Implementing a Modular OO Verifier

Cristian Gherghina[1] Cristina David[2] Huu Hai Nguyen[3] Wei-Ngan Chin[2,3]

[1] Department of Computer Science, Politehnica University of Bucharest
[2] Department of Computer Science, National University of Singapore
[3] Computer Science Programme, Singapore-MIT Alliance
{cristian,davidcri,nguyenh2,chinwn}@comp.nus.edu.sg

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. Therefore, we advocate for a fresh approach to OO verification which focuses on multiple specifications that cater to both calls with static and dynamic dispatching. We introduce a novel specification subsumption mechanism that can help avoid re-verification, where possible. Our aim is to lay the foundation for a practical verification system that is precise, concise and modular for sequential OO programs. We shall exploit the separation logic formalism to achieve this.

1. Introduction

Separation logic [8, 4] extends Hoare logic to support reasoning about shared mutable data structures. It adds two more connectives to classical logic: separating conjunction $*$, and separating implication $-\!\!*$. We only make use of the former connective, $h_1 * h_2$, which asserts that two heaps described by h_1 and h_2 are domain-disjoint.

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. The format we use is $\mathtt{y}::\mathtt{c}\langle \mathtt{t}, \mathtt{v}^* \rangle$ denoting a variable y that points to an object belonging to c-class whose actual type is t and its fields are $\mathtt{v}^*$. With this notation, for a class List with fields val and next, we represent a variable x that points to a list containing the values 1 and 2 as $\mathtt{x}::\mathtt{List}\langle \mathtt{v}_1, \mathtt{p}\rangle * \mathtt{p}::\mathtt{List}\langle \mathtt{v}_2, \mathtt{null}\rangle \wedge \mathtt{v}_1{=}1 \wedge \mathtt{v}_2{=}2$.

2. Lossless Casting

An issue we address is performing upcast/downcast operations statically in accordance with OO class hierarchy, and without loss of information where possible. Our object format allows lossless casting to be performed with the help of the Ext predicate that captures the fields added by a subclass. For instance, let us consider the following class hierarchy:

```
class A {int n;}
class B extends A {int m;}
```

When an object x of type B is first created, we may capture its state using the formula $\mathtt{x}::\mathtt{B}\langle \mathtt{t}, 1, 2, \mathtt{p}\rangle \wedge \mathtt{t}{=}\mathtt{B} \wedge \mathtt{p}{=}\mathtt{null}$. This formula indicates that the actual type of the object is t=B and that there is no record extension since p=null. With this object format, we can now perform an upcast to its parent A class by transforming it to: $\mathtt{x}::\mathtt{A}\langle \mathtt{t}, 1, \mathtt{q}\rangle * \mathtt{q}::\mathtt{Ext}\langle \mathtt{B}, 2, \mathtt{p}\rangle \wedge \mathtt{t}{=}\mathtt{B} \wedge \mathtt{p}{=}\mathtt{null}$. Though this cast operation is viewing the object as a member of A class, it is still a B object as the type information t=B indicates. Furthermore, we have created an extension record $\mathtt{q}::\mathtt{Ext}\langle \mathtt{B}, 2, \mathtt{p}\rangle$ that can capture the extra field of the B subclass.

There are also occasions when we are required to pass the full object with all its (extended) fields. This occurs for specifications of methods where subsequent overridings may change the extra fields of its subclass. To cater to this scenario, we introduce an $\mathtt{ExtAll}\langle \mathtt{t}_1, \mathtt{t}_2\rangle$ predicate that can capture all the extension records from a class $\mathtt{t}_1$ for an object with actual type $\mathtt{t}_2$. The ExtAll predicate can be defined as follows, where the notation $\mathtt{t}_3{<}\mathtt{t}_1$ denotes a class $\mathtt{t}_3$ and its immediate superclass $\mathtt{t}_1$:

$$\begin{array}{l}\mathtt{ExtAll}\langle \mathtt{t}_1, \mathtt{t}_2\rangle \equiv \mathtt{t}_1{=}\mathtt{t}_2 \wedge \mathtt{self}{=}\mathtt{null} \vee \mathtt{self}::\mathtt{Ext}\langle \mathtt{t}_3, \mathtt{v}^*, \mathtt{q}\rangle \\ \qquad * \mathtt{q}::\mathtt{ExtAll}\langle \mathtt{t}_3, \mathtt{t}_2\rangle \wedge \mathtt{t}_3{<}\mathtt{t}_1 \wedge \mathtt{t}_2{<:}\mathtt{t}_3 \ \mathbf{inv}\ \mathtt{t}_2{<:}\mathtt{t}_1\end{array}$$

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

3. Static and Dynamic Specifications

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. Most analysis techniques uphold Liskov's Substitutivity Principle [6] 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 [3, 2, 5] 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. A basic specification subsumption mechanism was originally formulated as follows.

DEFINITION 3.1 (**Specificaton Subsumption).** *Consider a method* A.mn *in class* A *with* $(\mathtt{pre}_\mathtt{A} \twoheadrightarrow \mathtt{post}_\mathtt{A})$ *as its pre/post specification, and its overriding method* B.mn *in subclass* B*, with a given pre/post specification* $(\mathtt{pre}_\mathtt{B} \twoheadrightarrow \mathtt{post}_\mathtt{B})$*. The specification* $(\mathtt{pre}_\mathtt{B} \twoheadrightarrow \mathtt{post}_\mathtt{B})$ *is said to be a subtype of* $(\mathtt{pre}_\mathtt{A} \twoheadrightarrow \mathtt{post}_\mathtt{A})$ *in support of method overriding, if the following subsumption relation holds:*

$$\frac{\mathtt{pre}_\mathtt{A} \wedge \mathit{type}(\mathtt{this}){<:}\mathtt{B} \implies \mathtt{pre}_\mathtt{B} \qquad \mathtt{post}_\mathtt{B} \implies \mathtt{post}_\mathtt{A}}{(\mathtt{pre}_\mathtt{B} \twoheadrightarrow \mathtt{post}_\mathtt{B}) <:_\mathtt{B} (\mathtt{pre}_\mathtt{A} \twoheadrightarrow \mathtt{post}_\mathtt{A})}$$

Consequently, the drawback that might arise is that specifications are typically imprecise (or weaker) for methods of superclasses. To improve precision of OO verification, we advocate for multiple specifications to be allowed for each method and we introduce two categories of specifications, known as static and dynamic specifications.

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.

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.

Each dynamic specification must satisfy the following two subsumption properties:

- Be a specification supertype of its static counterpart. This helps keep code re-verification to a minimum.
- Be a specification supertype of the dynamic specification of each overriding method in its sub-classes. This helps ensure behavioral subtyping.

Our technique is important as the majority of method dispatch operations (71%) are indeed statically known [1].

To illustrate our approach towards OO verification, let us consider the following example, highlighting the use of partial views for static specifications and of full views for the dynamic ones:

```
class A {
  int n, m;
  void set_n_to_1()
  static requires this::A⟨_, _⟩ ensures this::A⟨1, 2⟩;
  dynamic requires this::A⟨_, _⟩$ ensures this::A⟨1, 2⟩$
        ∨this::A⟨1, 3⟩$;
    {this.n = 1; this.m = 2; }}
class B extends A{
  void set_n_to_1()
  static requires this::B⟨_, _⟩ ensures this::B⟨1, 3⟩;
  dynamic requires this::B⟨_, _⟩$ ensures this::B⟨1, 3⟩$;
    {this.n = 1; this.m = 3; }
  A uses_dynamic(A v)
  static requires v::A⟨_, _⟩ ensures res::A⟨1, 2⟩∨res::A⟨1, 3⟩;
    {v.set_n_to_1(); return v; }}
  A uses_static()
  static requires true ensures res::A⟨1, 2⟩;
    {A v = new A(0, 0); v.set_n_to_1(); return v; }}
```

The dynamic specification of method set_n_to_1 in class A is weakened to ensure behavioral subtyping, becoming less precise than its static counterpart. As in method uses_static we know the exact type for the receiver v of the call v.set_n_to_1 to be A, we can use the static specification given to the method set_n_to_1 in class A. On the other hand, in the method uses_dynamic, due to dynamic dispatch, we are forced to use the more imprecise dynamic specification. Consequently, the postcondition that we are able to ensure for the uses_static method, res::A⟨1, 2⟩, is stronger than the one for uses_dynamic, res::A⟨1, 2⟩∨res::A⟨1, 3⟩.

4. Enhanced Specification Subsumption

We improve on the notion of specification subsumption given in Definition 3.1 by allowing postcondition checking to be strengthened with the residual heap state from precondition checking. This enhancement is courtesy of the frame rule from separation logic.

DEFINITION 4.1 (**Enhanced Spec. Subsumption).** *A pre/post annotation* preB ⋈→ postB *is said to be a subtype of another pre/post annotation* preA ⋈→ postA *if the following relation holds:*

$$\frac{\texttt{preA} \vdash \texttt{preB} * \Delta \qquad \texttt{postB} * \Delta \vdash \texttt{postA}}{(\texttt{preB} \bowtie\!\!\rightarrow \texttt{postB}) <: (\texttt{preA} \bowtie\!\!\rightarrow \texttt{postA})}$$

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.*

To illustrate the utility of the enhanced specification subsumption, let us consider the example given below for which we need to ensure that *static-spec*(A.foo) <: *dynamic-spec*(A.foo).

```
class A {int n;
  void foo()
  static requires this::A⟨_⟩ ensures this::A⟨1⟩
  dynamic requires this::A⟨_⟩$ ensures this::A⟨1⟩$
    {this.n = 1; }}
class B extends A{int m;
  void foo()
  static requires this::B⟨_, _⟩ ensures this::B⟨1, 1⟩
    {this.n = 1; this.m = 1; }}
```

For the required specification subsumption to hold, we must prove:

$$\texttt{this::A}\langle \texttt{t}, \_, \texttt{p}\rangle \bowtie\!\!\rightarrow \texttt{this::A}\langle \texttt{t}, 1, \texttt{p}\rangle <: \texttt{this::A}\langle \texttt{t}, \_, \texttt{q}\rangle * \texttt{q::ExtAll}\langle \texttt{A}, \texttt{t}\rangle \bowtie\!\!\rightarrow \texttt{this::A}\langle \texttt{t}, 1, \texttt{q}\rangle * \texttt{q::ExtAll}\langle \texttt{A}, \texttt{t}\rangle$$

The above subtyping only succeeds with our enhanced subsumption relation. We first show the contravariance of the preconditions:

$$\texttt{this::A}\langle \texttt{t}, \_, \texttt{q}\rangle * \texttt{q::ExtAll}\langle \texttt{A}, \texttt{t}\rangle \vdash \texttt{this::A}\langle \texttt{t}, \_, \texttt{p}\rangle * \Delta$$

This succeeds with $\Delta \equiv \texttt{p::ExtAll}\langle \texttt{A}, \texttt{t}\rangle$. We then prove covariance on the postconditions using:

$$\texttt{this::A}\langle \texttt{t}, 1, \texttt{p}\rangle * \Delta \vdash \texttt{this::A}\langle \texttt{t}, 1, \texttt{q}\rangle * \texttt{q::ExtAll}\langle \texttt{A}, \texttt{t}\rangle$$

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

5. Implementation

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 [7]. The main objective for building this prototype is to show the feasibility of our approach to enhanced OO verification based on a 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).

Class	VS (secs)	SSC (secs)
Example introduced in Section 3		
A	0.03	0.012
B	0.087	0.029
Example introduced in Section 4		
A	0.07	0.01
B	0.02	0.013
Counter example		
Cnt	0.05	0.016
FastCnt	0.044	0.012
PosCnt	0.065	0.048
TwoCnt	0.072	0.0226

Figure 1. Verification Times for Small Benchmarks

References

[1] D. F. Bacon and P. F. Sweeney. Fast static analysis of C++ virtual function calls. In *ACM OOPSLA*, pages 324–341, 1996.

[2] 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.

[3] K.K. Dhara and G. T. Leavens. Forcing behavioral subtyping through specification inheritance. In *IEEE / ACM SIGSOFT ICSE*, pages 258–267, 1996.

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

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

[6] 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.

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

[8] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE LICS*, Copenhagen, Denmark, July 2002.