Enhancing Modular OO Verification with Separation Logic

Wei-Ngan Chin[1,2] Cristina David[1] Huu Hai Nguyen[1,2] Shengchao Qin[3]

[1] Department of Computer Science, National University of Singapore, Singapore
[2] Computer Science Programme, Singapore-MIT Alliance, Singapore
[3] Department of Computer Science, Durham University, UK
{chinwn,davidcri,nguyenh2}@comp.nus.edu.sg shengchao.qin@durham.ac.uk

Abstract

Conventional specifications for object-oriented (OO) programs must adhere to behavioral subtyping in support of class inheritance and method overriding. However, this requirement inherently weakens the specifications of overridden methods in superclasses, leading to imprecision during program reasoning. To address this, we advocate a fresh approach to OO verification that focuses on the *distinction* and *relation* between specifications that cater to calls with static dispatching from those for calls with dynamic dispatching. We formulate a novel *specification subsumption* that can avoid code re-verification, where possible. Using a predicate mechanism, we propose a flexible scheme for supporting *class invariant* and *lossless casting*. Our aim is to lay the foundation for a *practical* verification system that is precise, concise and modular for sequential OO programs. We exploit the separation logic formalism to achieve this.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.3 [*Programming Languages*]: Language Construct and Features; F.3.1 [*Logics and Meanings of Programs*]: Specifying, Verifying and Reasoning about Programs

General Terms Languages, Theory, Verification

Keywords Automated Verification, Enhanced Subsumption, Separation Logic, Lossless Casting, Static and Dynamic Specifications.

1. Introduction

Object-based programs are hard to statically analyse mostly because of the need to track object mutations in the presence of aliases. Object-oriented (OO) programs are even harder, as we have to additionally deal with class inheritance and method overriding.

One major issue to consider when verifying OO programs is how to design specification for a method that may be overridden by another method down the class hierarchy, such that it conforms to method subtyping. In addition, it is important to ensure that subtyping is observed for object types in the class hierarchy, including any class invariant that may be imposed. From the point of conformance to OO semantics, most analysis techniques uphold Liskov's Substitutivity Principle (Liskov 1988) on behavioral subtyping. Under this principle, an object of a subclass can always be passed to a location where an object of its superclass is expected, as the object from each subclass must subsume the entire set of behaviors from its superclass. To enforce behavioral subtyping for OO programs, several past works (Dhara and Leavens 1996; Barnett et al. 2004; Kiniry et al. 2005) have advocated for class invariants to be inherited by each subclass, and for pre/post specifications of the overriding methods of its subclasses to satisfy a *specification subsumption* (or subtyping) relation with each overridden method of its superclass.

Each specification of a method (or a piece of code) is typically given as a pair $(\mathtt{pre}, \mathtt{post})$ of precondition `pre` and postcondition `post`. We use an infix notation $\mathtt{pre} \ast\!\!\!\to \mathtt{post}$ to denote such a specification. A basic *specification subsumption* mechanism was originally formulated as follows. Consider a method `B.mn` in class `B` with $(\mathtt{pre_B} \ast\!\!\!\to \mathtt{post_B})$ as its pre/post specification, and its overriding method `C.mn` in subclass `C`, with a given pre/post specification $(\mathtt{pre_C} \ast\!\!\!\to \mathtt{post_C})$. The specification $(\mathtt{pre_C} \ast\!\!\!\to \mathtt{post_C})$ is said to be a *subtype* of $(\mathtt{pre_B} \ast\!\!\!\to \mathtt{post_B})$ in support of method overriding, if the following subsumption relation holds:

$$\frac{\mathtt{pre_B} \wedge \mathit{type}(\mathtt{this}) <: \mathtt{C} \implies \mathtt{pre_C} \qquad \mathtt{post_C} \implies \mathtt{post_B}}{(\mathtt{pre_C} \ast\!\!\!\to \mathtt{post_C}) <:_{\mathtt{C}} (\mathtt{pre_B} \ast\!\!\!\to \mathtt{post_B})}$$

The two conditions are to ensure contravariance of preconditions, and covariance on postconditions. They follow directly from the subtyping principle on methods' specifications. As the two specifications are from different classes, we add the subtype constraint $\mathit{type}(\mathtt{this}) <: \mathtt{C}$ to allow the above subsumption relation to be checked for the *same* `C` subclass. To reflect this, we parameterize the subsumption operator $<:_{\mathtt{C}}$ with a `C`-class as its suffix.

The main purpose of using specification subsumption is to support modular reasoning by avoiding the need to re-verify the code of overriding method `C.mn` with the specification $(\mathtt{pre_B} \ast\!\!\!\to \mathtt{post_B})$ of its overridden method `B.mn`. In the case that specification subsumption does not hold, an alternative way to achieve behavioral subtyping is to use the *specification inheritance* technique of Dhara and Leavens (1996) to strengthen the specification of each overriding method with the specification of its overridden method, as follows:

Consider a method `B.mn` in class `B` with $(\mathtt{pre_B} \ast\!\!\!\to \mathtt{post_B})$ as its pre/post specification, and its overriding method `C.mn` in subclass `C`, with pre/post specification $(\mathtt{pre_C} \ast\!\!\!\to \mathtt{post_C})$. To ensure specification subsumption, we can strengthen the specification of the overriding method via *specification inheritance* with the intersection of their specifications, namely:

$$(\mathtt{pre_C} \ast\!\!\!\to \mathtt{post_C}) \bigwedge (\mathtt{pre_B} \wedge \mathit{type}(\mathtt{this}) <: \mathtt{C} \ast\!\!\!\to \mathtt{post_B}).$$

Specification inheritance requires the use of multiple specifications (or intersection type) to provide for a more expressive mechanism to describe each method. By inheriting a new specification for the overriding method, this technique uses code re-verification

POPL'08, January 7–12, 2008, San Francisco, California, USA.

itself to ensure that a behavioral subtyping property would be enforced. We can generalise a definition of the subsumption relation between two multiple specifications, as follows:

DEFINITION 1.1 (**Multi-Specifications Subsumption).** *Given two multiple specifications,* $\bigwedge_{j=1}^{n} \mathtt{specB_j}$ *(for class* B*) and* $\bigwedge_{i=1}^{m} \mathtt{specC_i}$ *(for subclass* C*), where each of* $\mathtt{specB_j}$ *and* $\mathtt{specC_i}$ *is a pre/post annotation of the form* $\mathtt{pre} \ast\!\!\to \mathtt{post}$*. We say that they are in specification subsumption relation,* $(\bigwedge_{i=1}^{m} \mathtt{specC_i}) <:_{\mathtt{C}} (\bigwedge_{j=1}^{n} \mathtt{specB_j})$*, if the following holds :* $\forall \mathtt{j}{\in}1..\mathtt{n}\ \exists \mathtt{i}{\in}1..\mathtt{m} \cdot \mathtt{specC_i} <:_{\mathtt{C}} \mathtt{specB_j}$.

While modular reasoning can be supported by the above subsumption relations, the reasoning were originally formulated in the framework of Hoare logic. Recently, separation logic has been proposed as an extension to Hoare logic, providing precise and concise reasoning for pointer-based programs. A key principle followed in separation logic is the use of local reasoning to facilitate modular analysis/reasoning. An early work on applying separation logic to the OO paradigm was introduced by Parkinson and Bierman (2005). In that work, two key concepts were identified. Firstly, an abstract predicate family $\mathtt{p_t(v_1, .., v_n)}$ was used to capture some program states for objects of class hierarchy with type t. Each abstract predicate has a visibility scope and is allowed to have a different number of parameters, depending on the actual type of its root object. Moreover, each predicate family acts as an *extensible* predicate for which incremental specification is given and verified for each class. Secondly, the concept of *specification compatibility* was introduced to capture the subsumption relation soundly, as follows:

A specification $\mathtt{pre_C} \ast\!\!\to \mathtt{post_C}$ is said to be *compatible* with $\mathtt{pre_B} \ast\!\!\to \mathtt{post_B}$ under all program contexts, if the following holds:

$$\frac{\forall \mathtt{code} \cdot \{\mathtt{pre_C}\}\mathtt{code}\{\mathtt{post_C}\} \implies \{\mathtt{pre_B}\}\mathtt{code}\{\mathtt{post_B}\}}{(\mathtt{pre_C} \ast\!\!\to \mathtt{post_C}) <: (\mathtt{pre_B} \ast\!\!\to \mathtt{post_B})}$$

Specification compatibility can be viewed as a more fundamental way to describe specification subsumption in terms of Hoare logic triples. However, it cannot be *directly* implemented, since its naive definition depends on exploring all possible program codes for compatibility. In this paper, we provide a *practical* alternative towards automated verification of OO programs that can support better precision and avoid unnecessary code re-verification. We use key principles of separation logic to achieve this.

1.1 Towards Better Precision and Reuse

The focus on specifications that support method overriding has a potential drawback that these specifications are typically imprecise (or weaker) for methods of superclasses. Such specifications typically have stronger preconditions (which restrict their applicability) and/or weaker postconditions (which lose precision). This drawback can cause imprecision for OO verification which has in turn spurred practical lessons on tips and tricks for specification writers (Kiniry et al. 2005). Furthermore, mechanisms such as specification inheritance may have unnecessary code re-verification, especially when specification subsumption holds.

Let us consider the specification of a simple up-counter class in Figure 1. This Cnt class is accompanied by three possible subclasses (i) FastCnt to support a faster tick operation, (ii) PosCnt which works only with positive numbers, (iii) TwoCnt which supports an extra backup counter.

Let us first design the specifications for instance methods of class Cnt *without* worrying about method overriding. A possible set of pre/post specifications is given below where this and res are variables denoting the receiver and result of each method.

```
void Cnt.tick()   static this::Cnt⟨n⟩ *→ this::Cnt⟨n+1⟩
void Cnt.set(int x)   static this::Cnt⟨n⟩ *→ this::Cnt⟨x⟩
int Cnt.get()   static this::Cnt⟨n⟩ *→ this::Cnt⟨n⟩∧res=n
```

```
class Cnt {  int val;
  Cnt(int v) {this.val:=v}
  void tick() {this.val:=this.val+1}
  int get() {this.val}
  void set(int x) {this.val:=x}
}
class FastCnt extends Cnt {
  FastCnt(int v) {this.val:=v}
  void tick() {this.val:=this.val+2}
}
class PosCnt extends Cnt inv this.val≥0 {
  PosCnt(int v) {this.val:=v}
  void set(int x) {if x≥0 then this.val:=x else error()}
}
class TwoCnt extends Cnt {  int bak;
  TwoCnt(int v, int b) {this.val:=v; this.bak:=b}
  void set(int x) {this.bak:=this.val; this.val:=x}
  void switch(int x)
  {int i:=this.val; this.val:=this.bak; this.bak:=i}
}
```

Figure 1. Example: Cnt and its subclasses

We refer to these as *static specifications* and precede them with the **static** keyword. They can be very precise as they were considered statically on a per method basis without concern for method overriding, and can be used whenever the actual type of the receiver is known. The notation $\mathtt{y::c\langle v_1, .., v_n\rangle}$ denotes that variable y is pointing to an object with the *actual type*[1] of c-class and where each field $\mathtt{y.f_i}$ is denoted by variable $\mathtt{v_i}$. For example, in $\mathtt{this::Cnt\langle n\rangle}$, the field this.val is denoted by variable n. This format for objects is used primarily for static specification. To describe an object type whose type is merely a *subtype* of the c-class, we shall use a different notation, namely $\mathtt{y::c\langle v^*\rangle\$}$, which implicitly captures an object extension with extra fields from its subclass.

If we take into account the possible overriding of the tick method by its corresponding method in the FastCnt subclass, we may have to weaken the postcondition of Cnt.tick. Furthermore, to guarantee the invariant this.val≥0 of the PosCnt class, we may have to strengthen the preconditions of methods Cnt.set, Cnt.tick and Cnt.get. These weakenings result in the following *dynamic specifications* which are the usual ones being considered for dynamically dispatched methods, where the type of the receiver is a subtype of its current class.

```
void Cnt.tick()  dynamic this::Cnt⟨n⟩$∧n≥0
                   *→ this::Cnt⟨b⟩$∧n+1≤b≤n+2
void Cnt.set(int x)
  dynamic this::Cnt⟨n⟩$∧x≥0 *→ this::Cnt⟨x⟩$
int Cnt.get()
  dynamic this::Cnt⟨n⟩$∧n≥0 *→ this::Cnt⟨n⟩$∧res=n
```

Such changes make the specifications of the methods in superclasses less precise, and are carried out to ensure behavioral subtyping. Furthermore, these specifications must also cater to potential modifications that may occur in the extra fields of the subclasses by their overriding methods either directly or indirectly. Due to conflicting requirements, we advocate the co-existence of both static and dynamic specifications. The former is important for precision and shall be used primarily for code verification, while the latter is needed to support method overriding and must be used for dynamically dispatched methods. Formally:

[1] To make our static specifications more reusable through inherited methods, we shall avoid the use of an explicit constraint, like type(this)=c, on the actual type of the receiver.

DEFINITION 1.2 (**Static Pre/Post).** *A specification is said to be* static *if it is meant to describe a single method declaration, and need not be used for subsequent overriding methods.*

DEFINITION 1.3 (**Dynamic Pre/Post).** *A specification is said to be* dynamic *if it is meant for use by a method declaration and its subsequent overriding methods.*

Past works, such as America (1991); Dhara and Leavens (1996); Liskov and Wing (1994); Parkinson (2005); Barnett et al. (2004); Muller (2002), are based primarily on dynamic specifications, though implicit static specifications via $\mathtt{type(this)}{=}\mathtt{c}$ can also be used in ESC/Java and Spec#, while JML uses *code contract* (Leavens et al. 2007, ch 15) as a form of static specification. However, these proposals for static specifications are somewhat ad-hoc, as they do not impose any relation between static and dynamic specifications. In our approach, we emphasize static specifications over dynamic specifications. Most importantly, we always ensure that the static specification of a method from a given class is always a *subtype* of the dynamic specification of the same method within the same class. This principle is important for modular verification, as we need only verify the code of each method *once* against its static specification. It is unnecessary to verify the corresponding dynamic specification since the latter is a specification supertype. Our proposal uses the following principles for OO verification, achieving both precision and reuse.

DEFINITION 1.4 (**Principles for Enhanced OO Verification).**

- *Static specification is given for each new method declaration, and may be added for inherited methods to support new auxiliary calls and subclasses with new invariants.*
- *Dynamic specification is either given or derived. Whether given or derived, each dynamic specification must satisfy two subsumption properties. Each must be a :*
  - *specification supertype of its static counterpart. This helps keep code re-verification to a minimum.*
  - *specification supertype of the dynamic specification of each overriding method in its subclasses. This helps ensure behavioral subtyping.*
- *Code verification is only performed for static specifications.*

1.2 Our Contributions

The main contributions of our paper are highlighted below:

- **Enhanced Specification Subsumption :** We improve on a classical specification subsumption relation. Apart from the usual checking for contravariance on preconditions and covariance on postconditions, we allow postcondition checking to be strengthened with the *residual heap state* from precondition checking. This enhancement is courtesy of the frame rule from separation logic which can improve modularity.
- **Static and Dynamic Specifications :** We advocate the coexistence of static and dynamic specifications, with an emphasis on the former. We impose an important subsumption relation between them. This principle allows for improved precision, while keeping code re-verifications to a minimum.
- **Lossless Casting :** We use a new object format that allows *lossless casting* to be performed. This format supports both *partial views* and *full views* for objects of classes that are suitable for static and dynamic specifications, respectively.
- **Statically-Inherited Methods :** New specifications may be given for inherited methods but must typically be re-verified. To avoid the need for *re-verification*, we propose for *specification subsumption* to be checked between each new static specification of the inherited method in a subclass against the static specification of the original method in the superclass. We identify a special category of *statically-inherited* methods that can safely avoid code re-verification for static specifications.
- **Deriving Specifications :** We propose techniques to *derive* dynamic specifications from static specifications, and show how *refinement* can be carried out to ensure behavioral subtyping.
- **Prototype System and Correctness Proof :** We have implemented a prototype system to validate our proposal, and formulated a set of lemmas on its correctness.

The next section provides more details of our approach in supporting objects for class inheritance, and methods via an enhanced specification subsumption relation.

2. Our Approach

Our approach to enhancing OO verification is based on separation logic. We shall describe how we adapt separation logic for reasoning about objects from a class hierarchy and how to write precise specifications that avoid unnecessary code re-verification.

2.1 Separation Logic and Aliasing

Separation logic (Reynolds 2002; Isthiaq and O'Hearn 2001) extends Hoare logic to support reasoning about shared mutable data structures. It adds two more connectives to classical logic: separating conjunction $*$, and separating implication $-\!\!*$. $h_1 * h_2$ asserts that two heaps described by h_1 and h_2 are domain-disjoint. $h_1 -\!\!* h_2$ asserts that if the current heap is extended with a disjoint heap described by h_1, then h_2 holds in the extended heap. In this paper we use only separating conjunction.

Existing formalisms from separation logic literature (e.g. Isthiaq and O'Hearn (2001); Reynolds (2002); Parkinson and Bierman (2005)) capture heap constraints (with an initial reference from $\mathtt{p}$) using two different notations, namely : i) $\mathtt{p} \mapsto [\ldots]$ to denote a pointer $\mathtt{p}$ to a single data object represented by $[\ldots]$, and ii) $\mathtt{pred(p,..)}$ for a pointer $\mathtt{p}$ to a set of linked objects in accordance to a predicate named $\mathtt{pred}$. In order to express both notations in a uniform manner, we introduce the notation $\mathtt{p{::}c}\langle..\rangle$ where $\mathtt{c}$ is either a data object or a heap predicate.

Aliasing can be locally specified and captured in separation logic. For instance, the formula $\mathtt{x{::}PosCnt}\langle\mathtt{a}\rangle{*}\mathtt{y{::}PosCnt}\langle\mathtt{b}\rangle$ specifies two distinct $\mathtt{PosCnt}$ objects referenced respectively by $\mathtt{x}$ and $\mathtt{y}$ that are non-aliased. In contrast, the formula $\mathtt{x{::}PosCnt}\langle\mathtt{a}\rangle{\wedge}\mathtt{x{=}y}$ specifies a single $\mathtt{PosCnt}$ object referenced by both $\mathtt{x}$ and its local alias $\mathtt{y}$. There may be other aliases to an object of the heap formula but the ability to perform local reasoning in separation logic allows us to ignore them. The following static method's specification captures both scenarios using multiple specifications of the form $\bigwedge(\mathtt{pre} \ast\!\!\rightarrow \mathtt{post})^*$ below:

$$\begin{array}{l}\mathtt{void\ simTick(PosCnt\ x,\ PosCnt\ y)}\\ \quad \mathtt{x{::}PosCnt}\langle\mathtt{a}\rangle{*}\mathtt{y{::}PosCnt}\langle\mathtt{b}\rangle \ast\!\!\rightarrow\\ \qquad \mathtt{x{::}PosCnt}\langle\mathtt{a{+}1}\rangle{*}\mathtt{y{::}PosCnt}\langle\mathtt{b{+}1}\rangle\\ \bigwedge\ \mathtt{x{::}PosCnt}\langle\mathtt{a}\rangle{\wedge}\mathtt{x{=}y} \ast\!\!\rightarrow \mathtt{x{::}PosCnt}\langle\mathtt{a{+}2}\rangle\\ \{\mathtt{x.tick();\,y.tick()}\}\end{array}$$

Note the effect of the method (specified in the post-conditions) can be very different depending on whether $\mathtt{x}$ and $\mathtt{y}$ are aliased, or not.

We support a heap predicate mechanism that can be *user-specified* to capture a group of related objects. As an example, we may use an $\mathtt{ll}$ predicate (or view) to capture a linear linked-list of $\mathtt{n}$ nodes, as follows:

$$\begin{array}{l}\mathtt{root{::}ll}\langle\mathtt{n}\rangle \equiv (\mathtt{root{=}null}{\wedge}\mathtt{n{=}0}) \vee (\exists \mathtt{i,m,q}\cdot\\ \qquad \mathtt{root{::}node}\langle\mathtt{i,q}\rangle{*}\mathtt{q{::}ll}\langle\mathtt{m}\rangle{\wedge}\mathtt{n{=}m{+}1})\ \textbf{inv}\ \mathtt{n{\geq}0}\\ \mathtt{class\ node\ \{int\ val;\ node\ next;\ ..methods..\}}\end{array}$$

For convenience, we name the first parameter of each predicate with a default name, called root, that may be omitted in the LHS. Each predicate may also have other parameters, such as pointers, integers or even bags and lists. These parameters correspond to *model fields* of some specification languages (Barnett et al. 2004; Muller 2002). Existential quantifiers may also be omitted without ambiguity. These simplifications result in the following more concise predicate definition:

$$\begin{array}{l}\mathtt{ll}\langle\mathtt{n}\rangle \equiv (\mathtt{root{=}null}\wedge\mathtt{n{=}0})\vee\\ \qquad \mathtt{root{::}node}\langle\mathtt{i,q}\rangle{*}\mathtt{q{::}ll}\langle\mathtt{n{-}1}\rangle\ \mathbf{inv}\ \mathtt{n{\geq}0}\end{array}$$

For simplicity, we shall also restrict our constraint domain to support essential features such as pointers, class subtyping and integers. The above definition captures an equivalence between a predicate and a heap formula in separation logic. Whenever the predicate holds, we can replace it by its heap formula. Similarly, whenever the formula holds, we can replace it by its corresponding predicate. The first technique is known as unfolding (or unrolling), while the second technique is known as folding (or rolling). These reasonings are sound and can be automated, as shown in Nguyen et al. (2007).

An important facet of separation logic is the frame rule:

$$\frac{\vdash\{\Delta\}\,\mathtt{e}\,\{\Delta_1\}}{\vdash\{\Delta{*}\Delta_R\}\,\mathtt{e}\,\{\Delta_1{*}\Delta_R\}}\ \mathit{modifies}(\mathtt{e})\cap \mathit{vars}(\Delta_R)=\emptyset$$

The side-condition says that the program e does not modify the variables in Δ_R. The frame rule captures the essence of "local reasoning": to understand how a piece of code works it should only be necessary to reason about the memory it actually accesses. Ordinarily, aliasing precludes such a possibility but the separation enforced by the $*$ connective allows local reasoning to be captured by the above rule. Through this frame rule, a specification of the heap being used by e can be arbitrarily extended as long as free variables of the extended part are not modified by e.

To automate the reasoning process, we have formalised in Nguyen et al. (2007) a procedure for entailment with frame inference capability :

DEFINITION 2.1 (Entailment with Frame Capability). *The entailment $\Delta_A{\vdash}\Delta_C * \Delta_R$ checks that heap nodes in the antecedent Δ_A are sufficiently precise to cover all nodes from the consequent Δ_C, and can return a residual heap state Δ_R (from Δ_A) that is not used.*

For example if we have $\Delta_A=\mathtt{x{::}ll}\langle\mathtt{n}\rangle\wedge\mathtt{n{>}5}$ and $\Delta_C=\mathtt{x{::}node}\langle\_,\mathtt{y}\rangle$, the entailment process would succeed (via unfolding $\mathtt{ll}\langle\mathtt{n}\rangle$ in Δ_A to match with the object node in Δ_C) and also return residual (or frame) $\Delta_R=\mathtt{y{::}ll}\langle\mathtt{n{-}1}\rangle\wedge\mathtt{n{>}5}$. That is:

$$\begin{array}{ll}\Delta_A & \equiv\ \mathtt{x{::}ll}\langle\mathtt{n}\rangle\wedge\mathtt{n{>}5}\\ & \equiv\ \mathtt{x{::}node}\langle\_,\mathtt{q}\rangle{*}\mathtt{q{::}ll}\langle\mathtt{n{-}1}\rangle\wedge\mathtt{n{>}5}\\ & \vdash \Delta_C * \Delta_R\\ & \vdash (\mathtt{x{::}node}\langle\_,\mathtt{y}\rangle) * \Delta_R\\ & \vdash (\mathtt{x{::}node}\langle\_,\mathtt{y}\rangle) * (\mathtt{y{::}ll}\langle\mathtt{n{-}1}\rangle\wedge\mathtt{n{>}5})\end{array}$$

2.2 Object View and Lossless Casting

For separation logic to work with OO programs, one key problem that we must address is a suitable format to capture the objects of classes. We should preferably also address the problem of performing upcast/downcast operations statically in accordance with the OO class hierarchy, and without loss of information where possible.

Consider two variables, x and y, which point to objects from Cnt class (with a single field) and TwoCnt class (with two fields), respectively. Intuitively, we may represent the first object by $\mathtt{x{::}Cnt}\langle\mathtt{v}\rangle$ where v denotes its field, and the second object by $\mathtt{y{::}TwoCnt}\langle\mathtt{v,b}\rangle$ where v, b denote its two fields. However, a fundamental problem that we must solve is how to cast the object of one class to that of its superclass, and vice-versa when needed. To do this without loss of information, we provide two extra information : (i) a variable to capture the *actual type* of a given object and (ii) a variable to capture the object's *record extension* that contains extra field(s) of its subclass. When a TwoCnt object is first created, we may capture its state using the formula :

$$\mathtt{y{::}TwoCnt}\langle\mathtt{t,v,b,p}\rangle\wedge\mathtt{t{=}TwoCnt}\wedge\mathtt{p{=}null}$$

The above formula indicates that the actual type of the object is $\mathtt{t{=}TwoCnt}$ and that there is no need for any record extension since $\mathtt{p{=}null}$. With this object format, we can now perform an upcast to its parent Cnt class by transforming it to:

$$\mathtt{y{::}Cnt}\langle\mathtt{t,v,q}\rangle{*}\mathtt{q{::}Ext}\langle\mathtt{TwoCnt,b,p}\rangle\wedge\mathtt{t{=}TwoCnt}\wedge\mathtt{p{=}null}$$

Though this cast operation is viewing the object as a member of Cnt class, it is still a TwoCnt object as the type information $\mathtt{t{=}TwoCnt}$ indicates. Furthermore, we have created an extension record $\mathtt{q{::}Ext}\langle\mathtt{TwoCnt,b,p}\rangle$ that can capture the extra b field of the TwoCnt subclass. For simplicity, we currently use an implicit pointer q to capture the extension record. This model allows a sequence of upcast operations to be easily captured. Such an upcast operation is *lossless* as we have sufficient information to perform the inverse downcast operation back to the original TwoCnt format. To allow lossless casting between Cnt and TwoCnt, we add an equivalence rule :

$$\mathtt{TwoCnt}\langle\mathtt{t,v,b,p}\rangle\equiv\mathtt{root{::}Cnt}\langle\mathtt{t,v,q}\rangle{*}\mathtt{q{::}Ext}\langle\mathtt{TwoCnt,b,p}\rangle$$

An unfold step (which replaces a term that matches the LHS by RHS) corresponds to an upcast operation, while the fold step (which replaces a term that matches the RHS by LHS) corresponds to a downcast operation. Such a rule can be derived from each superclass-subclass pairing. Formally:

DEFINITION 2.2 (**Lossless Casting).** *Given a class $\mathtt{c}\langle\mathtt{v}^*\rangle$ with fields $\mathtt{v}^*$ and its immediate subclass $\mathtt{d}\langle\mathtt{v}^*,\mathtt{w}^*\rangle$ where $\mathtt{w}^*$ denotes its extra fields, we shall generate the following casting rule that is coercible in either direction:*

$$\mathtt{root{::}d}\langle\mathtt{t},\mathtt{v}^*,\mathtt{w}^*,\mathtt{p}\rangle\ \equiv\ \mathtt{root{::}c}\langle\mathtt{t},\mathtt{v}^*,\mathtt{q}\rangle{*}\mathtt{q{::}Ext}\langle\mathtt{d},\mathtt{w}^*,\mathtt{p}\rangle$$

Note that for any object view $\mathtt{d}\langle\mathtt{t},\ldots\rangle$ it is always the case that the subtype relation $\mathtt{t{<:}d}$ holds as its invariant. Furthermore,the default root *parameter on the LHS may be omitted for brevity.*

Lossless casting is important for establishing the subsumption relation between each static specification and its dynamic counterpart, as the extension record can be preserved, if needed, by each static specification. Lossless casting is also important for the static specification of inherited methods which should preferably be inherited without the need for re-verification. This can be achieved by exploiting local reasoning which allows us to assert that an extension record need not be modified by each inherited method.

There are also occasions when we are required to pass the full object with all its (extended) fields. This occurs for dynamic specifications where subsequent overriding method may change the extra fields of its subclass. To cater to this scenario, we introduce an $\mathtt{ExtAll}\langle\mathtt{t_1,t_2}\rangle$ view that can capture all the extension records from a class $\mathtt{t_1}$ for an object with actual type $\mathtt{t_2}$. This scenario occurs for the dynamic specification of Cnt.set method, as shown below:

$$\begin{array}{l}\mathtt{this{::}Cnt}\langle\mathtt{t},\_,\mathtt{p}\rangle{*}\mathtt{p{::}ExtAll}\langle\mathtt{Cnt,t}\rangle\wedge\mathtt{x{\geq}0}\\ \quad {*}{\!\!\rightarrow}\ \mathtt{this{::}Cnt}\langle\mathtt{t,x,p}\rangle{*}\mathtt{p{::}ExtAll}\langle\mathtt{Cnt,t}\rangle\end{array}$$

Such a dynamic specification may be used with any subtype of Cnt. The entire object view must be passed to support dynamic specifications which are expected to cater to the current method and all subsequent overriding methods. The ExtAll predicate itself can be defined as follows:

$$\begin{array}{l}\mathtt{ExtAll}\langle\mathtt{t_1,t_2}\rangle\equiv\mathtt{t_1{=}t_2}\wedge\mathtt{root{=}null}\vee\mathtt{root{::}Ext}\langle\mathtt{t_3},\mathtt{v}^*,\mathtt{q}\rangle\\ \qquad\qquad {*}\mathtt{q{::}ExtAll}\langle\mathtt{t_3,t_2}\rangle\wedge\mathtt{t_3{<}t_1}\wedge\mathtt{t_2{<:}t_3}\ \mathbf{inv}\ \mathtt{t_2{<:}t_1}\end{array}$$

The notation $\mathtt{t_3{<}t_1}$ denotes a class $\mathtt{t_3}$ and its immediate superclass $\mathtt{t_1}$. The ExtAll predicate is used to generate all the extension records from class $\mathtt{t_1}$ to $\mathtt{t_2}$. For example, expression $\mathtt{x::ExtAll\langle Cnt, Cnt\rangle}$ yields $\mathtt{x{=}null}$, and $\mathtt{x::ExtAll\langle Cnt, TwoCnt\rangle}$ yields $\mathtt{x::Ext\langle TwoCnt, b, null\rangle}$.

Our format allows two kinds of object views to be supported:

DEFINITION 2.3 **(Full and Partial Views).** *We refer to the use of formula* $\mathtt{x::c\langle t, v^*, p\rangle{*}p::ExtAll\langle c, t\rangle}$ *as providing a* full view *for an object with actual type* $\mathtt{t}$ *that is being treated as a* $\mathtt{c}$*-class object, while* $\mathtt{x::c\langle t, v^*, p\rangle}$ *provides only a* partial view *with no extension record. For brevity, full views are also written as* $\mathtt{x::c\langle v^*\rangle\$}$*, while partial views are coded using* $\mathtt{x::c\langle v^*\rangle}$.

This distinction between *partial* and *full* views (for objects) follows directly from our decision to distinguish static from dynamic specifications. Partial views are typically used for the receiver object of static specifications, while full views are used by dynamic specifications. Some readers may contend that lossless casting of an object $\mathtt{x}$ from $\mathtt{d\langle v^*, w^*\rangle}$ to $\mathtt{c\langle v^*\rangle}$ may also be captured with the help of separating implication by representing the extension record using $\mathtt{x::c\langle v^*\rangle} \mathrel{-\!\!*} \mathtt{x::d\langle v^*, w^*\rangle}$. This approach works well for partial views, but cannot easily handle the `ExtAll` predicate required by full views. Moreover, by omitting separating implication, our current approach to automated verification is easier to build and prove.

2.3 Ensuring Class Invariants

Ensuring that class invariants hold can be rather intricate, with the key problems being *how* and *when* to check for the invariants. Based on the simplest assumption, one would expect object invariants to hold at all times. However, this assumption is impractical for mutable objects. One sensible solution is to expect invariants to hold based on visible state semantics, which is typically aligned to the boundaries of public methods. Even this approach may not be flexible enough. Thus, in Boogie (Barnet et al. 2004; Leino and Müller 2004), programmers are also allowed to use a specification field, called `valid`, that can indicate if the invariant for an object is being preserved or temporarily broken for mutation. Similarly, in (Middelkoop et al. 2006), programmers are allowed to indicate invariants that are inconsistent (not preserved) at some method boundary.

We aim for a similar level of flexibility but which still remains easy to use. To achieve this, we introduce the concept of an *invariant-enhanced view* for each class with a non-trivial invariant, as follows:

DEFINITION 2.4 **(Invariant-Enhanced View).** *Consider a class* $\mathtt{c}$ *with a non-trivial invariant* $\delta_{\mathtt{c}}$ ($\neq\mathtt{true}$) *over the fields* $\mathtt{v^*}$ *of the object. We shall define a new view of the form* $\mathtt{c\#I\langle v^*\rangle}$ *to capture its class invariant, as :* $\mathtt{c\#I\langle v^*\rangle} \equiv \mathtt{root::c\langle v^*\rangle}{*}\delta_{\mathtt{c}}$. *Furthermore, for each subclass* $\mathtt{d\langle v^*, w^*\rangle}$ *with an extra invariant* $\delta_{\mathtt{d}}$ *over the fields* $\mathtt{v^*, w^*}$*, we expect its invariant to be* $\delta_{\mathtt{c}}{*}\delta_{\mathtt{d}}$ *and shall provide a corresponding view :* $\mathtt{d\#I\langle v^*, w^*\rangle} \equiv \mathtt{root::d\langle v^*, w^*\rangle}{*}\delta_{\mathtt{c}}{*}\delta_{\mathtt{d}}$.

The use of separating conjunction to capture the class invariant allows a form of object ownership to be specified for heap objects present in $\delta_{\mathtt{c}}$. Furthermore, invariant-enhanced view can easily and explicitly indicate when an invariant can be enforced and when it can be assumed. If a $\mathtt{c\langle v^*\rangle}$ is being used, the class invariant is neither enforced nor assumed. If a $\mathtt{c\#I\langle v^*\rangle}$ is used in the precondition, its invariant must be enforced at each of its call sites, but can be assumed to hold at the beginning of its method declaration. If a $\mathtt{c\#I\langle v^*\rangle}$ is used in the postcondition, its invariant must be enforced at the end of its method declaration, but can be assumed to hold at the post-state of each of its call sites.

With the help of invariant-enhanced views, we can provide pre/post specifications that guarantee class invariants are always maintained by public methods. This can help ensure that all objects created and manipulated by public methods are guaranteed to satisfy their class invariants. Alternatively, it is also possible to allow some methods (typically private ones) to receive or produce objects *without* the invariant property. This corresponds to situations where the class invariant is temporarily broken. Our invariant-enhanced views can achieve this as they can be selectively and automatically enforced in pre/post annotations.

For example, the invariant-enhanced view of `PosCnt` is:

$$\mathtt{PosCnt\#I\langle t, v, p\rangle} \equiv \mathtt{root::PosCnt\langle t, v, p\rangle{*}v{\geq}0}$$

Two methods `get` and `tick` are being inherited from the `Cnt` superclass, while a third method `set` is re-defined to ensure the class invariant. We may provide new static specifications for these three respective methods, to incorporate the invariant-enhanced view. Figure 2 shows how this is done for our running example. It is sufficient to use a weaker precondition of the form $\mathtt{this::PosCnt\langle v\rangle}$ for *static-spec*($\mathtt{PosCnt.set}$) without compromising its postcondition $\mathtt{this::PosCnt\#I\langle x\rangle}$. This corresponds to a temporary violation of the class invariant of `PosCnt`.

2.4 Enhanced Specification Subsumption

With our use of more precise static specifications, we can now leverage on a better specification subsumption that can exploit the local reasoning capability of separation logic. In particular, the extended fields of objects that are not used should be preserved by specification subsumption. More formally, we define the enhanced form of specification subsumption, as follows:

DEFINITION 2.5 **(Enhanced Spec. Subsumption).** *A pre/post annotation* $\mathtt{preB} \mathrel{*\!\!\longrightarrow} \mathtt{postB}$ *is said to be a subtype of another pre/post annotation* $\mathtt{preA} \mathrel{*\!\!\longrightarrow} \mathtt{postA}$ *if the following relation holds:*

$$\frac{\mathtt{preA} \vdash \mathtt{preB}{*}\Delta \qquad \mathtt{postB}{*}\Delta \vdash \mathtt{postA}}{(\mathtt{preB} \mathrel{*\!\!\longrightarrow} \mathtt{postB}) <: (\mathtt{preA} \mathrel{*\!\!\longrightarrow} \mathtt{postA})}$$

Note that Δ *captures the* residual heap state *from the contravariance check on preconditions that is carried forward to assist in the covariance check on postconditions.*

As an example of its utility, consider the following specification subsumption that is expected to hold for enhanced OO verification.

$$\textit{static-spec}(\mathtt{Cnt.set}) <: \textit{dynamic-spec}(\mathtt{Cnt.set})$$

For the above to hold, we must prove:

$$\begin{array}{l} \mathtt{this::Cnt\langle t, v, p\rangle} \mathrel{*\!\!\longrightarrow} \mathtt{this::Cnt\langle t, x, p\rangle} \\ \quad <: \mathtt{this::Cnt\langle t, v, q\rangle{*}q::ExtAll\langle Cnt, t\rangle{\wedge}x{\geq}0} \mathrel{*\!\!\longrightarrow} \\ \qquad\quad \mathtt{this::Cnt\langle t, x, q\rangle{*}q::ExtAll\langle Cnt, t\rangle} \end{array}$$

The above subtyping <u>cannot</u> be proven with the basic specification subsumption relation from Sec 1 (without the use of a residual heap state), but succeeds with our enhanced subsumption relation.

We first show the contravariance of the preconditions:

$$\begin{array}{l} \mathtt{this::Cnt\langle t, v, q\rangle{*}q::ExtAll\langle Cnt, t\rangle{\wedge}x{\geq}0} \\ \quad \vdash \mathtt{this::Cnt\langle t, v, p\rangle}{*}\Delta \end{array}$$

This succeeds with $\Delta{\equiv}\mathtt{p::ExtAll\langle Cnt, t\rangle{\wedge}x{\geq}0}$. We then prove covariance on the postconditions using:

$$\mathtt{this::Cnt\langle t, x, p\rangle}{*}\Delta \vdash \mathtt{this::Cnt\langle t, x, q\rangle{*}q::ExtAll\langle Cnt, t\rangle}$$

This is proven with the help of residual heap state Δ (with an extension record) from the entailment of preconditions.

Our preservation of residual heap state is inspired by the needs of static specification. By the use of a new object format (with lossless casting) and a novel specification subsumption mechanism, we can now support a modular verification process in which re-verification is always avoided for dynamic specifications. Our enhanced specification subsumption can also be viewed as a practical

```
class Cnt { int val;
  Cnt(int v) static true *→ res::Cnt⟨v⟩
  {this.val:=v}
  void tick() static this::Cnt⟨v⟩ *→ this::Cnt⟨v+1⟩;
     dynamic this::Cnt⟨v⟩$∧v≥0 *→ this::Cnt⟨w⟩$∧v+1≤w≤v+2
  {this.val:=this.val+1}
  int get() static this::Cnt⟨v⟩ *→ this::Cnt⟨v⟩∧res=v
     dynamic this::Cnt⟨v⟩$∧v≥0 *→ this::Cnt⟨v⟩$
  {this.val}
  void set(int x) static this::Cnt⟨v⟩ *→ this::Cnt⟨x⟩;
     dynamic this::Cnt⟨v⟩$∧x≥0 *→ this::Cnt⟨x⟩$
  {this.val:=x}
}
class FastCnt extends Cnt {
  FastCnt(int v) static true *→ res::FastCnt⟨v⟩
  {this.val:=v}
  void tick() static this::FastCnt⟨v⟩ *→
     this::FastCnt⟨v+2⟩  {this.val:=this.val+2}
}
```

```
class PosCnt extends Cnt
    inv this.val≥0 {
  PosCnt(int v) static v≥0 *→ res::PosCnt#I⟨v⟩
  {this.val:=v}
  void tick() static this::PosCnt#I⟨v⟩ *→ this::PosCnt#I⟨v+1⟩
     dynamic this::PosCnt#I⟨v⟩$ *→ this::PosCnt#I⟨v+1⟩$
  int get() static this::PosCnt#I⟨v⟩ *→ this::PosCnt#I⟨v⟩∧res=v
  void set(int x) static this::PosCnt⟨v⟩∧x≥0 *→ this::PosCnt#I⟨x⟩
     dynamic this::PosCnt⟨v⟩$∧x≥0 *→ this::PosCnt#I⟨x⟩$
  {if x≥0 then this.val:=x else error()}
}
class TwoCnt extends Cnt { int bak;
  TwoCnt(int v, int b) static true *→ res::TwoCnt⟨v, b⟩
  {this.val:=v; this.bak:=b}
  void set(int x) static this::TwoCnt⟨v, _⟩ *→ this::TwoCnt⟨x, v⟩
  {this.bak:=this.val; this.val:=x}
  void switch(int x) static this::TwoCnt⟨v, b⟩ *→ this::TwoCnt⟨b, v⟩
  {int i:=this.val; this.val:=this.bak; this.bak:=i}
}
```

Figure 2. Static and Dynamic Specifications given for Cnt and its Subclasses

algorithm for implementing Parkinson's specification compatibility (Parkinson 2005). This link shall be formally proven later in Lemma 6.1.

3. Conformance to the OO Paradigm

We present mechanisms to ensure that method overriding and method inheritance are supported in accordance with the requirements of the OO paradigm.

3.1 Behavioral Subtyping with Dynamic Specifications

Dynamic specifications are meant for the methods of a given class and its subclasses. They must conform to the behavioral subtyping principle to support method overriding (and inheritance), as defined by the requirement below:

DEFINITION 3.1 (**Behavioral Subtyping Requirement).** *Given a dynamic specification* preC $*\!\!\rightarrow$ postC *in a method* mn *in class* C *and another dynamic specification* preD $*\!\!\rightarrow$ postD *of the corresponding method* mn *in a subclass* D*. We say that the two specifications adhere to the* behavioral subtyping requirement *using* (preD $*\!\!\rightarrow$ postD) $<:_{\texttt{D}}$ (preC $*\!\!\rightarrow$ postC)*, if the following subsumption holds :* preD $*\!\!\rightarrow$ postD $<:$ (preC$\wedge$*type*(this)$<:$D $*\!\!\rightarrow$ postC).

As shown above, we can use the enhanced specification subsumption relation to check for behavioral subtyping. For an example, consider the dynamic specification of method Cnt.set and its overriding method PosCnt.set. Assuming that these dynamic specifications are given, the behavioral subtyping requirement can be checked using:

$$\textit{dynamic-spec}(\texttt{PosCnt.set}) <:_{\texttt{PosCnt}} \textit{dynamic-spec}(\texttt{Cnt.set})$$

Hence, we have:

$$\texttt{this::PosCnt}\langle\_\rangle\$\wedge\texttt{x}\geq 0 *\!\!\rightarrow \texttt{this::PosCnt\#I}\langle\texttt{x}\rangle\$ \quad <:$$
$$\texttt{this::Cnt}\langle\texttt{v}\rangle\$\wedge\texttt{x}\geq 0\wedge(\textit{type}(\texttt{this})<:\texttt{PosCnt}) *\!\!\rightarrow \texttt{this::Cnt}\langle\texttt{x}\rangle\$$$

By contravariance of preconditions, we successfully prove:

$$\texttt{this::Cnt}\langle\texttt{v}\rangle\$\wedge\texttt{x}\geq 0\wedge(\textit{type}(\texttt{this})<:\texttt{PosCnt}) \vdash$$
$$\texttt{this::PosCnt}\langle\_\rangle\$\wedge\texttt{x}\geq 0 * \Delta$$

where Δ is derived to be $\texttt{x}\geq 0$. By covariance of postconditions, we can prove:

$$\texttt{this::PosCnt\#I}\langle\texttt{x}\rangle\$*\Delta \vdash \texttt{this::Cnt}\langle\texttt{x}\rangle\$$$

Hence, the above two dynamic specifications of Cnt.set and PosCnt.set conform to the behavioral subtyping requirement.

Dynamic specifications may also be given (or derived) for *inherited methods*, especially when their static specifications have been modified. As with method overriding, we continue to expect that the behavioral subtyping requirement holds between a dynamic specification (as supertype) for a method in a class and another dynamic specification (as subtype) for the same inherited method in the subclass. Let us consider Cnt.tick and its inherited method PosCnt.tick. Though no method overriding is present, we must still ensure *dynamic-spec*(PosCnt.tick) $<:_{\texttt{PosCnt}}$ *dynamic-spec*(Cnt.tick).

3.2 Statically-Inherited Methods

Under the OO paradigm, it is possible for a method mn in a class C to be inherited into its subclass D without any overriding. Furthermore, the user is free to add a new static/dynamic specification to such an inherited method for each subclass. Such a scenario may occur for a subclass with a strengthened invariant. For each inherited method of this subclass, we anticipate a new static specification possibly using its invariant-enhanced view. An important question to ask is if there is a need to re-verify this new static specification against the body of the inherited method.

We shall first consider static specification where the receiver is specified using partial view of form $\texttt{this::c}\langle\texttt{t}, \texttt{v}^*, \texttt{p}\rangle$. For this category of static specifications, we are expecting that each method invocation by this.mn(..) does not modify any fields in the extension record and is the same as that in the original method prior to method inheritance. To support the inheritance of static specifications which use partial views for their receivers, without re-verification, we identify a category of inherited methods that is semantically equivalent (modulo the receiver) to the original method in the superclass.

DEFINITION 3.2 (**Statically-Inherited Methods).** *Given a method* mn *with body* e *from class* A *that is being inherited into a subclass* B*, we say that this method is* statically-inherited*, if the following conditions hold:*

- *it has not been overridden in the* B *subclass.*
- *for all auxiliary calls* this.mn2(..) *for which* mn$\neq$mn2*, it must be the case that* B.mn2 *is* statically-inherited *from* A.mn2.

We can show that each *statically-inherited* method is *semantically equivalent* to the original method from its superclass. The above conditions ensure this by checking that the inherited method always invokes the same sequence of semantically equivalent method calls, as that when executed with a receiver object from its superclass. With this classification for *statically-inherited* methods, we can check inherited static specifications, as follows:

DEFINITION 3.3 (**Checking Inherited Static Specifications).** *Consider a method* mn *with static specification* spA *(using a partial view for its receiver) from class* A *that is being inherited into a subclass* B *with static specification* spB. *If this method has been* statically-inherited *into subclass* B, *we only need to check for specification subsumption* spA<:spB. *Otherwise, we have to re-verify the method body of* mn *with the new static specification* spB.

As an example, PosCnt.tick is statically-inherited from Cnt.tick (with a partial view $\texttt{this::Cnt}\langle\texttt{v}\rangle$), and we can conclude that both methods are semantically equivalent modulo the receiver. To avoid the re-verification of the static specification of PosCnt.tick, we only need to check for the following subtyping:

$$\textit{static-spec}(\texttt{Cnt.tick}) <: \textit{static-spec}(\texttt{PosCnt.tick})$$

Some other methods, such as PosCnt.get, FastCnt.get, FastCnt.set, TwoCnt.tick and TwoCnt.get, are also statically-inherited. For a counterexample that is <u>not</u> statically-inherited, consider:

```
class A {
  int foo { return this.goo() }
  int goo { return 1 }
}

class B extends A {
  int goo { return 2 }
}
```

The foo method cannot be statically-inherited in subclass B, since it invokes an auxiliary goo method that is *not* statically-inherited (in this case overridden). In other words, method B.foo() is not semantically equivalent (modulo the receiver) to A.foo() since they invoke different sequences of method calls when given the same parameters except for the receiver. As a result, we expect that the static specification (with partial view) for B.foo must be re-verified against its inherited method body from A.foo.

We have two solutions for handling methods that are not statically-inherited. One solution is to transform each method that is *not* statically-inherited into an overriding method. This is achieved by cloning the method declaration for each such method in its subclass. By doing so, we force code re-verification to be performed for the cloned methods, when inheriting static specifications into such non statically-inherited methods. A second solution is to utilize full views on the receivers of static specifications. By using full views on receivers, we shall be handling each method invocation of the form this.mn2(...) by using its corresponding dynamic specification. As a consequence, each such static specification (with full views on receiver) can always be inherited into any subclass without the need for re-verification, regardless of whether the method is statically-inherited or not. However, some loss in precision may occur since dynamic specifications are now used by each receiver during the verification of its method's body.

4. Deriving Specifications

While a static specification can give better precision, having to maintain both static and dynamic specifications may seem like more human effort is required by our approach to OO verification. To alleviate this, we provide the following set of derivation techniques that can be used, where needed.

- derive dynamic specifications from static counterparts.
- refine dynamic specifications to meet behavioral subtyping.
- inherit static specifications from method of superclass.

Let us initially assume that <u>none</u> of the dynamic specifications are given for our running example. We first present a simple technique for deriving a dynamic specification from its static counterpart, as follows:

DEFINITION 4.1 (**From Static to Dynamic Specification).** *Given a static specification* specS *for class* C, *we shall derive its dynamic counterpart* specD, *as follows:*

$$\begin{array}{rcl} \texttt{specD} & = & \rho_{\texttt{C}}\ \texttt{specS} \quad \textit{where} \\ \rho_{\texttt{C}} & = & [\texttt{this::C}\langle\texttt{v}^*\rangle\mapsto\texttt{this::C}\langle\texttt{v}^*\rangle\$, \\ & & \ \texttt{this::C\#I}\langle\texttt{v}^*\rangle\mapsto\texttt{this::C\#I}\langle\texttt{v}^*\rangle\$] \end{array}$$

Some examples of dynamic specifications that can be automatically derived from their static counterparts are:

$$\begin{array}{l} \textit{dynamic-spec}(\texttt{Cnt.get}) = \rho_{\texttt{Cnt}}\ \textit{static-spec}(\texttt{Cnt.get}) \\ \quad = \texttt{this::Cnt}\langle\texttt{v}\rangle\$ \ast\!\!\longrightarrow \texttt{this::Cnt}\langle\texttt{v}\rangle\$ \wedge \texttt{res=v} \\ \textit{dynamic-spec}(\texttt{Cnt.tick}) = \rho_{\texttt{Cnt}}\ \textit{static-spec}(\texttt{Cnt.tick}) \\ \quad = \texttt{this::Cnt}\langle\texttt{v}\rangle\$ \ast\!\!\longrightarrow \texttt{this::Cnt}\langle\texttt{v+1}\rangle\$ \\ \textit{dynamic-spec}(\texttt{PosCnt.get}) = \rho_{\texttt{PosCnt}}\ \textit{static-spec}(\texttt{PosCnt.get}) \\ \quad = \texttt{this::PosCnt\#I}\langle\texttt{v}\rangle\$ \ast\!\!\longrightarrow \texttt{this::PosCnt\#I}\langle\texttt{v}\rangle\$ \wedge \texttt{res=v} \\ \textit{dynamic-spec}(\texttt{FastCnt.tick}) \\ \quad = \rho_{\texttt{FastCnt}}\ \textit{static-spec}(\texttt{FastCnt.tick}) \\ \quad = \texttt{this::FastCnt}\langle\texttt{v}\rangle\$ \ast\!\!\longrightarrow \texttt{this::FastCnt}\langle\texttt{v+2}\rangle\$ \end{array}$$

This technique can help us derive dynamic specifications that are almost identical to static specifications, and are especially relevant for methods (e.g. in final classes) where overriding is not possible. However, these automatically derived dynamic specifications may fail to meet the behavioral subtyping requirement. Failure of behavioral subtyping can be due to two possible reasons:

1. Dynamic specification of method in superclass is too strong, or
2. Dynamic specification of method in subclass is too weak.

We propose two refinement techniques for related pairs of dynamic specifications to help them conform to behavioral subtyping. A conventional way is to use *specification inheritance* (or *specialization*) to strengthen the dynamic specification of the overriding method. However, in our approach, this technique of strengthening the dynamic specifications of a method in the subclass may violate a key requirement that the dynamic specification be a supertype of its static counterpart. Thus, prior to using specification specialization, we must either check that each inherited dynamic specification is indeed a supertype of the static specification from the overriding method, or can be made to inherit the static specification from the overridden method, as follows :

DEFINITION 4.2 (**Specification Specialization).** *Given a dynamic specification* $\texttt{preD}_{\texttt{A}} \ast\!\!\longrightarrow \texttt{postD}_{\texttt{A}}$ *and its static specification* $\texttt{preS}_{\texttt{A}} \ast\!\!\longrightarrow \texttt{postS}_{\texttt{A}}$ *for a method* mn *in class* A, *and its overriding method in a subclass* B *with static specification* $\texttt{preS}_{\texttt{B}} \ast\!\!\longrightarrow \texttt{postS}_{\texttt{B}}$. *A dynamic specification* $(\texttt{preD}_{\texttt{A}} \wedge \textit{type}(\texttt{this}) <: \texttt{B} \ast\!\!\longrightarrow \texttt{postD}_{\texttt{A}})$ *can be added to the overriding method of the* B *subclass if either of the following occurs:*

- $\texttt{preS}_{\texttt{B}} \ast\!\!\longrightarrow \texttt{postS}_{\texttt{B}} <:_{\texttt{B}} \texttt{preD}_{\texttt{A}} \ast\!\!\longrightarrow \texttt{postD}_{\texttt{A}}$ *holds, or*
- $\rho_{\texttt{A}\to\texttt{B}}(\texttt{preS}_{\texttt{A}} \ast\!\!\longrightarrow \texttt{postS}_{\texttt{A}})$ *can be inherited into the static specification of* mn *in class* B *and successfully verified.*

Note that $\rho_{\texttt{A}\to\texttt{B}} = [\texttt{this::A}\langle\texttt{v}^*\rangle\mapsto\texttt{this::B}\langle\texttt{v}^*,\texttt{w}^*\rangle\ ,\texttt{this::A\#I}\langle\texttt{v}^*\rangle \mapsto\texttt{this::B}\langle\texttt{v}^*,\texttt{w}^*\rangle\ast\delta_{\texttt{A}}]$ *where* $\texttt{w}^*$ *are free variables of the extended record, while* $\delta_{\texttt{A}}$ *captures the invariant of* A *class. The refined dynamic specification for the overriding method is obtained via intersection type,* $(\texttt{preD}_{\texttt{B}} \ast\!\!\longrightarrow \texttt{postD}_{\texttt{B}}) \ \bigwedge\ (\texttt{preD}_{\texttt{A}} \wedge \textit{type}(\texttt{this}) <: \texttt{B} \ast\!\!\longrightarrow \texttt{postD}_{\texttt{A}})$.

As an example, the pair of dynamic specifications for Cnt.get and PosCnt.get do not conform to behavioral subtyping. We may therefore attempt to strengthen the dynamic specification of PosCnt.get by specification specialization through the following multi-specification:

$$\texttt{this}::\texttt{PosCnt}\#\texttt{I}\langle\texttt{v}\rangle\$ \ast\!\!\rightarrow \texttt{this}::\texttt{PosCnt}\#\texttt{I}\langle\texttt{v}\rangle\$ \wedge \texttt{res}{=}\texttt{v} \ \bigwedge$$
$$\texttt{this}::\texttt{Cnt}\langle\texttt{v}\rangle\$\wedge \textit{type}(\texttt{this}){<:}\texttt{PosCnt} \ast\!\!\rightarrow \texttt{this}::\texttt{Cnt}\langle\texttt{v}\rangle\$\wedge\texttt{res}{=}\texttt{v}$$

However, the inherited dynamic specification from Cnt.get is *not* a supertype of *static-spec*(PosCnt.get). Hence, in order to proceed with this refinement, we must also inherit the static specification of *static-spec*(Cnt.get) into PosCnt.get, as follows:

$$\texttt{this}::\texttt{PosCnt}\#\texttt{I}\langle\texttt{v}\rangle \ast\!\!\rightarrow \texttt{this}::\texttt{PosCnt}\#\texttt{I}\langle\texttt{v}\rangle \wedge \texttt{res}{=}\texttt{v} \ \bigwedge$$
$$\texttt{this}::\texttt{PosCnt}\langle\texttt{v}\rangle \ast\!\!\rightarrow \texttt{this}::\texttt{PosCnt}\langle\texttt{v}\rangle \wedge \texttt{res}{=}\texttt{v}$$

This strengthened static specification is now a subtype of the correspondingly derived dynamic specification. Furthermore, behavioral subtyping holds between the new dynamic specifications of Cnt.get and PosCnt.get. A caveat about specification specialization is that the strengthened static specification of the method in the subclass may *not* always guarantee the invariant property. For example, $\texttt{this}::\texttt{PosCnt}\#\texttt{I}\langle\texttt{v}\rangle \ast\!\!\rightarrow \texttt{this}::\texttt{PosCnt}\#\texttt{I}\langle\texttt{v}\rangle \wedge \texttt{res}{=}\texttt{v}$ guarantees that the class invariant of PosCnt is preserved, but not $\texttt{this}::\texttt{PosCnt}\langle\texttt{v}\rangle \ast\!\!\rightarrow \texttt{this}::\texttt{PosCnt}\langle\texttt{v}\rangle \wedge \texttt{res}{=}\texttt{v}$. It is thus possible for successfully verified calls of this method to violate the class invariant property, but the above multi-specification is fully aware of when each such violation occurs through the use of different predicates. This violation of a class invariant is one reason why Findler et al. (2001) considered specification inheritance to be a potentially 'unsound' derivation technique.

As a complement to specification specialization, we propose a dual mechanism that weakens the specification of the overridden method instead. We refer to this new technique as *specification abstraction*. Instead of an intersection type, we use a *union type* to obtain a weaker dynamic specification for the overridden method. Formally:

DEFINITION 4.3 (**Specification Abstraction).** *Given a dynamic specification* $\texttt{preD}_\texttt{A} \ast\!\!\rightarrow \texttt{postD}_\texttt{A}$ *for a method* mn *in class* A*, and its overriding method in a subclass* B *with dynamic specification* $\texttt{preD}_\texttt{B} \ast\!\!\rightarrow \texttt{postD}_\texttt{B}$*. If behavioral subtyping does not hold between these dynamic specifications, we can generalise the specification of the overridden method using the following union type:*

$$\textit{dynamic-spec}(\texttt{A.mn}) = (\texttt{preD}_\texttt{A} \ast\!\!\rightarrow \texttt{postD}_\texttt{A})$$
$$\bigvee \rho_{\texttt{B}\rightarrow\texttt{A}}(\texttt{preD}_\texttt{B}) \ast\!\!\rightarrow \exists \texttt{w}^*\cdot\rho_{\texttt{B}\rightarrow\texttt{A}}(\texttt{postD}_\texttt{B})$$
$$\rho_{\texttt{B}\rightarrow\texttt{A}} = [\texttt{this}::\texttt{B}\langle\texttt{v}^*,\texttt{w}^*\rangle\${\mapsto}\texttt{this}::\texttt{A}\langle\texttt{v}^*\rangle\$,\ \texttt{this}::\texttt{B}\#\texttt{I}\langle\texttt{v}^*,\texttt{w}^*\rangle\${\mapsto}\texttt{this}::\texttt{A}\#\texttt{I}\langle\texttt{v}^*\rangle\$\ast\delta_\texttt{B}]$$

We refer to this process as specification abstraction*. It is a safe operation that weakens the dynamic specification of an overridden method to the point where behavioral subtyping holds.*

As an example, consider the derived dynamic specifications from a pair of methods Cnt.tick and FastCnt.tick where behavioral subtyping does not currently hold. We are unable to apply specification specialization, as the inherited static specification of Cnt.tick cannot be verified by the overriding method of FastCnt.tick. However, with the help of specification abstraction, we can obtain the following union type for *dynamic-spec*(Cnt.tick) instead.

$$\texttt{this}::\texttt{Cnt}\langle\texttt{v}\rangle\$ \ast\!\!\rightarrow \texttt{this}::\texttt{Cnt}\langle\texttt{v}{+}1\rangle\$ \bigvee$$
$$\texttt{this}::\texttt{Cnt}\langle\texttt{v}\rangle\$ \ast\!\!\rightarrow \texttt{this}::\texttt{Cnt}\langle\texttt{v}{+}2\rangle\$$$

Our current separation logic prover is able to directly handle intersection types but *not* union types for its multi-specifications. We propose to handle union type by the following translation instead:

$$(\texttt{pre}_1 \ast\!\!\rightarrow \texttt{post}_1) \bigvee (\texttt{pre}_2 \ast\!\!\rightarrow \texttt{post}_2)$$
$$\Longrightarrow (\texttt{pre}_1 \wedge \texttt{pre}_2) \ast\!\!\rightarrow (\texttt{post}_1 \vee \texttt{post}_2)$$

For brevity, we shall omit the formal details of how normalization (of separation logic formulae) is carried out for the above translation. In the case of Cnt.tick, we can perform normalization to obtain the following weakened dynamic specification:

$$\texttt{this}::\texttt{Cnt}\langle\texttt{v}\rangle\$ \ast\!\!\rightarrow \texttt{this}::\texttt{Cnt}\langle\texttt{w}\rangle\$ \wedge (\texttt{w}{=}\texttt{v}{+}1{\vee}\texttt{w}{=}\texttt{v}{+}2)$$

It would appear that the use of specification abstraction loses modularity, due to its dependence on the dynamic specifications of overriding methods. However, this is not true. Firstly, the purpose of specification abstraction is to derive dynamic specifications which need not be re-verified. Secondly, we maintain modularity as each static specification is verified once, but need only be re-verified when the specifications it depends on change. Though changes may occur for a dynamic specification that a method depends on; the necessity for re-verification is analogous to a modular compilation system which re-compiles a module whenever the type interface it depends on changes.

While our approach can theoretically derive all dynamic specifications, we shall also allow the option for users to directly specify dynamic specifications, where required. This option is especially helpful in supporting modular open-ended classes that could be further extended with new subclasses. Our overall procedure for selectively but automatically deriving dynamic specifications shall be as follows:

DEFINITION 4.4 (**Deriving Dynamic Specifications).** *We derive and refine dynamic specifications, as follows:*

- *If the dynamic specifications of both overridden and overriding methods are given, check for the behavioral subtyping requirement.*
- *If only the dynamic specification of an overridden method is given, derive the dynamic specification of the overriding method and then use* specification specialization *to refine it.*
- *If only the dynamic specification of an overriding method is given, derive the dynamic specification of the overridden method and then use* specification abstraction *to refine it.*
- *Otherwise, derive both dynamic specifications and then use* specification abstraction *to refine the dynamic specification of the overridden method in the superclass.*

Note that the procedure is geared towards the preservation of class invariants, where possible, as it favours specification abstraction over specification specialization.

Lastly, it may also be possible for static specifications to be omitted for some statically-inherited methods. We propose a way to derive static specifications for such methods, as follows:

DEFINITION 4.5 (**Deriving Static Specifications).** *Given a method* mn *from class* A *with static specification* spA*, and a subclass* B *where the same method has been* statically-inherited*. If no static specification is given for* B.mn*, we can derive a static specification for it, as follows :*

$$\textit{static-spec}(\texttt{B.mn}) = [\texttt{this}::\texttt{A}\langle\texttt{v}^*\rangle{\mapsto}\texttt{this}::\texttt{B}\langle\texttt{v}^*,\texttt{w}^*\rangle]\ \texttt{spA}$$

The extra fields, $\texttt{w}^*$*, in the subclass are never modified by each statically-inherited method.*

The above substitution is only applicable for partial views, and it is not needed for full views which will remain unchanged when deriving static specifications.

Though specification derivation techniques are important aids that make it easier for users to adopt our OO verification methodology, they are not fundamental in the current work. In the rest of this

$$
\begin{array}{lll}
P & ::= \mathit{tdecl}^{*}\ e & \mathit{tdecl} ::= \mathit{classt} \mid \mathit{viewt} \\
\mathit{classt} & ::= \multicolumn{2}{l}{\mathtt{class}\ c_1\ \mathtt{extends}\ c_2\ \mathtt{inv}\ \kappa{\wedge}\pi\ \{\ (t\ v)^{*} \mathit{meth}^{*}\ \}} \\
\tau & ::= \mathtt{int} \mid \mathtt{bool} \mid \mathtt{void} & t ::= c \mid \tau \\
\mathit{viewt} & ::= \mathtt{view}\ c\langle v^{*}\rangle \equiv \Phi\ \mathtt{inv}\ \pi & \mathit{sp} ::= \bigwedge (\Phi_{pr} *\!\!\!\longrightarrow \Phi_{po})^{*} \\
\mathit{meth} & ::= \multicolumn{2}{l}{t\ \mathit{mn}\ ((t\ v)^{*})\ [\mathtt{static}\ \mathit{sp}_1]\ [\mathtt{dynamic}\ \mathit{sp}_2]\ [\{e\}]} \\
e & ::= \multicolumn{2}{l}{\mathtt{null} \mid k \mid v \mid v.f \mid v{:=}e \mid v_1.f{:=}v_2 \mid \mathtt{new}\ c(v^{*}) \mid v{:}c} \\
 & \quad \multicolumn{2}{l}{\mid e_1; e_2 \mid t\ v;\ e \mid v.\mathit{mn}(v^{*}) \mid \mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2} \\
 & \quad \multicolumn{2}{l}{\mid (c)\ v \mid \mathtt{while}\ v\ \mathtt{where}\ \mathit{sp}\ \mathtt{do}\ e} \\
\Phi & ::= \bigvee (\exists v^{*} \cdot \kappa{\wedge}\pi)^{*} & \pi ::= \gamma{\wedge}\phi{\wedge}\beta \\
\gamma & ::= \multicolumn{2}{l}{v_1{=}v_2 \mid v{=}\mathtt{null} \mid v_1{\neq}v_2 \mid v{\neq}\mathtt{null} \mid \gamma_1{\wedge}\gamma_2} \\
\kappa & ::= \mathtt{emp} \mid v{::}c\langle v^{*}\rangle \mid \kappa_1 * \kappa_2 & \beta ::= v{=}c \mid v{<:}c \\
\Delta & ::= \multicolumn{2}{l}{\Phi \mid \Delta_1{\vee}\Delta_2 \mid \Delta{\wedge}\pi \mid \Delta_1{*}\Delta_2 \mid \exists v{\cdot}\Delta} \\
\phi & ::= \multicolumn{2}{l}{\mathtt{true} \mid \mathtt{false} \mid a_1{=}a_2 \mid a_1{\leq}a_2 \mid c{<}v \mid \phi_1{\wedge}\phi_2} \\
 & \quad \multicolumn{2}{l}{\mid \phi_1{\vee}\phi_2 \mid \neg\phi \mid \exists v \cdot \phi \mid \forall v \cdot \phi} \\
a & ::= \multicolumn{2}{l}{k \mid v \mid k{\times}a \mid a_1{+}a_2 \mid -a \mid \mathit{max}(a_1,a_2) \mid \mathit{min}(a_1,a_2)}
\end{array}
$$

Figure 3. A Core Object-Oriented Language

paper, we shall assume that all required dynamic and static specifications are available, and proceed to describe core components of our enhanced OO verification system.

5. Enhanced OO Verification

We shall now formalise our verification system. We consider a simple sequential language with just the basic features from the OO paradigm. Some omitted features, such as exceptions, static fields and static methods, can be handled in an orthogonal manner and do not cause any difficulty to our verification system.

5.1 A Core OO Language

We provide a simple OO language in Figure 3, and assume that type-checking is done on the program and specified constraints prior to verification. This core language is the target of some preprocessing steps. A program consists of a list of class and view declarations and an expression which corresponds to the `main` method in many languages. We assume that the super class of each class is explicitly declared, except for `Object` at the top of the class hierarchy. We also use `this` as a special variable referring to the receiver object, and `super` to refer to a superclass's method invocation. For each view definition, we declare an invariant π over the parameters $\{\mathtt{root}, v^{*}\}$ that is valid for each instance of the view. Also, Φ is a normalised form of Δ, γ captures pointer constraint, β type information, ϕ arithmetic constraints, while π a pure formula without any heap. Each method *meth* and `while` loop is declared with intersection type(s) of the form $\bigwedge(\Phi_{pr} *\!\!\!\longrightarrow \Phi_{po})^{*}$. For simplicity, we assume that variable names declared in each method are all distinct and use the pass-by-value parameter mechanism. Primed notation is used to capture the latest value of local variables and may appear in the postcondition of loops. For example :

$$\mathtt{while}\ \mathtt{x}{<}0\ \mathtt{where}\ \mathtt{true} *\!\!\!\longrightarrow (\mathtt{x}{>}0{\wedge}\mathtt{x}'{=}\mathtt{x}) \vee (\mathtt{x}{\leq}0{\wedge}\mathtt{x}'{=}0)$$
$$\mathtt{do}\ \{\ \mathtt{x}{:=}\mathtt{x}{+}1\ \}$$

Here `x` and $\mathtt{x}'$ denote the old and new values of variable `x` at the entry and exit of the loop, respectively. Note that precondition `true` captures the loop's invariant, while postcondition $(\mathtt{x}{>}0{\wedge}\mathtt{x}'{=}\mathtt{x}) \vee (\mathtt{x}{\leq}0{\wedge}\mathtt{x}'{=}0)$ captures the loop's effects.

5.2 Verification System

Our verification system for OO programs is implemented in a modular fashion. It processes the class declarations in a top-down manner whereby the methods of superclasses are verified before those of the subclasses. We shall assume that static specifications are given, and that dynamic specifications are already given (or automatically derived). Also, for each method that is *not* statically inherited in a subclass, we shall clone the method for that subclass. There are three major subsystems present, namely: (i) View Generator, (ii) Inheritance Checker, and (iii) Code Verifier. These are elaborated next.

5.2.1 View Generator

For each subclass in the class hierarchy, we must generate a lossless upcasting rule in accordance with Defn 2.2. However, the format $\mathtt{Ext}\langle \mathtt{c}, \mathtt{v}^{*}, \mathtt{p}\rangle$ actually denotes a family of record extensions that is distinct for each subclass `c`. To distinguish them clearly in our implementation, we provide a set of specialised record extensions of the form $\mathtt{Ext_c}\langle \mathtt{v}^{*}, \mathtt{p}\rangle$ instead. With this change, we can generate the following casting rules for our running example:

$$
\begin{array}{ll}
\mathtt{PosCnt}\langle \mathtt{t}, \mathtt{n}, \mathtt{p}\rangle & \equiv \mathtt{root}{::}\mathtt{Cnt}\langle \mathtt{t}, \mathtt{n}, \mathtt{q}\rangle{*}\mathtt{q}{::}\mathtt{Ext}_{\mathtt{PosCnt}}\langle \mathtt{p}\rangle \\
\mathtt{FastCnt}\langle \mathtt{t}, \mathtt{n}, \mathtt{p}\rangle & \equiv \mathtt{root}{::}\mathtt{Cnt}\langle \mathtt{t}, \mathtt{n}, \mathtt{q}\rangle{*}\mathtt{q}{::}\mathtt{Ext}_{\mathtt{FastCnt}}\langle \mathtt{p}\rangle \\
\mathtt{TwoCnt}\langle \mathtt{t}, \mathtt{n}, \mathtt{b}, \mathtt{p}\rangle & \equiv \mathtt{root}{::}\mathtt{Cnt}\langle \mathtt{t}, \mathtt{n}, \mathtt{q}\rangle{*}\mathtt{q}{::}\mathtt{Ext}_{\mathtt{TwoCnt}}\langle \mathtt{b}, \mathtt{p}\rangle
\end{array}
$$

Correspondingly, we may also provide an `ExtAll` view for the class hierarchy. In the case of our running example, we can generate the following definition for the `ExtAll` view:

$$
\begin{array}{l}
\mathtt{ExtAll}\langle \mathtt{t_1}, \mathtt{t_2}\rangle \equiv (\mathtt{t_1}{=}\mathtt{t_2}){\wedge}\mathtt{root}{=}\mathtt{null} \\
\quad \vee\ \mathtt{root}{::}\mathtt{Ext}_{\mathtt{PosCnt}}\langle \mathtt{q}\rangle{*}\mathtt{q}{::}\mathtt{ExtAll}\langle \mathtt{PosCnt}, \mathtt{t_2}\rangle \\
\quad\quad {\wedge}\mathtt{PosCnt}{<}\mathtt{t_1}{\wedge}\mathtt{t_2}{<:}\mathtt{PosCnt} \\
\quad \vee\ \mathtt{root}{::}\mathtt{Ext}_{\mathtt{FastCnt}}\langle \mathtt{q}\rangle{*}\mathtt{q}{::}\mathtt{ExtAll}\langle \mathtt{FastCnt}, \mathtt{t_2}\rangle \\
\quad\quad {\wedge}\mathtt{FastCnt}{<}\mathtt{t_1}{\wedge}\mathtt{t_2}{<:}\mathtt{FastCnt} \\
\quad \vee\ \mathtt{root}{::}\mathtt{Ext}_{\mathtt{TwoCnt}}\langle \mathtt{b}, \mathtt{q}\rangle{*}\mathtt{q}{::}\mathtt{ExtAll}\langle \mathtt{TwoCnt}, \mathtt{t_2}\rangle \\
\quad\quad {\wedge}\mathtt{TwoCnt}{<}\mathtt{t_1}{\wedge}\mathtt{t_2}{<:}\mathtt{TwoCnt}
\end{array}
$$

Lastly, for each subclass with a non-trivial invariant, we also generate two invariant-enhanced views for this subclass. For our running example, only the subclass `PosCnt` has an invariant. Hence, our generator will provide the following:

$$\mathtt{PosCnt\#I}\langle \mathtt{t}, \mathtt{n}, \mathtt{p}\rangle \equiv \mathtt{root}{::}\mathtt{PosCnt}\langle \mathtt{t}, \mathtt{n}, \mathtt{q}\rangle{\wedge}\mathtt{n}{\geq}0$$

In summary, the above shows how we explicitly generate predicate views for casting and class invariants. In practice, our prototype verification system creates these views on demand during entailment checking itself.

5.2.2 Inheritance Checker

This subsystem ensures that specifications of added methods are consistent with class inheritance and method overriding requirements. Whenever a new subclass `B` is added, we expect a set of new overriding methods and another set of statically-inherited methods. We propose to check for consistency, as follows:

- Firstly, we check that each static specification is a subtype of the dynamic specification.
- For each new overriding method `B.mn`, we identify the nearest overridden method in a superclass of `B`. We then check that each given dynamic specification is a subtype of the given dynamic specification of its overridden method in its superclass.
- For each statically-inherited method `B.mn`, we check that its given static specification is a supertype of the corresponding static specification in its superclass. If a dynamic specification is also given, we check that it is a subtype of the given dynamic specification in its superclass.

Some of the static and dynamic specifications may have been automatically derived. As these derived specifications are correct by construction, we shall not be checking for the specification subsumption relation amongst them.

5.2.3 Code Verifier

To support verification, we shall use Hoare-style rule of the form $\vdash \{\Delta_1\}e\{\Delta_2\}$. This rule is applied in a forward manner. Given a heap state Δ_1 and an expression e, we expect the above verification to succeed and also produce a poststate Δ_2. There are four features in our core language that are peculiar to the OO paradigm, namely (i) the object constructors, (ii) the cast constructs, (iii) instance method invocations and (iv) super calls. Let us discuss how they are handled.

For the object constructor, we use the following rule where each primed variable $\mathtt{v}'_\mathtt{i}$ captures the latest value of variable $\mathtt{v_i}$:

$$\frac{\Delta_1{=}\Delta * \mathtt{res}{::}\mathtt{c}\langle \mathtt{c}, v'_1, .., v'_n, \mathtt{null}\rangle}{\vdash \{\Delta\}\, \mathtt{new}\ \mathtt{c}(v_1, .., v_n)\, \{\Delta_1\}}$$

This rule produces an object of actual type c using partial view.

Consider a cast construct $(\mathtt{c})(\mathtt{v{:}c_1})$ where $\mathtt{v{:}c_1}$ captures the compile-time type of v inserted before verification. We shall treat it as being equivalent to a primitive call of the form:

$$\begin{array}{ll}\mathtt{c}\ \mathtt{cast_c}\ (\mathtt{c_1}\ \mathtt{v})\ \textbf{static} & \mathtt{v}{::}\mathtt{c_1}\langle \mathtt{t}, ..\rangle \ast\!\!\!\longrightarrow \mathtt{v}{::}\mathtt{c_1}\langle \mathtt{t}, ..\rangle {\wedge} \mathtt{t}{<:}\mathtt{c} \\ & \wedge\ \mathtt{true} \ast\!\!\!\longrightarrow \mathtt{true}\end{array}$$

The above declaration allows the cast construct to possibly fail at runtime. If casting succeeds, we may expect that the actual type of the object to be a subtype of c, as captured by the first pre/post annotation. The second pre/post annotation is added for completeness, and may be used if we are unable to establish the heap state of v.

Another important feature to consider is instance method call of the form $(\mathtt{v{:}c}).\mathtt{mn}(\mathtt{v_1..v_n})$. We first identify the best possible type of v using $\beta{=}\mathtt{findtype}(\Delta, \mathtt{v{:}c})$. The result β will tell us if we have the actual type $\mathtt{t{=}c_1}$ or the best static type $\mathtt{t{<:}c_1}$ where $\mathtt{t}{=}\mathit{type}(\mathtt{v})$ and $\mathtt{c_1{<:}c}$. Note that $\mathtt{c_1}$ can be more precise than the compile-time type c due to our use of flow- and path-sensitive reasoning. If the actual type is known, we choose the static specification of method mn from class $\mathtt{c_1}$. Otherwise, we choose its dynamic specification instead. This decision is captured by $\mathtt{spec}{=}\mathtt{findspec}(\mathtt{P}, \beta, \mathtt{mn})$ where P denotes the entire OO program. The overall rule is:

$$\frac{\begin{array}{c}\beta{=}\mathtt{findtype}(\Delta, v{:}c) \qquad \rho = [v_1{\mapsto}v'_1, ..., v_n{\mapsto}v'_n] \\ \mathtt{findspec}(\mathtt{P}, \beta, \mathtt{mn}) = \bigwedge_{i=1}^{m}(\mathtt{pre_i} \ast\!\!\!\longrightarrow \mathtt{post_i}) \\ \exists i \in 1..m\ \cdot\ (\Delta {\vdash} (\rho\, \mathtt{pre_i}) * \Delta_i \quad \Delta_r = (\Delta_i * \mathtt{post}_i))\end{array}}{\vdash \{\Delta\}\, (v{:}c).\mathtt{mn}(v_1..v_n)\, \{\Delta_r\}}$$

If Δ is a disjunctive formula with different types for v, we can use findtype/findspec operations in the entailment procedure, so that the best specification is selected for either the actual or the static type of the object at v for each disjunct. For multi-specifications, we choose the first specification whose precondition holds. We assume that these multiple specifications are ordered to yield a more precise result ahead of the less precise ones.

We can easily deal with the invocation of super methods. This feature can be used in place of the receiver this parameter to refer to the overridden method. It can be easily handled by our approach since super method calls are essentially *static* calls that can be precisely captured by static specifications. Consider an overridden method mn in a superclass A and a call super.mn(..) being used in an overriding method in subclass B. We can handle this super call by re-writing it to this.A.mn(..). In this case, our verification process will select the *static* specification of the overridden method in class A to use. Past works, such as Parkinson and Bierman (2005); Kiniry et al. (2005), do <u>not</u> handle super method calls for verification well, as there is an inherent mismatch between super method calls (which are static calls) and the mechanism based on dynamic specifications.

6. Correctness

There are several soundness results that are needed to show the overall safety of our verification system.

The semantics of our constraints is that of separation logic (Reynolds 2002), with extensions to handle our shape views. To define the semantic model we assume sets *Loc* of locations (positive integer values), *Val* of primitive values, with $0 \in \mathit{Val}$ denoting null, *Var* of variables (program variables and other meta variables), and *ObjVal* of object values stored in the heap, with $c[f_1{\mapsto}\nu_1, .., f_n{\mapsto}\nu_n]$ denoting an object value of data type c where $\nu_1, .., \nu_n$ are current values (from the domain $\mathit{Val} \cup \mathit{Loc}$) of the corresponding fields $f_1, .., f_n$. This object denotation shall be abbreviated as $c(\nu_1, .., \nu_n)$. Let $s, h \models \Phi$ denote that stack s and heap h form a model of the constraint Φ, with h, s from the following concrete domains:

$$\begin{array}{llll} h & \in \mathit{Heaps} & =_{df} & \mathit{Loc} \rightharpoonup_{fin} \mathit{ObjVal} \\ s & \in \mathit{Stacks} & =_{df} & \mathit{Var} \rightarrow \mathit{Val} \cup \mathit{Loc}\end{array}$$

As we use a first-order language, lexical scoping can be easily enforced by only allowing elements from the topmost stack frame to be accessible at runtime. A complete definition of the model for separations constraints can be found in Nguyen et al. (2007).

We use a small-step dynamic semantics for our language (Fig. 3) but extended with pass-by-reference parameters. For simplicity, we assume that all while loops have been transformed to equivalent tail-recursive methods with the help of pass-by-reference parameters. The machine configuration is represented by $\langle s, h, e\rangle$ where s denotes the current stack, h denotes the current heap, and e denotes the current program code. The semantics assumes unlimited stack and heap spaces. Each reduction step can then be formalized as a small-step transition of the form: $\langle s, h, e\rangle {\hookrightarrow} \langle s_1, h_1, e_1\rangle$. The full set of transitions is given in Fig. 4. We have introduced an intermediate construct $\mathtt{ret}(v^*, e)$, with e to denote the residual code of its call, to model the outcome of call invocation. It is also used to handle local blocks.

The following lemma highlights a key result showing that our use of specification subsumption relation is sound for avoiding re-verification, as follows:

LEMMA 6.1 (**Soundness of Enhanced Spec. Subsumption).** *Given that code* e *has been successfully verified using* $\mathtt{preB} \ast\!\!\!\longrightarrow \mathtt{postB}$. *If specification subsumption* $\mathtt{preB} \ast\!\!\!\longrightarrow \mathtt{postB} <: \mathtt{preA} \ast\!\!\!\longrightarrow \mathtt{postA}$ *holds, then its specification supertype* $\mathtt{preA} \ast\!\!\!\longrightarrow \mathtt{postA}$ *is guaranteed to verify successfully for the same* e *code.*

Proof: From the premise of specification subsumption (Defn 2.5), we can obtain: $\mathtt{preA}{\vdash}\mathtt{preB} * \Delta$ and $\mathtt{postB}{*}\Delta{\vdash}\mathtt{postA}$. In our context, preconditions preA, preB and their entailment's residual Δ do not contain any primed variables, while only primed variables are modified indirectly by our program. Hence, adding a formula with only unprimed variables, such as Δ, to both pre/post always satisfies the side condition of the frame rule. Let e denote the method body which has been preprocessed to a form where pass-by-value parameters are never modified. Let $\{\mathtt{v_1}, .., \mathtt{v_n}\}$ denote the set of free variables in e, and let $\mathtt{N}{=}\bigwedge_{\mathtt{i}=1}^{\mathtt{n}}(\mathtt{v}'_\mathtt{i}{=}\mathtt{v_i})$. From the premise that $\mathtt{preB} \ast\!\!\!\longrightarrow \mathtt{postB}$ is a verified specification for the code, we have $\vdash \{\mathtt{preB}{\wedge}\mathtt{N}\}\, \mathtt{e}\, \{\mathtt{postB}\}$. In order to show that its specification supertype $\mathtt{preA} \ast\!\!\!\longrightarrow \mathtt{postA}$ is also verifiable for the same code, we need to derive $\vdash \{\mathtt{preA}{\wedge}\mathtt{N}\}\, \mathtt{e}\, \{\mathtt{postA}\}$. We conclude based on the following steps:

$\vdash \{\mathtt{preB}{\wedge}\mathtt{N}\}\, \mathtt{e}\, \{\mathtt{postB}\}$	premise
$\vdash \{\mathtt{preB}{\wedge}\mathtt{N}{*}\Delta\}\, \mathtt{e}\, \{\mathtt{postB}{*}\Delta\}$	frame rule
$\vdash \{\mathtt{preA}{\wedge}\mathtt{N}\}\, \mathtt{e}\, \{\mathtt{postB}{*}\Delta\}$	precondition strengthening
$\vdash \{\mathtt{preA}{\wedge}\mathtt{N}\}\, \mathtt{e}\, \{\mathtt{postA}\}$	postcondition weakening □

The above proof uses the following Consequence Lemma stating the soundness of precondition strengthening and postcondition weakening:

LEMMA 6.2 (**Consequence Rule).** *The following verification holds:*

$$\frac{P'{\vdash}P \quad \vdash \{P\}\, \mathtt{e}\, \{Q\} \quad Q{\vdash}Q'}{\vdash \{P'\}\, \mathtt{e}\, \{Q'\}}$$

$$\langle s,h,v\rangle\hookrightarrow\langle s,h,s(v)\rangle \quad \langle s,h,k\rangle\hookrightarrow\langle s,h,k\rangle \quad \langle s,h,v.f\rangle\hookrightarrow\langle s,h,h(s(v))(f)\rangle \quad \langle s,h,v{:=}k\rangle\hookrightarrow\langle s[v\mapsto k],h,()\rangle$$

$$\langle s,h,();e\rangle\hookrightarrow\langle s,h,e\rangle \quad \langle s,h,\{t\ v;\ e\}\rangle\hookrightarrow\langle [v\mapsto\_]{+}s,h,\mathtt{ret}(v,e)\rangle \quad \langle s,h,\mathtt{ret}(v^*,k)\rangle\hookrightarrow\langle s{-}\{v^*\},h,k\rangle$$

$$\frac{\langle s,h,e_1\rangle\hookrightarrow\langle s_1,h_1,e_3\rangle}{\langle s,h,e_1;e_2\rangle\hookrightarrow\langle s_1,h_1,e_3;e_2\rangle} \quad \frac{s(v){=}\mathtt{true}}{\langle s,h,\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rangle\hookrightarrow\langle s,h,e_1\rangle} \quad \frac{s(v){=}\mathtt{false}}{\langle s,h,\mathtt{if}\ v\ \mathtt{then}\ e_1\ \mathtt{else}\ e_2\rangle\hookrightarrow\langle s,h,e_2\rangle}$$

$$\frac{\langle s,h,e\rangle\hookrightarrow\langle s_1,h_1,e_1\rangle}{\langle s,h,v{:=}e\rangle\hookrightarrow\langle s_1,h_1,v{:=}e_1\rangle} \quad \frac{\langle s,h,e\rangle\hookrightarrow\langle s_1,h_1,e_1\rangle}{\langle s,h,\mathtt{ret}(v^*,e)\rangle\hookrightarrow\langle s_1,h_1,\mathtt{ret}(v^*,e_1)\rangle} \quad \frac{r=h(s(v_1))[f\mapsto s(v_2)] \quad h_1=h[s(v_1)\mapsto r]}{\langle s,h,v_1.f:=v_2\rangle\hookrightarrow\langle s,h_1,()\rangle}$$

$$\frac{\begin{array}{c}\mathit{fields}(c)=[t_1\ f_1,..,t_n\ f_n] \quad \iota\notin\mathit{dom}(h)\\ r{=}c[f_1\mapsto s(v_1),..,f_n\mapsto s(v_n)]\end{array}}{\langle s,h,\mathtt{new}\ c(v_1,,v_n)\rangle\hookrightarrow\langle s,h{+}[\iota\mapsto r],\iota\rangle} \quad \frac{\begin{array}{c}s_1{=}[w_i\mapsto s(v_i)]_{i=m}^n{+}s \quad h(s(v_0))=c[\ldots]\\ t_0\ \mathit{mn}((\mathtt{ref}\ t_i\ w_i)_{i=1}^{m-1},(t_i\ w_i)_{i=m}^n)\ \{e\}\in\mathit{meth}(c)\end{array}}{\langle s,h,v_0.\mathit{mn}(v_1,..,v_n)\rangle\hookrightarrow\langle s_1,h,\mathtt{ret}(\{w_i\}_{i=m}^n,[v_i/w_i]_{i=1}^{m-1}e)\rangle}$$

$$\frac{s_1{=}[w_i\mapsto s(v_i)]_{i=m}^n{+}s \quad t_0\ \mathit{mn}((\mathtt{ref}\ t_i\ w_i)_{i=1}^{m-1},(t_i\ w_i)_{i=m}^n)\ \{e\}\in\mathit{meth}(c)}{\langle s,h,v_0.c.\mathit{mn}(v_1,..,v_n)\rangle\hookrightarrow\langle s_1,h,\mathtt{ret}(\{w_i\}_{i=m}^n,[v_i/w_i]_{i=1}^{m-1}e)\rangle}$$

Figure 4. Dynamic Semantics

Proof Sketch: Based on the premise, we have a set of s,h such that $\langle s,h,\mathtt{e}\rangle\hookrightarrow^*\langle s_1,h_1,v\rangle$ and $s,h\models P\wedge s_1{+}[\mathtt{res}\mapsto v],h_1\models \mathit{Post}(Q)$. By Galois connection, we have $s_1{+}[\mathtt{res}\mapsto v],h_1\models \mathit{Post}(Q')$. Thus, for all $s,h\models P'$, we have $\vdash\{P'\}\ \mathtt{e}\ \{Q'\}$. □

We extract the post-state of a heap constraint by:

DEFINITION 6.1 (**Poststate).** *Given a constraint Δ, $\mathit{Post}(\Delta)$ captures the relation between primed variables of Δ. That is :*

$\mathit{Post}(\Delta)\ =_{df}\ \rho\,(\exists V{\cdot}\Delta)$, *where*
$V=\{v_1,..,v_n\}$ *denotes all unprimed program variables in* Δ
$\rho=[v_1'\mapsto v_1,..,v_n'\mapsto v_n]$

The next two lemmas state some results on statically-inherited methods for which re-verification is proven not to be needed.

LEMMA 6.3 (**Equivalence of Statically-Inherited Methods).** *Consider a method* mn *from class* A *that satisfies the conditions of being statically-inherited into a* B *subclass. Assuming that*

$$\bigwedge\ \begin{array}{l}\langle s,h_1,\mathtt{o.mn(p^*)}\rangle\hookrightarrow^*\langle s_1,h_3,v\rangle\\ h_1{=}h{+}[s(\mathtt{o})\mapsto\mathtt{A}(v_1..v_n)]\\ h_3{=}h'{+}[s(\mathtt{o})\mapsto\mathtt{A}(w_1..w_n)]\end{array}$$

then

$$\bigwedge\ \begin{array}{l}\langle s,h_2,\mathtt{o.mn(p^*)}\rangle\hookrightarrow^*\langle s_1,h_4,v\rangle\\ h_2{=}h{+}[s(\mathtt{o})\mapsto\mathtt{B}(v_1..v_n,v_{n+1}..v_m)]\\ h_4{=}h'{+}[s(\mathtt{o})\mapsto\mathtt{B}(w_1..w_n,v_{n+1}..v_m)]\end{array}$$

Proof Sketch : Using the conditions of Defn 3.2, we can prove the above by an induction on the dynamic semantics (see Fig. 4) over execution of the body of statically-inherited methods. □

LEMMA 6.4 (**Soundness of Statically-Inherited Specifications).** *Consider a method* mn *from class A that has been successfully verified against its static specification* $\mathtt{preS_A} \ast\!\!\rightarrow \mathtt{postS_A}$, *and a subclass B that statically-inherits* mn *with static specification* $\mathtt{preS_B} \ast\!\!\rightarrow \mathtt{postS_B}$. *Assuming that a specification subsumption relation of the form* $\mathtt{preS_A} \ast\!\!\rightarrow \mathtt{postS_A} <: \mathtt{preS_B} \ast\!\!\rightarrow \mathtt{postS_B}$ *holds, then* B.mn *is guaranteed to verify successfully against its specification* $\mathtt{preS_B} \ast\!\!\rightarrow \mathtt{postS_B}$.

Proof Sketch : Follows from Lemmas 6.3 and 6.1. □

We shall now show a result regarding behavioral subtyping.

LEMMA 6.5 (**Soundness of Behavioral Subtyping).** *Consider a method* mn *from class* A *with dynamic specification* $\mathtt{preD_A} \ast\!\!\rightarrow \mathtt{postD_A}$ *and that*

$$\bigwedge\ \begin{array}{l}s,h_1\models\mathtt{preD_A}\\ h_1{=}h{+}[s(\mathtt{o})\mapsto\mathtt{A}(v^*)]\\ \langle s,h_1,\mathtt{o.mn(p^*)}\rangle\hookrightarrow^*\langle s_3,h_3,v\rangle\\ s_3+[\mathtt{res}\mapsto v],h_3\models\mathit{Post}(\mathtt{postD_A})\end{array}$$

If we assume a similar object from a subclass B *such that*

$$\bigwedge\ \begin{array}{l}s,h_2\models\mathtt{preD_A}\\ h_2{=}h{+}[s(\mathtt{o})\mapsto\mathtt{B}(v^*,w^*)]\end{array}$$

and we call the overriding method, then we obtain :

$$\bigwedge\ \begin{array}{l}\langle s,h_2,\mathtt{o.mn(p^*)}\rangle\hookrightarrow^*\langle s_4,h_4,v\rangle\\ s_4+[\mathtt{res}\mapsto v],h_4\models\mathit{Post}(\mathtt{postD_A})\end{array}$$

Proof Sketch : Follows from Defn 3.1 of the behavioral subtyping requirement and Lemma 6.1. □

Lastly, we prove the soundness of our verification system using preservation and progress lemmas.

LEMMA 6.6 (**Preservation).** *If*

$$\vdash\{\Delta\}\ e\ \{\Delta_2\} \quad s,h\models\mathit{Post}(\Delta) \quad \langle s,h,e\rangle\hookrightarrow\langle s_1,h_1,e_1\rangle$$

Then there exists Δ_1 such that $s_1,h_1\models\mathit{Post}(\Delta_1)$ and $\vdash\{\Delta_1\}\ e_1\ \{\Delta_2\}$.

Proof Sketch: By induction on e. □

LEMMA 6.7 (**Progress).** *If $\vdash\{\Delta\}\ e\ \{\Delta_1\}$, and $s,h\models\mathit{Post}(\Delta)$, then either e is a value, or there exist s_1,h_1, and e_1, such that*

$$\langle s,h,e\rangle\hookrightarrow\langle s_1,h_1,e_1\rangle.$$

Proof Sketch: By induction on e. □

THEOREM 6.8 (**Soundness of Verification).** *Consider a closed term e without free variables in which all methods have been successfully verified. Assuming that $\vdash\{\mathtt{true}\}\ e\ \{\Delta\}$, then either $\langle[],[],e\rangle\hookrightarrow^*\langle[],h,v\rangle$ terminates with a value v such that the following $([\mathtt{res}\mapsto v],h)\models\Delta$ holds, or it diverges $\langle[],[],e\rangle\not\hookrightarrow^*$.*

Proof Sketch: Follows from Lemma 6.6 and Lemma 6.7. □

7. Related Work

In support of modular reasoning on properties of object-oriented programs, the notion of behavioral subtyping has been intensively studied in the last two decades, e.g. Liskov (1988); America (1991); Liskov and Wing (1994); Dhara and Leavens (1996); Meyer (1997); Findler et al. (2001); Muller (2002); Parkinson (2005). The notion of specification inheritance, where an overriding method inherits the specifications of all the overridden methods, was first introduced in Eiffel (Meyer 1997). As an effort to relate these two notions, Dhara and Leavens (1996) presented a modular specification technique which automatically forces behavioral subtyping through specification inheritance. More recently, Leavens and Naumann (2006) proposed a formal characterization for behavioral subtyping and modular reasoning. The basic idea of modular reasoning, which the authors call supertype abstraction, is that reasoning about an invocation, say E.m(), is based on the specification associated with the static type of the receiver expression E. In Leavens and Naumann (2006), the authors proved the equivalence between supertype abstraction and behavioral subtyping. The new formalization is supposed to serve as a semantic foundation for object-oriented specification languages.

Various embodiments of these proposals have been implemented in both static and runtime verification tools and been applied to rich specification and programming languages such as ESC/Java (Flanagan et al. 2002), JML (Leavens et al. 2006), Spec# (Barnett et al. 2004), and ESpec (Ostroff et al. 2006). The Krakatoa tool (Marché et al. 2004; Marché and Paulin-Mohring 2005) translates JML specifications into the input language for the Why verification tool (Filliâtre 2003). Verification conditions generated by the Why tool can then be discharged by different theorem provers. However, to the best of our knowledge, neither inheritance nor method overriding is supported by their system. Software model checking frameworks (Robby et al. 2003; Hatcliff et al. 2003) have also been used in the verification of OO programs. Inference mechanisms for loop invariants have been proposed in Nimmer and Ernst (2002); Pasareanu and Visser (2004) amongst others, and they can make verification even easier to use. However, most of these works are based primarily on the idea of dynamic specifications. Even when static specifications are added, like code contracts in JML (Leavens et al. 2007, ch 15), they did not enforce an important subtyping relation between a static specification and its dynamic counterpart. Moreover, in comparison with our approach, Spec# is more restrictive in handling overriding as it does not allow any changes in the precondition of the overriding method.

Using the rules of behavioral subtyping, Findler et al. have formalized hierarchy violations and blame assignment for pre and postcondition failures (Findler and Felleisen 2001; Findler et al. 2001). They identified a problem (related to preservation of class invariants) that arises from synthesizing the specifications of overriding methods through specification inheritance. This problem is caused by specification inheritance's manner of enforcing behavioral subtyping which may wrongly assume that the original specification of an overriding method is too weak. In our proposal, we can avoid this problem by using *specification abstraction* instead of specification inheritance, if class invariants are to be preserved for the overriding methods. Furthermore, while Findler and Felleisen (2001) and Findler et al. (2001) focus on checking the correctness of contracts at run-time, we propose a static verification system.

The problem of writing specifications for programs that use various forms of modularity where the internal resources of a module should not be accessed by the module's clients, is tackled in several papers (O'Hearn et al. 2004; Parkinson and Bierman 2005; Leavens and Muller 2007). In O'Hearn et al. (2004) the internal resources of a module are hidden from its clients using a so called hypothetical frame rule, whereas in Parkinson and Bierman (2005) the notion of abstract predicates is introduced. While O'Hearn et al. (2004) only supports single instances of the hidden data structure, abstract predicates can deal with dynamic instantiation of a module. To support the reasoning of concurrent programs, a rule similar to the hypothetical frame rule is used in O'Hearn (2007) to model critical regions with resource invariants. For soundness reason, precise resource invariants were also required in O'Hearn et al. (2004) and O'Hearn (2007) due to the desire to support both the conjunction rule and the hypothetical frame rule. As both rules are not used in our current system, we do not suffer from this problem and can safely support less precise heap states and multiple specifications. Visibility modifiers are taken into consideration in Leavens and Muller (2007) where a set of rules for information hiding in specifications for Java-like languages is given. Moreover, the authors demonstrate their application on the specification language JML. However, some JML tools, including ESC/Java2 (Flanagan et al. 2002; Cok and Kiniry 2004) ignore visibility modifiers in specifications.

The emergence of separation logic (Isthiaq and O'Hearn 2001; Reynolds 2002) provides a novel way to handle the challenging aliasing issues for heap-manipulating programs. Parkinson and Bierman (2005) and Parkinson (2005) recently extended separation logic to handle OO programs. They advocated the use of abstract predicate families indexed by types to reason about objects from a class hierarchy. Their approach supports full object views and essentially dynamic specifications, though for one class at a time. They also require every inherited method to be re-specified and mostly re-verified for OO conformance. Furthermore, no implementation exists. In comparison, we have designed a more comprehensive system with static specifications, partial views and modular mechanisms to minimise on re-verification and to handle super method calls. These issues were informally referred to as untamed open problems in Sec 6.5 of Parkinson (2005).

8. Conclusion

We have presented an enhanced approach to OO verification based on the co-existence of both static and dynamic specifications, together with a principle that each static specification be a subtype of its corresponding dynamic specification. Our approach attempts to track the actual type of each object, where possible, to allow static specifications to be preferably used. We have built our work on the formalism of separation logic, and have designed a new object format that allows each object to assume the form of its superclass via lossless casting. Another useful feature of our proposal is a new specification subsumption relation for pre/post specifications that is novel in using the residual heap state from precondition checking to assist in postcondition checking.

We have constructed a prototype system for verifying OO programs. Our prototype is built using Objective CAML augmented with an automatic Presburger solver, called Omega (Pugh 1992). While Presburger arithmetic is limited to integer constraints, we have also provided hooks in our system to invoke Isabelle and MONA provers. These extensions allow us to support sets/bags/lists constraints, where required. The main objective for building this prototype is to show the feasibility of our approach to enhanced OO verification based on a synergistic combination of static and dynamic specifications. As an initial study, we have successfully verified a set of small benchmark programs. The verification process consists of two parts: verification of the given static specifications against the bodies of the corresponding methods (VS) and the specification subtyping checking meant to avoid re-verification of all dynamic specifications and some static specifications of statically-inherited methods (SSC). What we are mainly interested in is the ratio between the VS timing and the SSC timing. As subsumption checking on specification is typically cheaper than verifying a piece of program code against its specification, we expect that VS will dominate the total verification time. This assumption is indeed validated by the examples we tried. For instance, in the counter example presented in the paper, the time taken by the SSC (with 16 checks) is 0.06 seconds, while VS (with 11 verifies) takes 0.18 seconds. For examples with larger code bases, we expect the ratio between the VS and SSC to increase.

One fundamental question that may arise is whether static specifications are really necessary? Some readers may contend that it is possible to incorporate the effect of static specification by adding *type*(this)=c into the precondition of a dynamic pre/post annotation. As discussed in our paper, this approach is only a partial solution to static specification as (i) it does not cater to statically-inherited methods which support reuse of static specifications, (ii) it does not handle super method calls which are really static method invocations, and (iii) it does not help enforce class invariants in subclasses when dynamic specifications of a superclass (without the class invariant property) are being inherited. By making each static specification be a subtype of its dynamic specification, we can limit code verification to only static specifications. The underlying philosophy of static specification is better served by *partial view* and *lossless casting*. Perhaps, the ultimate goal for OO verification is to use *completely* static specifications – with dynamic specifications derived on demand! Our solution can be viewed as a significant step towards this utopia.

Post-Submission Note : Independent of our work, Parkinson and Bierman (2008) proposed in the same proceedings a similar distinction and relation between static and dynamic specifications, in support of modular verification and the handling of direct method calls. Like us, they can support inheritance and overriding, while avoiding unnecessary re-verifications. However, there are at least two differences. Firstly, they continue to rely on the rather powerful concept of abstract predicate families, while we have avoided its use in our work. Secondly, we have a marginal emphasis on static specifications over dynamic specifications, as we advocate for the latter to be derived from the former, when needed, using the refinement techniques of specification specialization and abstraction.

Acknowledgements : Cristian Gherghina implemented a prototype system for OO verification. Florin Craciun, Hugh Anderson, Martin Rinard, Peter O'Hearn and anonymous reviewers provided insightful feedbacks on various aspects of this work. Matthew Parkinson and Gavin Bierman conducted helpful last minute discussions with us. This work is supported by an A*STAR-funded research project R-252-000-233-305 on "A Constructive Framework for Dependable Software". Shengchao Qin is supported in part by the EPSRC project EP/E021948/1.

References

P. America. Designing an object-oriented programming language with behavioural subtyping. In *the REX School/Worshop on Foundations of Object-Oriented Languages*, pages 60–90, 1991.

M Barnet, R. DeLine, M. Fahndrich, K.R.M Leino, and W. Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, 2004.

D. R. Cok and J. Kiniry. ESC/Java2: Uniting ESC/Java and JML. In *Int'l Workshop on Construction and Analysis of Safe, Secure, and Interoperable Smart Devices*, pages 108–128, 2004.

K. K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *IEEE/ACM Intl. Conf. on Software Engineering*, pages 258–267, 1996.

J. C. Filliâtre. Why: a multi-language multi-prover verification tool. Technical Report 1366, LRI, Université Paris Sud, March 2003.

R. B. Findler and M. Felleisen. Contract soundness for object-oriented languages. In *SIGPLAN Object-Oriented Programming Systems, Languages and Applications*, pages 1–15, 2001.

R. B. Findler, M. Latendresse, and M. Felleisen. Behavioral contracts and behavioral subtyping. In *ESEC/SIGSOFT Foundations of Software Engr.*, 2001.

C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended Static Checking for Java. In *ACM PLDI*, June 2002.

J. Hatcliff, X. Deng, M. B. Dwyer, G. Jung, and V. P. Ranganath. Cadena: An integrated development, analysis, and verification environment for component-based systems. In *IEEE/ACM Intl. Conf. on Software Engineering*, 2003.

S. Isthiaq and P.W. O'Hearn. BI as an assertion language for mutable data structures. In *ACM POPL*, London, January 2001.

J. Kiniry, E. Poll, and D. Cok. Design by contract and automatic verification for Java with JML and ESC/Java2. ETAPS tutorial, 2005.

G. T. Leavens and Peter Muller. Information hiding and visibility in interface specifications. In *IEEE/ACM Intl. Conf. on Software Engineering*, pages 385–395, Washington, DC, USA, 2007. IEEE Computer Society.

G. T. Leavens and David A. Naumann. Behavioral subtyping is equivalent to modular reasoning for object-oriented programs. Technical Report 06-36, Department of Computer Science, Iowa State University, 2006.

G. T. Leavens, A.L. Baker, and C. Ruby. Preliminary design of JML: A behavioral interface specification language for Java. *ACM SIGSOFT Software Engineering Notes*, 31(3):1–38, 2006.

G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Mller, and J. Kiniry. JML Reference Manual (DRAFT), February 2007.

K. R. M. Leino and P. Müller. Object invariants in dynamic contexts. In *ECOOP*, pages 491–516, 2004.

B. H. Liskov. Data abstraction and hierarchy. *ACM SIGPLAN Notices*, 23 (5):17–34, May 1988. Revised version of the keynote address given at OOPSLA'87.

B. H. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Trans. on Programming Languages and Systems*, 16(6):1811–1841, 1994.

C. Marché and C. Paulin-Mohring. Reasoning about Java programs with aliasing and frame conditions. In *18th Int'l Conf. on Theorem Proving in Higher Order Logics*. Springer, LNCS, August 2005.

C. Marché, C. Paulin-Mohring, and X. Urbain. The KRAKATOA tool for certification of JAVA/JAVACARD programs annotated in JML. *Journal of Logic and Algebraic Programming*, 58(1–2):89–106, 2004.

B. Meyer. *Object-oriented Software Construction*. Prentice Hall. Second Edition., 1997.

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

P. Muller. *Modular specification and verification of object-oriented programs*. Springer, New York, NY, USA, 2002. ISBN 3-540-43167-5.

H. H. Nguyen, C. David, S.C. Qin, and W.N. Chin. Automated Verification of Shape And Size Properties via Separation Logic. In *Intl Conf. on Verification, Model Checking and Abstract Interpretation*, Nice, France, January 2007.

J. W. Nimmer and M. D. Ernst. Invariant inference for static checking. In *ESEC/SIGSOFT Foundations of Software Engr.*, pages 11–20, 2002.

P. W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comput. Sci.*, 375(1-3):271–307, 2007.

P. W. O'Hearn, H. Yang, and J. C. Reynolds. Separation and Information Hiding. In *ACM POPL*, Venice, Italy, January 2004.

J. Ostroff, C. Wang, E. Kerfoot, and F. A. Torshizi. Automated model-based verification of object-oriented code. Technical Report CS-2006-05, York University, Canada, May 2006.

M. J. Parkinson. *Local Reasoning for Java*. PhD thesis, Computer Laboratory, University of Cambridge, 2005. UCAM-CL-TR-654.

M. J. Parkinson and G. M. Bierman. Separation logic and abstraction. In *ACM POPL*, pages 247–258, 2005.

M. J. Parkinson and G. M. Bierman. Separation logic, abstraction and inheritance. In *ACM POPL*, 2008.

C. Pasareanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In *SPIN Workshop*, April 2004.

W. Pugh. The Omega Test: A fast practical integer programming algorithm for dependence analysis. *Communications of the ACM*, 8:102–114, 1992.

J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE Logic in Computer Science*, Copenhagen, Denmark, July 2002.

Robby, M. B. Dwyer, and J. Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *ESEC/SIGSOFT Foundations of Software Engr.*, pages 267–276, 2003.