provided their class is visible in c. There are also properties to describe the invariants of an object's owned objects, its owner, its transitively owned objects, and its transitive owners. A property of the form $\mathbb{p}_1;\mathbb{p}_2$ denotes a composition of properties, which behaves similarly to function composition when interpreted. The formal definitions of the interpretations of these regions and properties can be found in App. B.

Ownership Technique. As shown in Fig. 6, *OT* requires that in visible states, all objects owned by the owner of **this** must satisfy their invariants ($\mathbb{X}$).

Invariants are allowed to depend on fields of the object itself (at the current class), as in I1 in Fig. 1, and all its **rep** objects, as in I2. Other client invariants such as I4 and I5) and subclass invariants that depend on inherited fields (such as I3) are not permitted. Therefore, a field update potentially affects the invariants of the modified object and of all its (transitive) owners ($\mathbb{D}$).

A method may update fields of **this** ($\mathbb{U}$). Since an updated field is declared in the enclosing class or a superclass, the invariants potentially affected by the update are those of **this** (for the enclosing class and its superclasses, which addresses the subclass challenge) 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 ($\mathbb{D}$). Therefore, methods may break the invariants of the transitive owners of **this** ($\mathbb{V}$). For example, the invariant I2 of Person (Fig. 1) is legal only because account is a **rep** field (Fig. 7). Account's method withdraw need not preserve Person'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, which addresses the multi-object challenge. As an example, the method spend has to re-establish Person's invariant after the call to account.withdraw.

Since the invariants of the owners of **this** might not hold, *OT* disallows calls on references other than **rep** and **peer** references ($\mathbb{C}$). For instance, the call holder.notify() in method sendReport is not permitted because holder is in an ancestor ownership context.

The proof obligations for method calls ($\mathbb{B}$) must cover those invariants expected by the callee that are vulnerable to the caller. This intersection contains the invariant of the caller, if the caller and the callee are peers because the callee might call back; it is otherwise empty (**rep**s cannot callback their owners).

Visibility Technique. *VT* relaxes the restrictions of *OT* in two ways. First, it permits invariants of a class c to depend on fields of peer objects, provided that these invariants are visible in c ($\mathbb{D}$). Thus, *VT* can handle multi-object structures that are not organised hierarchically. For instance, in addition to the invariants permitted by *OT*, *VT* permits invariants I4 and I5 in Fig. 1. Visibility is transitive, thus, the invariant must also be visible wherever fields of c are updated. Second, *VT* permits field updates on peers of **this** ($\mathbb{U}$).

These relaxations make more invariants vulnerable. Therefore, $\mathbb{V}$ includes additionally the invariants of the peers of **this**. This addition is also reflected in the proof obligations before peer calls ($\mathbb{B}$) and before the end of a method ($\mathbb{E}$). For instance, method withdraw must be proven to preserve the invariant of the associated DebitCard, which does not in general succeed in our example.

5.2 Instantiation for *Oval*

Fig. 8 shows our example in *Oval* using ownership parameters [5] to describe heap topologies. The ownership parameter o denotes the owner of the current object;

p denotes the owner of o and specifies the position of holder in the hierarchy, more precisely than the **any** modifier in Universe types.

```
class Account[o,p] {
  DebitCard⟨o⟩ card;
  Person⟨p⟩ holder;
  ...
  void withdraw(int amount)⟨this,this⟩
    { ... }
  void sendReport()⟨bot,p⟩
    { ... }
}

class Person[o] {
  Account⟨this⟩ account;
  ...
  void spend(int amount)⟨this,this⟩
    { account.withdraw(amount); }
  void notify ()⟨bot,top⟩
    { ... }
}
```

Fig. 8. Ownership parameters and method contracts in *Oval*.

Method Contracts. Ownership parameters are also used to describe expected and vulnerable invariants, which are specific to each method. Every *Oval* program extends method signatures with a contract $\langle \mathsf{I}, \mathsf{E} \rangle$: the expected invariants at visible states ($\mathbb{X}$) are the invariants of the object characterised by I and all objects transitively owned by this object; the vulnerable invariants ($\mathbb{V}$) are the object at E and its transitive owners. These properties are syntactically characterised by Ls in the code (and Ks in typing rules), where:

$$L ::= \mathsf{top} \mid \mathsf{bot} \mid \mathsf{this} \mid X \qquad\qquad K ::= L \mid K ; \mathsf{rep}$$

and where X stands for the class' owner parameters.[4] An ordering $L \preceq L'$ is defined, expressing that at runtime the object denoted by L will be transitively owned by the object denoted by L'. This is used to formally specify various restrictions in the technique, for example that for all method contracts, $\mathsf{I} \preceq \mathsf{E}$ must hold.

In class Account (Fig. 8), withdraw() expects the current object and the objects it transitively owns to be valid (I=**this**) and, during execution, this method may invalidate the current object and its transitive owners (E=**this**). The contract of sendReport() does not expect any objects to be valid at visible states (I=**bot**) but may violate object p and its transitive owners (E=p).

Subcontracting. Call-backs are handled via *subcontracting*, which is defined using the order $L \preceq L'$.To interpret *Oval*'s subcontracting in our framework, we use $\mathsf{SC}(\mathsf{I}, \mathsf{E}, \mathsf{I}', \mathsf{E}', K)$, which holds iff:

$$\mathsf{I} \prec \mathsf{E} \;\Rightarrow\; \mathsf{I}' \preceq \mathsf{I} \qquad \mathsf{I} = \mathsf{E} \;\Rightarrow\; \mathsf{I}' \prec \mathsf{I} \qquad \mathsf{I}' \prec \mathsf{E}' \;\Rightarrow\; \mathsf{E} \preceq \mathsf{E}' \qquad \mathsf{I}' = \mathsf{E}' \;\Rightarrow\; \mathsf{E} \preceq K$$

[4] We discuss a slightly simplified version of *Oval*, where we omit the existential owner parameter '$*$', and *non-rep* fields, a refinement whereby only the current object's owners depend on such fields. Both enhance the expressiveness of the language, but are not central to our analysis.

where I, E characterise the caller, I', E' characterise the callee, and K stands for the callee's owner. The first two requirements ensure that the caller guarantees the invariant expected by the callee. The other two conditions ensure that the invariants vulnerable to the callee are also vulnerable to the caller. For instance, the call holder.notify() in method sendReport satisfies subcontracting because caller and callee do not expect any invariants, and the callee has no vulnerable invariants. In particular, the receiver of a call may be owned by any of the owners of the current receiver, provided that subcontracting is respected ($\mathbb{C}$).

Given that $\mathsf{I} \preceq \mathsf{E}$ for all well-formed methods, and that $\mathbb{B}_{c,m,\mathbb{r}} = \mathsf{emp}$, the first two requirements of subcontracting exactly give *(S1)*, while the latter two exactly give *(S3)* from Def. 5 in the next section – more in [8].

Regions and Properties. To express *Oval* in our framework, we define regions and properties as follows (see App. B for their interpretations):

$$\mathbb{r} \in \mathrm{R} ::= \mathsf{emp} \mid \mathsf{self} \mid c\langle\overline{K}\rangle \mid \mathbb{r} \sqcup \mathbb{r} \qquad \mathbb{p} \in \mathrm{P} ::= \mathsf{emp} \mid \mathsf{self} \mid K \mid K;\mathsf{rep}^* \mid K;\mathsf{own}^*$$

As already stated, expected and vulnerable properties depend on the contract of the method and express $\mathbb{X}$ as $\mathsf{I};\mathsf{rep}^*$ and $\mathbb{V}$ as $\mathsf{E};\mathsf{own}^*$ (see Fig. 6). Similarly to *OT*, invariant dependencies are restricted to an object and the objects it transitively owns ($\mathbb{D}$). Therefore, I1 and I4 are legal, as well as I3, which depends on an inherited field. *Oval* imposes a restriction on contracts that the expected and vulnerable invariants of every method intersect at most at this. Consequently, at the end of a method, one has to prove the invariant of the current receiver, if $\mathsf{I} = \mathsf{E} = \mathsf{this}$, and nothing otherwise ($\mathbb{E}$). In the former case, the method is allowed to update fields of its receiver; no updates are allowed otherwise ($\mathbb{U}$). Therefore, spend and withdraw are the only methods in our example that are allowed to make field updates. *Oval* does not impose proof obligations on method calls ($\mathbb{B}$ is empty), but addresses the call-back challenge through subcontracting. Therefore, call-backs are safe because the callee cannot expect invariants that are temporarily broken. With the existing contracts in Fig. 8, subcontracting permits spend to call account.withdraw(), and withdraw to call **this**.sendReport(), and also sendReport to call holder.notify(). The last two subcalls may potentially lead to callbacks, but are safe because the contracts of sendReport and notify do not expect the receiver to be in a valid state ($\mathsf{I}=\mathbf{bot}$).

Subclassing and Subcontracting. Oval also requires subcontracting between a superclass method and an overriding subclass method. As we discuss later, this does not guarantee soundness [22], and we found a counterexample (*cf.* Sec. 6). Therefore, we require that a subclass expects no more than the superclass, and vice versa for vulnerable invariants, and that if an expected invariant in the superclass is vulnerable in the subclass, then it must also be expected in the subclass:[5]

$$\mathsf{I}' \preceq \mathsf{I} \preceq \mathsf{E} \preceq \mathsf{E}' \qquad \mathsf{I} = \mathsf{E}' \;\Rightarrow\; \mathsf{I}' = \mathsf{E}'$$

[5] Note, that we had erroneously omitted the latter requirement in [7].

where I, E, I′, E′ characterise the superclass, resp. subclass, method. This requirement gives exactly *(S5)* from Def. 5. It allows $I' = I = E = E'$ which is forbidden in Oval. We refer to the verification technique with the above requirement for method overriding as *Oval′*.

5.3 Summary

In spite of differences in, *e.g.,* the underlying type systems and the logics used, our framework allows us to extract comparable information about these three techniques. We summarise here the commonalities and differences in the results.

1. *Invariant semantics:* In *OT* and *VT*, the invariants expected at the beginning of withdraw are I1, I2, and I3 for the receiver, as well as I5 for the associated DebitCard (which is a **peer**). For withdraw in *Oval*, I=this, therefore the expected invariants are I1, I2, and I3 for the receiver.
2. *Invariant restrictions:* Invariants I2 and I5 are illegal in *OT* and *Oval*, while they are legal in *VT* (which allows invariants to depend on the fields of **peer**s). Conversely, I3 is illegal in *OT* and *VT* (it mentions a field from a superclass), while it is legal in *Oval*.
3. *Proof obligations:* In *OT*, before the call to **this**.sendReport() and at the end of the body of withdraw, we have to establish I1 and I2 for the receiver. In addition to these, in *VT* we have to establish I5 for the debit card. In *Oval*, the same invariants as for *OT* have to be proven, but only at the end of the method because call-backs are handled through subcontracting. In addition, I3 is required.[6] In all three techniques, withdraw is permitted to leave the invariant I4 of the owning Person object broken. It has to be re-established by the calling Person method.
4. *Program restrictions:* *OT* and *VT* forbid the call holder.notify() (**rep**s cannot call their owners), while *Oval* allows it. On the other hand, if method sendReport required an invariant of its receiver (for instance, to ensure that holder is non-null), then *Oval* would prevent method withdraw from calling it, even though the invariants of the receiver might hold at the time of the call. The proof obligations before calls in *OT* and *VT* would make such a call legal.

6 Well-Structured Verification Techniques

We now identify conditions on the components of a verification technique that are sufficient for soundness, state a general soundness theorem, and discuss soundness of the techniques presented in Sec. 5

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

[6] This means that verification of a class requires knowledge of a subclass. The *Oval* developers plan to solve this modularity problem by requiring that any inherited method has to be re-verified in the subclass [22].

(S1) $\mathbb{r} \sqsubseteq \mathbb{C}_{c,m,c'm'} \Rightarrow (\mathbb{r} \triangleright \mathbb{X}_{c',m'}) \setminus (\mathbb{X}_{c,m} \setminus \mathbb{V}_{c,m}) \subseteq \mathbb{B}_{c,m,\mathbb{r}}$
(S2) $\mathbb{V}_{c,m} \cap \mathbb{X}_{c,m} \subseteq \mathbb{E}_{c,m}$
(S3) $\mathbb{C}_{c,m,c',m'} \triangleright (\mathbb{V}_{c',m'} \setminus \mathbb{E}_{c',m'}) \subseteq \mathbb{V}_{c,m}$
(S4) $\mathbb{U}_{c,m,c'} \triangleright \mathbb{D}_{c'} \subseteq \mathbb{V}_{c,m}$
(S5) $c' <: c \Rightarrow \mathbb{X}_{c',m} \subseteq \mathbb{X}_{c,m} \wedge \mathbb{V}_{c',m} \setminus \mathbb{E}_{c',m} \subseteq \mathbb{V}_{c,m} \setminus \mathbb{E}_{c,m}$

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

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

meaning that the interpretation of a viewpoint-adapted property $\mathbb{r} \triangleright \mathbb{p}$ *w.r.t.* an address ι is equal to the union of the interpretations of $\mathbb{p}$ *w.r.t.* each object in the interpretation of $\mathbb{r}$.

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

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

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

Note that the five soundness conditions presented here are slightly weaker than those in the previous version of this work [7]. [7]

Soundness Results. The five conditions from Def. 5 guarantee soundness of a verification technique (Def. 4), provided that the programming language has a sound type system (see Def. 15 in App. A).

[7] Namely, *(S3)* and *(S5)* are weaker, and thus less restrictive, here. In [7], instead of *(S3)* we required the stronger version $\mathbb{C}_{c,m,c',m'} \triangleright (\mathbb{V}_{c',m'} \setminus \mathbb{X}_{c',m'}) \subseteq \mathbb{V}_{c,m}$, and a similarly stronger version for *(S5)*. However, the two versions are equivalent when $\mathbb{E}_{c,m}$ is the minimal set allowed by *(S2)*, *i.e.*, when $\mathbb{E}_{c,m} = \mathbb{V}_{c,m} \cap \mathbb{X}_{c,m}$ for all c and m. In all techniques presented here, $\mathbb{E}_{c,m}$ is minimal in the above sense.

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

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

Unsoundness of Oval. The original *Oval* proposal [23] is unsound because it requires subcontracting for method overriding. As we said in the previous section, subcontracting corresponds to our *(S1)* and *(S3)*. This gives, for $c' <: c$, the requirements that $\mathbb{X}_{c',m'} \subseteq \mathbb{X}_{c,m} \setminus \mathbb{V}_{c,m}$, and $\mathbb{V}_{c',m'} \setminus \mathbb{E}_{c',m'} \subseteq \mathbb{V}_{c,m}$, which do not imply *(S5)*. We were alerted by this discrepancy, and using only the $\mathbb{X}$, $\mathbb{E}$ and $\mathbb{V}$ components (no type system properties, nor any other component), we constructed the following counterexample.

```
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 C1<o> {
   void mm() <bot,this> {...}
}
```

The call c.mm() is checked using the contract of C1::mm; it expects the callee to re-establish the invariant of the receiver (c), and is type correct. However, the body of C2::mm may break the receiver's invariants, but has no proof obligations ($\mathbb{E}_{\mathsf{C2},\mathsf{mm}} = \mathsf{emp}$). Thus, the call c.mm() might break the invariants of c, thus breaking the contract of m. The reason for this problem is, that the—initially appealing—parallel between subcontracting and method overriding does not hold. The authors confirmed our findings [22].

Soundness of the Presented Techniques.

Theorem 7 *The verification techniques OT, VT, and Oval′ are well-structured.*

Corollary 8 *The verification techniques OT, VT, and Oval′ are sound.*

Our proof of Corollary 8 confirmed soundness claims from the literature. We found that the semi-formal arguments supporting the original soundness claims at times missed crucial steps. For instance, the soundness proofs for *OT* and *VT* [27] do not mention any condition relating to *(S3)* of Def. 5; in our formal proof, *(S3)* was vital to determine what invariants still hold after a method returns. We relegate proofs of the theorems to the companion report [8].

7 Related Work

Object invariants trace back to Hoare's implementation invariants [12] and monitor invariants [13]. They were popularised in object-oriented programming by Meyer [24]. Their work, as well as other early work on object invariants [20, 21] 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 [27] for a discussion). However, once the requirements are clear, a formalisation within our framework seems straightforward.

The idea of regions and properties is inspired from type and effects systems [33], which have been extremely widely applied, *e.g.*, to support race-free programs and atomicity [10].

The verification techniques based on the Boogie methodology [1, 3, 18, 19] do not use a visible state semantics. Instead, each method specifies in its precondition which invariants it requires. Extending our framework to Spec# requires two changes. First, even though Spec# permits methods to specify explicitly which invariants they require, the default is to require the invariants of its arguments and all their peer objects. These defaults can be modelled in our framework by allowing method-specific properties $\mathbb{X}$. Second, Spec# checks invariants at the end of expose blocks instead of the end of method bodies. Expose blocks can easily be added to our formalism.

In separation logic [15, 32], object invariants are generally not as important as in other verification techniques. Instead, predicates specifying consistency criteria can be assumed/proven at *any* point in a program [28]. Abstract predicate families [29] allow one to do so without violating abstraction and information hiding. Parkinson and Bierman [30] show how to address the subclass challenge with abstract predicates. Their work as well as Chin *et al.*'s [4] allow programmers to specify which invariants a method expects and preserves, and do not require subclasses to maintain inherited invariants. The general predicates of separation logic provide more flexibility than can be expressed by our framework.

We know of only one technique based on visible states that cannot be directly expressed in our framework: Middelkoop *et al.* [26] use 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 properties; one for the pre-state heap and one for the post-state heap. Since this generality is not needed for any of the other techniques, we omitted a formal treatment in this paper.

Some verification techniques exclude the pre- and post-states of so-called helper methods from the visible states [16, 17]. Helper methods can easily be expressed in our framework by choosing different parameters for helper and non-helper methods. For instance in JML, $\mathbb{X}$, $\mathbb{B}$, and $\mathbb{E}$ are empty for helper methods, because they neither assume nor have to preserve any invariants.

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

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

8 Conclusions

We presented a framework that describes verification techniques for object invariants in terms of seven parameters and separates verification concerns from those of the underlying type system. Our formalism is parametric *w.r.t.* the type system of the programming language and the language used to describe and to prove assumptions. We illustrated the generality of our framework by instantiating it to describe three existing verification techniques. We identified sufficient conditions on the framework parameters that guarantee soundness, and we proved a universal soundness theorem. Our unified framework offers the following important advantages:

1. It allows a simpler understanding and separation of verification concerns. In particular, most of the aspects in which verification techniques differ are distilled in terms of subsets of the parameters of our framework.
2. It facilitates comparisons since relationships between parameters can be expressed at an abstract level (*e.g.*, criteria for well-structuredness in Def. 5), and the interpretations of regions and properties as sets allow formal comparisons of techniques in terms of set operations.
3. It expedites the soundness analysis of verification techniques, since checking the soundness conditions of Def. 5 is significantly simpler than developing soundness proofs from scratch.
4. It captures the design space of *sound* visible states based verification techniques.

We are currently using our framework in developing verification techniques for static methods, and plan to use it to develop further, more flexible, techniques.

Acknowledgements We thank Rustan Leino, Matthew Parkinson, Ronald Middelkoop, John Potter, Yi Lu, as well as the POPL, FOOL and ECOOP referees for their feedback. 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.

References

1. M. Barnett, R. DeLine, M. Fähndrich, K. R. M. Leino, and W. Schulte. Verification of object-oriented programs with invariants. *JOT*, 3(6):27–56, 2004.
2. M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *CASSIS*, LNCS, pages 49–69. Springer-Verlag, 2005.
3. M. Barnett and D. Naumann. Friends need a bit more: Maintaining invariants over shared State. In *MPC*, volume 3125 of *LNCS*, pages 54–84. Springer, 2004.

4. W.-N. Chin, C. David, H. H. Nguyen, and S. Qin. Enhancing modular OO verification with separation logic. In *POPL*, pages 87–99. ACM Press, 2008.
5. D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *OOPSLA*, volume 33(10), pages 48–64. ACM Press, 1998.
6. W. Dietl and P. Müller. Universes: Lightweight ownership for JML. *JOT*, 4(8):5–32, October 2005.
7. S. Drossopoulou, A. Francalanza, and P. Müller. A unified framework for verification techniques for object invariants. In *FOOL*, 2008.
8. S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summmers. A unified framework for verification techniques for object invariants — ecoop'08 full paper. Available from `http://www.doc.ic.ac.uk/~ajs300m/papers/frameworkFull.pdf`, 2008.
9. M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *OOPSLA*, pages 337–350. ACM Press, 2007.
10. C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *PLDI*, pages 338–349. ACM Press, 2003.
11. R. Hähnle and W. Mostowski. Verification of safety properties in the presence of transactions. In *CASSIS*, volume 3362 of *LNCS*, pages 151–171, 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 *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 *POPL*, pages 14–26. ACM Press, 2001.
16. G. T. Leavens and P. Müller. Information hiding and visibility in interface specifications. In *ICSE*, pages 385–395. IEEE, 2007.
17. 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.
18. K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, volume 3086 of *LNCS*, pages 491–516. Springer-Verlag, 2004.
19. K. R. M. Leino and W. Schulte. Using history invariants to verify observers. In *ESOP*, volume 4421 of *LNCS*, pages 316–330. Springer-Verlag, 2007.
20. B. Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.
21. B. Liskov and J. Wing. A behavioral notion of subtyping. *ACM ToPLAS*, 16(6):1811–1841, 1994.
22. Y. Lu and J. Potter. Soundness of Oval. Priv. Commun., June 2007.
23. Y. Lu, J. Potter, and J. Xue. Object Invariants and Effects. In *ECOOP*, volume 4609 of *LNCS*, pages 202–226. Springer-Verlag, 2007.
24. B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, 1988.
25. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
26. R. Middelkoop, C. Huizing, R. Kuiper, and E. J. Luit. Invariants for non-hierarchical object structures. *Electr. Notes Theor. Comput. Sci.*, 195:211–229, 2008.
27. P. Müller, A. Poetzsch-Heffter, and G. T. Leavens. Modular invariants for layered object structures. *Science of Computer Programming*, 62:253–286, 2006.
28. M. Parkinson. Class invariants: the end of the road? In *International Workshop on Aliasing, Confinement and Ownership*, 2007.

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

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

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

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

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

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

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

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

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

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

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

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

(ad-frame)
$$\frac{h \vdash_{\mathcal{V}} \sigma' \cdot e_r}{h \vdash_{\mathcal{V}} \sigma \cdot \sigma' \cdot \mathsf{ret}\, e_r\ \mathsf{prv}\ \mathbb{E}_{\sigma'}}$$

Fig. 9. Well-verified runtime expressions.

29. M. Parkinson and G. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258. ACM Press, 2005.
30. M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *POPL*, pages 75–86. ACM Press, 2008.
31. A. Poetzsch-Heffter. Specification and verification of object-oriented programs. Habilitation thesis, Technical University of Munich, 1997.
32. J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.
33. J. P. Talpin and P. Jouvelot. The Type and Effect Discipline. In *LICS*, pages 162–173. IEEE Computer Society, 1992.

A Appendix—The Framework

Definition 9 *A runtime structure is a tuple*

$$\text{RS}\textsc{truct} = (\textsc{Hp}, \textsc{Adr}, \simeq, \preceq, dom, cls, fld, upd, new)$$

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

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

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

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

(H2) $h \simeq h' \Rightarrow dom(h) = dom(h'),\ \ cls(h, \iota) = cls(h', \iota)$

(H3) $h \preceq h' \Rightarrow dom(h) \subseteq dom(h'),\ \ \forall \iota \in dom(h). cls(h, \iota) = cls(h', \iota)$

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

(H5) $new(h, \iota, t) = h', \iota' \Rightarrow h \preceq h',\ \ \iota' \in dom(h') \backslash dom(h)$

Definition 10 *$E[\cdot]$ and $F[\cdot]$ are defined as follows:*

$E[\cdot] ::= [\cdot] \mid E[\cdot].f \mid E[\cdot].f = e \mid \iota.f = E[\cdot] \mid E[\cdot].m(e) \mid \iota.m(E[\cdot]) \mid E[\cdot]\ \textsf{prv}\,\mathbb{p} \mid \textsf{ret}\,E[\cdot]$
$F[\cdot] ::= [\cdot] \mid F[\cdot].f \mid F[\cdot].f = e \mid \iota.f = F[\cdot] \mid F[\cdot].m(e) \mid \iota.m(F[\cdot]) \mid F[\cdot]\ \textsf{prv}\,\mathbb{p} \mid \sigma\cdot F[\cdot]$
$\quad \mid \textsf{call}\,F[\cdot] \mid \textsf{ret}\,F[\cdot]$

Definition 11 *A programming language is a tuple*

$$PL = (\text{Prg}, \text{RStruct}, \text{R}, \text{P})$$

where R *and* P *are sets, and* Prg *is a set where every $\Pi \in$* Prg *is a tuple*

$$\Pi = \begin{pmatrix} \mathcal{F}, \mathcal{M}, \mathcal{B}, <: \text{(class definitions)} & \sqsubseteq, [\![\cdot]\!] \text{ (inclusion and interpretations)} \\ \models, \vdash \quad \text{(invariant and type satisfaction)} & \end{pmatrix}$$

with signatures:

$\mathcal{F}\ :\ \text{Cls} \times \text{Fld} \rightharpoonup \text{Typ} \times \text{Cls}$ $\qquad \mathcal{M}\ :\ \text{Cls} \times \text{Mthd} \rightharpoonup \text{Typ} \times \text{Typ}$
$\mathcal{B}\ :\ \text{Cls} \times \text{Mthd} \rightharpoonup \text{Expr} \times \text{Cls}$
$<:\ \subseteq\ \text{Cls} \times \text{Cls}\ \cup\ \text{Typ} \times \text{Typ}$ $\qquad \sqsubseteq\ \subseteq\ \text{R} \times \text{R}$
$[\![\cdot]\!]\ :\ \text{R} \times \text{Hp} \times \text{Adr} \to \mathcal{P}(\text{Adr})$ $\qquad [\![\cdot]\!]\ :\ \text{P} \times \text{Hp} \times \text{Adr} \to \mathcal{P}(\text{Adr} \times \text{Cls})$
$\models\ \subseteq\ \text{Hp} \times \text{Adr} \times \text{Cls}$ $\qquad \vdash\ \subseteq\ (\text{Env} \times \text{Expr} \cup \text{Hp} \times \text{RExpr}) \times \text{Typ}$

where every $\Pi \in$ Prg *must satisfy the constraints:*

(P1) $\mathcal{F}(c, f) = t, c' \Rightarrow c <: c'$ $\qquad$ *(P2)* $\mathcal{B}(c, m) = e, c' \Rightarrow c <: c'$
(P3) $\mathcal{F}(cls(h, \iota), f) = t, \_ \Rightarrow \exists v. fld(h, \iota, f) = v$ $\qquad$ *(P4)* $\mathbb{r}_1 \sqsubseteq \mathbb{r}_2 \Rightarrow [\![\mathbb{r}_1]\!]_{h,\iota} \subseteq [\![\mathbb{r}_2]\!]_{h,\iota}$
(P5) $[\![\mathbb{r}]\!]_{h,\iota} \subseteq\ dom(h)$ $\qquad$ *(P6)* $h \preceq h' \Rightarrow [\![\mathbb{p}]\!]_{h,\iota} \subseteq [\![\mathbb{p}]\!]_{h',\iota}$
(P7) $\mathbb{r}\, c <: \mathbb{r}'\, c' \Rightarrow\ \mathbb{r} \sqsubseteq \mathbb{r}',\ c <: c'$

Definition 12 *Stack $\overline{\sigma}$ is* valid *w.r.t. heap h in a verification technique $\mathcal{V}$, denoted by $h \vdash_{\mathcal{V}} \overline{\sigma}$, iff:*

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

Definition 13 *The function stack* : RExpr $\to$ Stk* *yields the stack of a runtime expression:*

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

Definition 14 *For every program, the judgement:*

$\vdash_{\textit{wf}}\ :\ (\text{Hp} \times \text{Stk} \times \text{Stk} \times \text{R})\ \cup\ (\text{Env} \times \text{Hp} \times \text{Stk})\ \cup\ \text{Prg}$ *is defined as:*

$$- \vdash_{\textit{wf}} \Pi\ \Leftrightarrow \begin{cases} \textit{(F1)} & \mathcal{M}(c, m) = t, t' \ \Rightarrow\ \exists e.\ \mathcal{B}(c, m) = e, \_, \quad c, m, t \vdash e : t' \\ \textit{(F2)} & c <: c',\ \mathcal{F}(c', f) = t, c'' \ \Rightarrow\ \mathcal{F}(c, f) = t', c'',\ t' = t \\ \textit{(F3)} & c <: c',\ \mathcal{M}(c, m) = t, t',\ \mathcal{M}(c', m) = t'', t''' \ \Rightarrow\ t = t'',\ t' = t'''' \\ \textit{(F4)} & c <: c',\ \mathcal{B}(c', m) = e', c'' \ \Rightarrow\ \exists c'''.\ \mathcal{B}(c, m) = e, c''',\ c''' <: c'' \end{cases}$$

$$- h, \sigma \vdash_{\textit{wf}} \sigma' : \mathbb{r} \quad \Leftrightarrow \quad \sigma' = (\iota, \_, \_, \_), \quad h, \sigma \vdash \iota : \mathbb{r}\ \_$$

$$- \Gamma \vdash_{\textit{wf}} h, \sigma \quad \Leftrightarrow \quad \begin{cases} \exists c, m, t, \iota, v. \quad \Gamma = c, m, t, \quad \sigma = (\iota, v, c, m), \\ \qquad\qquad\qquad cls(h, \iota) <: c, \quad h, \sigma \vdash v : t \end{cases}$$

Definition 15 *A programming language* PL *has a* sound type system *if all programs $\Pi \in$* PL *satisfy the constraints:*

(T1) $\Gamma \vdash e : t,\ \ t <: t' \ \Rightarrow\ \Gamma \vdash e : t'$ $\qquad$ *(T2)* $h \vdash \mathrm{e}_r : t,\ \ t <: t' \ \Rightarrow\ h \vdash \mathrm{e}_r : t'$
(T3) $h \vdash \mathrm{e}_r : t,\ \ h \simeq h' \ \Rightarrow\ h' \vdash \mathrm{e}_r : t$ $\qquad$ *(T4)* $h \vdash \sigma\cdot\iota : \_\ c \ \Rightarrow\ cls(h, \iota) <: c$
(T5) $h \vdash \sigma\cdot\iota.m(v) : t \ \Rightarrow\ h \vdash \sigma\cdot\iota : \mathbb{r}\, c\ \mathcal{M}(c, m) = t', t,\ \ h \vdash \sigma\cdot v : t'$
(T6) $\sigma = (\iota, \_, \_, \_),\ h \vdash \sigma\cdot\iota' : \mathbb{r}\ \_ \ \Rightarrow\ \iota' \in [\![\mathbb{r}]\!]_{h,\iota}$
(T7) $\Gamma \vdash e : \mathbb{r}\, c,\ \ \Gamma \vdash h, \sigma \ \Rightarrow\ h \vdash \sigma\cdot e : \mathbb{r}\, c$
(T8) $\vdash_{\textit{wf}} \Pi,\ \ h, \sigma \vdash \mathrm{e}_r : t\ \ \mathrm{e}_r, h \longrightarrow \mathrm{e}'_r, h' \ \Rightarrow\ h', \sigma \vdash \mathrm{e}'_r : t$

B Appendix—The Instantiations

Müller et al. We assume an additional heap operation, which gives an object's owner: own : $\textsc{Hp} \times \textsc{Adr} \rightarrow \textsc{Adr} \times \textsc{Cls}$.
Regions are interpreted as follows:

$$\begin{array}{rlrl} [\![\mathsf{self}]\!]_{h,\iota} &= \{\iota\} & [\![\mathsf{any}]\!]_{h,\iota} &= dom(h) \\ [\![\mathsf{rep}\langle c\rangle]\!]_{h,\iota} &= \{\iota' \mid own(h,\iota') = \iota\, c\} & [\![\mathsf{emp}]\!]_{h,\iota} &= \emptyset \\ [\![\mathsf{peer}]\!]_{h,\iota} &= \{\iota' \mid own(h,\iota') = own(h,\iota)\} & [\![\mathbb{r}_1 \sqcup \mathbb{r}_2]\!]_{h,\iota} &= [\![\mathbb{r}_2]\!]_{h,\iota} \cup [\![\mathbb{r}_2]\!]_{h,\iota} \end{array}$$

Properties are interpreted as follows:

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

The predicate $\mathsf{intrsPeer}(\mathbb{r})$, is defined as:

$$\begin{array}{l} \mathsf{intrsPeer}(\mathsf{emp}) = \mathsf{intrsPeer}(\mathsf{rep}\langle c\rangle) = \mathit{false} \\ \mathsf{intrsPeer}(\mathsf{self}) = \mathsf{intrsPeer}(\mathsf{peer}) = \mathsf{intrsPeer}(\mathsf{any}) = \mathit{true} \\ \mathsf{intrsPeer}(\mathbb{r}_1 \sqcup \mathbb{r}_2) = \mathsf{intrsPeer}(\mathbb{r}_1) \mid\mid \mathsf{intrsPeer}(\mathbb{r}_2) \end{array}$$

Lu et al. We interpret regions as follows:

$$\begin{array}{l} [\![\mathsf{emp}]\!]_{h,\iota} = \emptyset \quad [\![\mathsf{self}]\!]_{h,\iota} = \{\iota\} \quad [\![\mathbb{r} \sqcup \mathbb{r}']\!]_{h,\iota} = [\![\mathbb{r}]\!]_{h,\iota} \cup [\![\mathbb{r}']\!]_{h,\iota} \\ [\![c\langle \overline{K}\rangle]\!]_{h,\iota} = \{\iota' \mid h \vdash \iota' : c\langle\bar{\iota}\rangle, \forall i.\, \iota_i \in \{\![K_i]\!\}_{h,\iota}\} \end{array}$$

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

$$\begin{array}{rl} [\![\mathsf{emp}]\!]_{h,\iota} = [\![\mathsf{top}]\!]_{h,\iota} = [\![\mathsf{bot}]\!]_{h,\iota} = \emptyset & [\![\mathsf{self}]\!]_{h,\iota} = \{(\iota,c) \mid ...\} \\ [\![K]\!]_{h,\iota} = \{(\iota',c) \mid \iota' \in \{\![K]\!\}_{h,\iota},\ cls(h,\iota') <: c\} & \\ [\![K;\mathbb{p}]\!]_{h,\iota} = \begin{cases} all(h) & K = \mathsf{top}, \mathbb{p} = \mathsf{rep}^* \ \vee K = \mathsf{bot}, \mathbb{p} = \mathsf{own}^* \\ \bigcup_{(\iota',c)\in[\![K]\!]_{h,\iota}} [\![\mathbb{p}]\!]_{h,\iota'} & \mathbb{p} \in \{\mathsf{rep}^*, \mathsf{own}^*\} \end{cases} & \\ [\![\mathsf{rep}^*]\!]_{h,\iota} = \{\iota' \mid h \vdash \iota' \preceq^* \iota\} & [\![\mathsf{own}^*]\!]_{h,\iota} = \{\iota' \mid h \vdash \iota \preceq^* \iota'\} \\ \{\![X]\!\}_{h,\iota} = \{\iota_i \mid h \vdash \iota : c\langle\bar{\iota}\rangle,\ c \text{ has formal parameters } \bar{X},\ X = X_i\} & \end{array}$$

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

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