Temporal Verification Theories for Java-like Classes

Suad Alagić, Mark Royer, and Dennis Crews

Department of Computer Science
University of Southern Maine
Portland, ME 04104-9300
e-mail: {alagic,mroyer,crews}@cs.usm.maine.edu

Abstract. We consider Java-like object types equipped with assertions as in recent proposals and implementations. The first issue that we consider is the formal notion of an object-oriented type extended with logic-based constraints along with the notions of inheritance and substitutability for such extended types. The second issue is a suitable logic for explicitly expressing properties of sequences of object states, particular cases of which appear for mutator methods in Java-related and other object-oriented assertion languages. The third issue is a suitable prover technology and the required techniques for verifying properties of object types extended with logic-based constraints. We present our solution for these problems based on the view of object types as temporal theories along with a model theory and the required verification techniques. The temporal logic-based approach makes it possible to reason about properties of sequences of object states which allows verification of behavioral subtyping requirements that are based on history properties.

1 Introduction

This paper investigates the formal basis and the verification techniques for the application of modern verification systems to object-oriented languages extended with logic-based constraints. The starting point is the fact that an object-oriented type signature when extended with logic-based constraints (method preconditions, method postconditions and class invariants in particular, as in Eiffel [27], JML [24, 20] or OCL [33]) may be viewed as a theory. This makes it possible to apply a tool such as PVS [28], which is based on the view of types as theories.

The types as theories view requires a suitably defined general notion of a class as a theory along with the notion of a class theory morphism expressing the semantic relationships among classes. The underlying model theory is then required to tie the inheritance of class constraints and behaviorally compatible object substitutability. Following [15, 14], a theory of behavioral compatibility presented in [2] abstracts over a variety of logics that have been proposed for expressing the behavioral (semantic) features of the object-oriented paradigm, and it applies to any well-defined notion of a model (interpretation) of a class

signature. The key component of this model theory is the *Satisfaction Condition* [15, 14] which ties precisely the inheritance of class constraints and behaviorally compatible object substitutability [2].

In addition to the PVS view of types as theories [28], the choice of PVS is also based on its sophisticated type system with predicate subtyping and bounded parametric polymorphism which makes it possible to represent parametric classes that appear in Java 5.0 when equipped with assertions. The types as theories view a presented in [2] is thus extended in this paper to parametric theories following [15]. Very general, and even higher-order logic capabilities of PVS allow specification of a suitable temporal logic theory which is the basis of our representation techniques for Java-like classes as temporal theories [4].

The first challenge in this paper is representation of classes as program verification system theories and representation of inheritance and subtyping as defined in object-oriented languages in terms of import of theories and subtyping as defined in PVS. Another major challenge is in representing methods that mutate the underlying object state within the functional, abstract data type based view of types in PVS. Behavioral subtyping as defined in [26] is not based on the notion of types as theories. Because of that a further challenge is in the development of a technique for verifying behavioral subtyping requirements within the framework of a program verification system based on theories. Finally, in order to appropriately represent the fact that the lifetime of an object is a sequence of object states, a more sophisticated temporal logic is required. This leads to temporal class theories defined in a program verification system (PVS in particular) and the development of techniques for verification of behavioral compatibility among classes. This in particular allows verification of behavioral subtyping requirements that are based on history properties [26].

2 Constraints for parametric classes

A sample parametric interface `Collection`, in the JML notation [24, 20], extended with assertions, is given below as an illustration. The difference in comparison with the current version of JML is in the usage of parametric polymorphism as in Java 5.0. To our knowledge, JML is currently being extended with features such as genericity. The method `count` is a pure function that returns the number of occurrences of its argument object in the underlying `Collection` object. The mutator methods `insert` and `delete` that change the underlying object state are equipped with method preconditions and method postconditions. The postconditions for the methods `insert` and `delete` are in fact specified in a temporal logic style because they refer to both the current and the previous object states. The latter are denoted using the keyword `old` as in Eiffel or JML, and OCL has an alternative notation. This temporal logic aspect of the assertion languages of JML and Eiffel is implicit in these assertion languages. In this paper we make it explicit and develop both the formal theory and verification techniques based on the view of classes as temporal theories.

```
public interface Collection<T> {
 /*@ pure @*/
 public int count(T o);

 /*@ ensures \result <==> this.count(o) > 0; pure  @*/
 public boolean belongs(T o);

 /*@ ensures this.count(o) == \old(this.count(o)) + 1 &&
     (\forall T o1; !o1.equals(o) ==>
                 (this.count(o1) == \old(this.count(o1)))); @*/
    public void insert(T o);

 /*@ requires this.belongs(o);
  ensures this.count(o) == \old(this.count(o)) - 1 &&
     (\forall T o1; !o1.equals(o) ==>
                 (this.count(o1) == \old(this.count(o1)))); @*/
   public void delete(T o);

 /*@ invariant (\forall T o; this.count(o) >= 0); @*/
}
```

An implementing class of the interface `Collection<T>` would naturally include a constructor with the postcondition `(\forall T o; this.count(o) == 0)`. The first order predicate calculus-based constraints appear in the sample class `Collection` given above. We show in section 4 how a temporal logic is used to express the object-state changes caused by mutator methods that require usage of the operator `old`.

PVS techniques in this paper are targeted toward verification of behavioral compatibility conditions among classes such as behavioral subtyping. As an illustration, a parametric class `Bag` equipped with assertions is constructed in such a way that `Bag` is a behavioral subtype of `Collection`. The preconditions and postconditions of the inherited mutator methods `insert` and `delete` remain the same as in the supertype `Collection`. The semantics of the additional `Bag` methods `union` and `intersect` are defined by the postconditions of these methods.

```
public interface Bag <T> extends Collection <T>{

 /*@ ensures (\forall T o;
     this.count(o) >= b.count(o) ==>
             this.union(b).count(o) == this.count(o) &&
     this.count(o) < b.count(o) ==>
             this.union(b).count(o) == b.count(o));
   pure @*/
 public Bag<T> union(Bag<T> b);

 /*@ ensures (\forall T o;
     this.count(o) < b.count(o) ==>
          this.intersect(b).count(o) == this.count(o) &&
```

```
     this.count(o) >= b.count(o) ==>
          this.intersect(b).count(o) == b.count(o));
    pure @*/
  public Bag<T> intersect(Bag<T> b);
}
```

An interplay of bounded parametric polymorphism and behavioral subtyping is illustrated below in the interface `OrderedCollection` whose type parameter is bounded by `Comparable`.

```
public interface OrderedCollection
                      <T extends Comparable> extends Collection<T> {
 /*@ ensures \result >= 0; pure @*/
 public int index(T x);

 /*@ invariant (\forall T x1,x2; this.belongs(x1) && this.belongs(x2) ==>
   (x1.compareTo(x2) <= 0 ==> this.index(x1) <= this.index(x2))); @*/
}
```

The condition that the actual type substituted for `T` is a subtype of `Comparable` will be checked by the type system. This will guarantee that the actual type parameter is equipped with comparison methods. But this provides no guarantee that these methods have the desired properties as specified by assertions in the interface `Comparable`. The required check is that the actual parameter substituted for `T` is a behavioral subtype of `Comparable`. The type system has no way of checking this behavioral requirement. JML might be able to check it when extended with bounded parametric polymorphism. This is an example of a condition that one would like to be verified by invoking a prover.

3 Class theories

In order to verify properties of a class equipped with assertions, the class must be transformed into a specification that can be handled by PVS. PVS specifications are theories. A theory of a class consists of the type signatures of its methods represented in the standard functional style along with a collection of logic-based constraints, which are sentences expressed in the chosen logic.

We will use the notion of a class signature to denote the collection of type signatures of methods of the class. Extending the notion of a class signature with constraints expressed as sentences of a particular logic leads to the notion of a class theory.

A class theory $Th_A = (\Sigma_A, E_A)$ consists of a class signature Σ_A and a finite set E_A of Σ_A sentences.

Given a theory (Σ, E), $Mod(\Sigma)$ will denote a collection of Σ models and $\models_\Sigma$ will denote the satisfaction relation between models and Σ sentences. A Σ model provides an interpretation of the signatures of functions given in Σ. The fact that a model M satisfies a sentence e is thus denoted as

$M \models_\Sigma e$.

Given a theory (Σ, E), $closure_\Sigma(E)$ denotes the set of Σ sentences containing E defined as

$e \in closure_\Sigma(E)$ iff $M \models_\Sigma e$ for all Σ models M such that $M \models_\Sigma E$.

Theory specifications in this paper are typically not closed. This is why they are often called theory presentations.

Given a theory (Σ, E), E^* will denote a collection of Σ models M defined as $E^* = \{M \mid M \models_\Sigma E\}$.

The paradigm underlying PVS is naturally functional. But mutator methods, i.e., methods that change the underlying object state are not functions. In our approach, a mutator method is represented as a binary predicate that defines pairs of object states such that the second state may be obtained from the first by invoking the mutator. This is our PVS representation (based on [31]) of constraints that requires usage of the operator `old` in Eiffel or JML to express properties of pairs of object states. The object state prior to the invocation of a mutator m is characterized by a precondition predicate pre_m and the state after invocation of a mutator m is specified by a predicate $post_m$. While mutators are predicates that are required to hold for pairs of object states, preconditions and postconditions are predicates required to hold in particular object states.

So a class theory (Σ_A, E_A) will be equipped with the following predicates for each mutator m of A:
$pre_m : A \rightarrow bool$, $post_m : A \rightarrow bool$, and
$mutator_m : A, A \rightarrow bool$,
i.e., $pre_m \in \Sigma_A$, $post_m \in \Sigma_A$, and $mutator_m \in \Sigma_A$.

In addition, the set of sentences E_A will be extended with the following sentence for each mutator m of A expressing the requirement for correctness of a mutator:

$\forall(o_1, o_2 : A)(pre_m(o1) \wedge mutator_m(o1, o2) \Rightarrow post_m(o2)) \in E_A$

However, there is more to the above specification since o_1 and o_2 correspond to object states before and after execution of a mutator m. Explicit representation of this fact in this paper will be based on a suitable temporal logic introduced in section 4.

If $Th_A = (\Sigma_A, E_A)$ and $Th_B = (\Sigma_B, E_B)$ are class theories, $\phi : Th_A \rightarrow Th_B$ is a theory morphism iff

$\phi : \Sigma_A \rightarrow \Sigma_B$ is a type signature morphism such that $e \in E_A$ implies $\phi(e) \in closure(E_B)$.

A type signature morphism is a function $\phi : \Sigma_A \rightarrow \Sigma_B$. This function induces a mapping of terms and hence a mapping of Σ_A sentences into Σ_B sentences. If e is a Σ_A sentence then $\phi(e)$ denotes its corresponding Σ_B sentence under the mapping ϕ.

Following [15], $C[T :: B]$ will denote a theory with a type parameter T whose bound is B. Such a theory is viewed as a theory morphism

$\eta_B : B \rightarrow C[T :: B]$.

This morphism embeds the theory of the bound B into the parametric theory C. The diagram 1. is required to be a pushout [15]. This property will be

explained in section 5 as it applies to inheritance of parametric classes. In PVS this is expressed by an import as

```
C[(IMPORTING B) T: TYPE FROM B]
```

Instantiation of C with the actual parameter D is represented by the following commutative diagram of theory morphisms:

$$\begin{array}{ccc} B & \xrightarrow{\eta_B} & C[T :: B] \\ f\downarrow & & \downarrow c_{<D/T>} \\ D & \xrightarrow{\eta_D} & C[D] \end{array}$$

Diagram 1: Instantiation of a parametric type

In the above diagram, the existence of a theory morphism $f : B \rightarrow D$ is required, since it guarantees the semantic compatibility of the actual parameter D with respect to the bound B. In PVS this is expressed by an import clause and PVS subtyping:

```
D: THEORY
BEGIN IMPORTING B
      D: TYPE FROM B
% body of D
END D
```

$c_{<D/T>} : C[T :: B] \rightarrow C[D]$ is the substitution morphism induced by $T \rightarrow D$. The morphism $\eta_D : D \rightarrow C[D]$ embeds the theory of the actual parameter D into the instantiated parametric theory $C[D]$. In PVS this import of D into $C[D]$ is naturally implied.

4 Temporal verification theories

The main advantage of temporal logics for the object-oriented paradigm is that they provide explicit support for the view that the lifetime of an object is in fact a sequence of object states. In addition, the rules for behavioral subtyping as defined in [26] are based on properties of sequences of object states (history properties) and temporal logics offer a paradigm for reasoning about sequences of states.

In the temporal logic underlying PVS theories developed in this paper a sequence of object states is a function

$seq : TIME \rightarrow state$

where $TIME$ is just the set of natural numbers.
The notation for a set of functions with the domain A and codomain B is $[A \rightarrow B]$. A temporal predicate $TemporalPred$ is now a function:

$TemporalPred : [TIME \rightarrow state] \times TIME \rightarrow bool.$

So given a state sequence $seq : TIME \rightarrow state$, and a time instant t, $TemporalPred$ evaluates to a boolean value for the state $seq(t)$.

Standard temporal operators of the logic in this paper are:

ALWAYS, NEXT, UNTIL, and *EVENTUALLY*.

These operators act on temporal predicates so that the type of *ALWAYS, NEXT*, and *EVENTUALLY* is $[TemporalPred \rightarrow TemporalPred]$. The type of the temporal operator *UNTIL* is

$[TemporalPred \times TemporalPred \rightarrow TemporalPred]$.

In the temporal theory given below the temporal operators are specified along with the standard boolean operator `AND`. All the remaining standard boolean operators (`OR`, `IMPLIES` and `NOT`) are also necessarily defined for temporal predicates. The boolean constants are also specified as temporal using overloading available in PVS. `TYPE+` denotes an uninterpreted nonempty type. Usage of upper case identifiers is only for readability purposes and has no significance in PVS.

```
temporal [state: TYPE+]: THEORY
BEGIN  TIME: TYPE = nat
       stateSequence: TYPE = [TIME -> state]
       TemporalPred: TYPE = [stateSequence, TIME -> bool]
 ss: VAR stateSequence
 p,q: VAR TemporalPred
 t,j,k: VAR TIME

ALWAYS: [TemporalPred -> TemporalPred] = (LAMBDA p:
               (LAMBDA ss, j: FORALL t: t >= j IMPLIES p(ss,t)));

NEXT:   [TemporalPred -> TemporalPred] =  (LAMBDA p:
                                     (LAMBDA ss, t: p(ss, t+1)));

EVENTUALLY: [TemporalPred -> TemporalPred] = (LAMBDA p:
                    (LAMBDA ss,j: EXISTS k: k >= j AND p(ss,k)));

UNTIL: [TemporalPred, TemporalPred -> TemporalPred] =
     (LAMBDA p,q: (LAMBDA ss,j: EXISTS k: k >= j AND q(ss,k) AND
         FORALL t: j <= t AND t < k IMPLIES p(ss,t)));

AND:    [TemporalPred,TemporalPred -> TemporalPred] =
               (LAMBDA (p,q): (LAMBDA ss,t: p(ss,t) AND q(ss,t)));
% definition of OR, NOT, and IMPLIES

isValid(p): bool = FORALL ss: p(ss, 0)
predicateToTemporal(p: PRED[state]):
                            TemporalPred = (LAMBDA ss, t: p(ss(t)))
CONVERSION predicateToTemporal
END temporal
```

Since the predicate evaluation function evaluates a predicate with respect to a sequence of object state, the semantics of ordinary predicates must be defined in the temporal paradigm. This is done in the above theory by a function `predicateToTemporal` which takes an ordinary predicate on the object state and defines its semantics when this predicate is viewed as a temporal one. For a predicate `p` on the object state, its corresponding temporal predicate `predicateToTemporal(p)` will evaluate to true for a sequence of object

states `seq` and a time instant `t` iff `p` evaluates to true in the state `seq(t)`. The `CONVERSION` statement in the above theory will automatically perform a conversion of an ordinary predicate into a temporal one whenever required.

In the `mutators` theory given below, the effect of mutator methods is expressed in the above defined temporal paradigm.

```
mutators[state: TYPE+]: THEORY
BEGIN   IMPORTING temporal[state]
        MUTATOR: TYPE = [state,state -> bool]
 m: VAR MUTATOR
 s: VAR state
 pre,post,p,q: VAR pred[state]
 seq: VAR stateSequence
 t,t1,t2:   VAR TIME

 correctMutator(pre,m,post): TemporalPred =
      (LAMBDA seq,t: (FORALL t1,t2:(t <= t1 AND t1 < t2) IMPLIES
     (pre(seq(t1)) AND m(seq(t1),seq(t2)) IMPLIES post(seq(t2))) ))

 mutatorConstraint(pre, m, post, seq,t1,t2): bool =
  t1 < t2 AND  pre(seq(t1)) AND m(seq(t1),seq(t2)) AND post(seq(t2))
% mutator composition etc.
END mutators
```

The type parameter `state` stands for the type of the receiver object. The `mutators` theory is based on the `temporal` theory so that it allows modeling of object state changes along the time axis. The predicate `correctMutator` is a temporal specification of the usual Hoare-style correctness. The predicate `mutatorConstraint` will be used in restricting the sequences of object states to those that are created by application of mutators of a particular theory as in the `Collection` example given below. The limitation of the above theory is that it does not specify that object identity remains invariant along the time axis.

The `Collection` theory imports the mutators theory with the actual type parameter `Collection` standing for the state type. Methods `insert` and `delete` are defined as mutator predicates. Two temporal theorems `insertTheorem` and `deleteTheorem` are defined using the temporal operator `ALWAYS` to express properties of sequences of `Collection` object states. These two theorems are successfully proved by PVS.

```
Collection[T: TYPE+]: THEORY
BEGIN
      Collection: TYPE+
      IMPORTING mutators[Collection]
 c,c1,c2: VAR Collection
 o,o1: VAR T
 t,t1,t2: VAR TIME
 seq: VAR stateSequence

 % Definitions
 count(c,o): int
```

```
 belongs(c,o): bool =  count(c,o) > 0

 % Mutators
 insert(o): MUTATOR = (LAMBDA (c1,c2): count(c2,o) = count(c1,o) + 1 AND
                 FORALL (o1| o1 /= o): count(c1,o1) = count(c2,o1))
 preInsert(o):  pred[Collection] =  (LAMBDA c: count(c,o) >= 0)
 postInsert(o): pred[Collection] = (LAMBDA c: belongs(c,o))

 delete(o): MUTATOR = (LAMBDA (c1,c2): count(c2,o) = count(c1,o) - 1
           AND FORALL (o1|o1 /= o): count(c1,o1) = count(c2,o1))
 preDelete(o):  pred[Collection] =  (LAMBDA c: belongs(c,o))
 postDelete(o): pred[Collection] = (LAMBDA c: TRUE)

% Mutator Correctness theorems
 insertTheorem: THEOREM isValid(ALWAYS(
                correctMutator(preInsert(o),insert(o),postInsert(o))))
 deleteTheorem: THEOREM isValid(ALWAYS(
                correctMutator(preDelete(o),delete(o),postDelete(o))))

 % Invariant and initial state
 initialState: AXIOM (FORALL (o, seq): count(seq(0), o) >= 0)
 naturalCount: TemporalPred = (LAMBDA seq, t:
                          (FORALL (o): count(seq(t),o) >= 0))

 nextState(seq,t1,t2): bool = (FORALL (o):
   mutatorConstraint(preInsert(o), insert(o),postInsert(o),seq,t1,t2) OR
   mutatorConstraint(preDelete(o), delete(o),postDelete(o),seq,t1,t2) )

 naturalCountStates: TemporalPred = (LAMBDA seq, t1:
               (FORALL t2: naturalCount(seq,t1) AND nextState(seq,t1,t2)
                                          IMPLIES naturalCount(seq,t2)))
 countTheorem: THEOREM isValid(ALWAYS(naturalCountStates))
END Collection
```

The fact that the theory `Collection` is parametric is reflected in the signatures of method preconditions, method postconditions and mutator predicates. For a method m we have the following representation:

$pre_m : T \rightarrow [Collection[T] \rightarrow bool]$,
$post_m : T \rightarrow [Collection[T] \rightarrow bool]$, and
$mutator_m : T \rightarrow [Collection[T] \times Collection[T] \rightarrow bool]$.

A predicate $nextState : [TIME \rightarrow state] \times TIME \times TIME \rightarrow bool$ specifies valid pairs of object states as those that are created by invocation of the mutators `insert` and `delete`. The initial state axiom is the assumption about constructors of objects requiring that initial object states satisfy the invariant. Given the above, the reasoning behind the proof includes the assumption that all state transitions satisfy the `nextState` predicate. This assumption is justified because `insertTheorem` and `deleteTheorem` were proved to hold. The proof

then amounts to verifying that if the invariant holds in a particular object state, it will hold in a subsequent object state, where the two states are specified by the `nextState` predicate. Since the `initialState` axiom guarantees that the invariant holds in the initial object state, these conditions produce the desired result: `isValid(ALWAYS(naturalCountStates))`.

5 Inheritance and substitutability

The relationship between a class A and its subclass B is represented in the corresponding class theories as a pair of functions

- A type signature morphism $\phi : \Sigma_A \rightarrow \Sigma_B$
- The abstraction function $Mod(\phi) : Mod(\Sigma_B) \rightarrow Mod(\Sigma_A)$

The signature morphism ϕ is represented by importing the theory A into the theory B so that Σ_A is a subsignature of Σ_B. The abstraction function $Mod(\phi)$ maps a model for Σ_B into a model for Σ_A. The PVS representation technique has the following form:

```
A: THEORY
BEGIN A: TYPE
% body of theory A
END A
```

```
B: THEORY
BEGIN  IMPORTING A
       B: TYPE FROM A
 % body of theory B
END B
```

In PVS the subtype declaration `B: TYPE FROM A` is equivalent to

```
B_pred: [A -> bool]
B: TYPE =(B_pred)
```

where `(B_pred)` denotes a type that satisfies `B_pred`. This is the PVS notion of predicate subtyping. So the clause `IMPORTING A` corresponds to the signature morphism $import_B^A : \Sigma_A \rightarrow \Sigma_B$ and the clause `B: TYPE FROM A` corresponds to the abstraction map

$Mod(import_B^A) : Mod(\Sigma_B) \rightarrow Mod(\Sigma_A)$.

If Σ is a type signature, then $Sen(\Sigma)$ denotes the set of Σ sentences of a particular logic. Sentences in $Sen(\Sigma)$ are constructed starting with the terms that are based on function signatures from Σ and applying the rules of a particular logic. $Sen(import_B^A) : Sen(\Sigma_A) \rightarrow Sen(\Sigma_B)$ will denote the inclusion of sentences over the subsignature Σ_A of Σ_B into the set of sentences over Σ_B. The relationships described above are represented in the diagram below:

$$\begin{array}{ccc} Mod(\Sigma_A) & \models_{\Sigma_A} & Sen(\Sigma_A) \\ Mod(import_B^A)\uparrow & & \downarrow Sen(import_B^A) \\ Mod(\Sigma_B) & \models_{\Sigma_B} & Sen(\Sigma_B) \end{array}$$

Diagram 2: Inheritance and substitutability

The above diagram illustrates duality of inheritance, expressed by the arrow $Sen(import_B^A)$, and substitutability, expressed by the arrow $Mod(import_B^A)$, where the two have opposite directions [2, 14].

The `IMPORT` clause in fact includes the whole theory of A into the theory of B so that all sentences of A will be available in B. This situation is a particular case of the notion of a theory morphism. So the desired effect of the import of the theory A into the theory of B is

$Mod(import_B^A) : E_B^* \to E_A^*$.

The PVS notion of predicate subtyping has the following implication on modeling inheritance of methods. A method `m` of `A` with the signature

`m: [A,C2,...,A,...,Cm -> A]`

will be available in `B` with exactly the same signature, just like in the Java invariant subtyping rule for signatures of inherited methods. However, since `B` is a PVS subtype of `A`, the effect would be as if `m` is available in `B` with the signature `m: [B,C2,...,B,...,Cm -> A]`. Otherwise, overriding the signature of `m` in `B` to a signature such as

`m: [B,C2,...,B,...,Cm -> B]`

which has covariant change of the result type as in Java 5.0 requires definition of a new function `m` in `B`.

The fact that a theory $K[T :: B]$ is representing a subclass K of a parametric class C is represented by a theory morphism:

$C[T :: B] \to K[T :: B]$

expressed by an import in the PVS notation

```
K [(IMPORTING B) T: TYPE FROM B]
BEGIN
      IMPORTING C[T]
      K: TYPE FROM C[T]
% body of K
END K
```

According to the above we have a theory morphism $\phi : C[T :: B] \to K[T :: B]$. Given a theory morphism $f : B \to D$, instantiation of the parametric theory $K[T :: B]$ produces a theory morphism $k_{<D/T>} : K[T :: B] \to K[D]$ according to diagram 1. By composition,

$\phi : C[T :: B] \to K[T :: B]$ and $k_{<D/T>} : K[T :: B] \to K[D]$
produce a theory morphism $k_{<D/T>}\phi : C[T :: B] \to K[D]$

so that we have the following commutative diagram 3 of theory morphisms:

$$\begin{array}{ccc} B & \xrightarrow{\eta_B} & C[T :: B] \\ f\downarrow & & \downarrow k_{<D/T>}\phi \\ D & \xrightarrow{\eta_D} & K[D] \end{array}$$

Diagram 3: Inheritance and instantiated types

The above diagram 3 along with diagram 1 produces a unique theory morphism $C[D] \rightarrow K[D]$. This is the pushout property [15]. This morphism guarantees compatibility of $K[D]$ with $C[D]$.

A theory `OrderedCollection` illustrates representation of bounded parametric polymorphism in PVS. A theory `Comparable` is equipped with a partial ordering `<=`. The type parameter of the theory `OrderedCollection` has a bound `Comparable`, hence a valid actual type for `T` must be a PVS subtype of `Comparable`. `OrderedCollection` is also defined as a PVS subtype of `Collection`.

```
OrderedCollection [(IMPORTING Comparable)
                          T: TYPE+ FROM Comparable]: THEORY
BEGIN   IMPORTING Collection[T]
        OrderedCollection: TYPE+ FROM Collection[T]
        IMPORTING mutators[OrderedCollection]
  seq: VAR  stateSequence[OrderedCollection]
  t:   VAR  TIME[OrderedCollection]
  i1,i2: VAR T

  index: [OrderedCollection, T -> nat ]
  isOrdered: temporal[OrderedCollection].TemporalPred =
             (LAMBDA seq, t: FORALL(i1,i2):
       belongs(seq(t),i1) AND belongs(seq(t),i2) IMPLIES
           (compareTo(i1,i2) <= 0 IMPLIES
                index(seq(t),i1) <= index(seq(t),i2)))
END OrderedCollection
```

6 Verifying behavioral subtyping

The notion of behavioral subtyping as defined in [26] is based on assertions (method preconditions, method postconditions and type invariants). The effect of the behavioral subtyping rules for mutators of a class A and its subclass B may be expressed by the following sentence:

$$\forall(o_1, o_2 : B)(pre_m^A(o1) \wedge mutator_m^B(o1, o2) \Rightarrow post_m^A(o2))$$

where o_1 and o_2 correspond to object states prior to and after invocation of a mutator m. So if the precondition of the mutator m in the class A is satisfied, and the mutator m as redefined in the class B is executed, the postcondition of the mutator m as specified in A will be satisfied. This will indeed be the case if the rules of behavioral subtyping:

$\forall(o : B)(pre_m^A(o) \Rightarrow pre_m^B(o))$,
$\forall(o : B)(post_m^B(o) \Rightarrow post_m^A(o))$,

are satisfied for the mutator m along with the condition for correctness of the mutator m in B:

$\forall(o_1, o_2 : B)(pre_m^B(o1) \wedge mutator_m^B(o1, o2) \Rightarrow post_m^B(o2))$.

The PVS notion of subtyping has its limitations with respect to inheritance and subtyping in object-oriented languages. The above sentences are written in such a way that they reflect the PVS view of predicate subtyping. Class theories consist of method signatures and the associated sentences. In fact, typical object types considered in this paper such as `Collection` and `Bag` do not come with the specification of object state in Java itself. Hence, all of the above applies directly to these Java types.

A theory of a subclass extends the theory of the superclass by additional method signatures and additional sentences. Our main goal is verification of behavioral compatibility (behavioral subtyping) of a subclass with respect to the superclass. Behavioral compatibility is expressed in terms of sentences that involve method invocation only, hence the PVS notion of subtyping is adequate for this purpose. It would not be adequate if we were trying to reason explicitly about components of object states. The object state of a subtype is an extension of the object state of the supertype. The abstraction (typically projection) function $Mod(import)$ maps the state of a subtype object to the state of the corresponding supertype object. The abstraction function would then have to appear in the above sentences.

The construction of a class theory is performed in such a way that it extends the superclass theory. In other words, the superclass theory is a subtheory [15, 2] of the subclass theory. The behavioral subtyping conditions are verified by constructing a suitable PVS theory as described below. A `Bag` theory is specified by importing `Collection` theory and defining `Bag` as a PVS subtype of `Collection`. The theory `Bag` introduces functions `union` and `intersect` that apply to bags. These two functions satisfy the properties specified in this theory by making use of the predicates `postUnion` and `postIntersect` defined in this theory. The theorems `insertTheorem` and `deleteTheorem` have the same form as in `Collection` theory.

```
Bag[T: TYPE+] : THEORY
BEGIN IMPORTING Collection[T]
      Bag: TYPE+ FROM Collection[T]
      IMPORTING mutators[Bag]
  b,b1,b2: VAR Bag
  o: VAR T
  t,t1,t2: VAR TIME[Bag]
  seq: VAR stateSequence[Bag]

 union:     [Bag,Bag -> Bag]
 intersect: [Bag,Bag -> Bag]

 postUnion(b1,b2): pred[Bag] = (LAMBDA (b): (FORALL (o):
    (count(b1,o) >= count(b2,o) IMPLIES count(b,o) = count(b1,o)) AND
    (count(b1,o) < count(b2,o) IMPLIES  count(b,o) = count(b2,o)) ))
```

```
 postIntersect(b1,b2): pred[Bag] = (LAMBDA (b): (FORALL(o):
    (count(b1,o) >= count(b2,o) IMPLIES count(b,o) = count(b2,o)) AND
    (count(b1,o) < count(b2,o)  IMPLIES  count(b,o) = count(b1,o)) ))

% union and intersect theorems via postUnion and postIntersect

  preInsert(o): pred[Bag] = (LAMBDA (b): Collection.preInsert(o)(b))
  postInsert(o): pred[Bag] = (LAMBDA (b): Collection.postInsert(o)(b))
  insert(o): MUTATOR[Bag] = (LAMBDA b1,b2: Collection.insert(o)(b1,b2))

  preDelete(o): pred[Bag] = (LAMBDA (b): Collection.preDelete(o)(b))
  postDelete(o): pred[Bag] = (LAMBDA (b): Collection.postDelete(o)(b))
  delete(o): MUTATOR[Bag] = (LAMBDA b1,b2: Collection.delete(o)(b1,b2))

% spec. of initial state axiom and nextState

  insertTheorem: THEOREM temporal[Bag].isValid(
      temporal[Bag].ALWAYS(correctMutator(preInsert(o),insert(o),
                                                    postInsert(o))))
  deleteTheorem:  THEOREM temporal[Bag].isValid(
      temporal[Bag].ALWAYS(correctMutator(preDelete(o),delete(o),
                                                    postDelete(o))))
  nextStateTheorem: THEOREM (FORALL seq,t1,t2:
   Bag.nextState(seq,t1,t2) IMPLIES Collection.nextState(seq,t1,t2))
% Count theorem
END Bag
```

PVS verifies the theorems in the above specification. `Bag` is defined in such a way that the abstract type `Bag` is a subtype of the abstract type `Collection` equipped with additional properties expressed as theorems. This means that the `Bag` theory is intended to be a consistent extension of the `Collection` theory. The `nextStateTheorem` illustrates a behavioral subtyping requirement related to history properties [26]. This theorem states that sequences of object states of the subtype are in fact valid sequences of states of the supertype.

In order to verify the behavioral subtyping conditions a suitable PVS theory called `BagBehavior` is constructed.

```
BagBehavior[T: TYPE+]: THEORY
BEGIN  IMPORTING Bag[T]

 insertBehavior: THEOREM FORALL(o:T, B: Bag):
 (Collection.preInsert(o)(B) IMPLIES Bag.preInsert(o)(B))
  AND Bag.postInsert(o)(B) IMPLIES Collection.postInsert(o)(B)))

 deleteBehavior: THEOREM (FORALL (o: T, B: Bag):
    (Collection.preDelete(o)(B) IMPLIES Bag.preDelete(o)(B))
  AND (Bag.postDelete(o)(B) IMPLIES Collection.postDelete(o)(B)))

 insertHistory: THEOREM (FORALL (o: T, B1,B2: Bag)
   (Bag.insert(o)(B1,B2) IMPLIES Collection.insert(o)(B1,B2))
```

```
 deleteHistory: THEOREM (FORALL (o: T, B1,B2: Bag)
   (Bag.delete(o)(B1,B2) IMPLIES Collection.delete(o)(B1,B2))

 insertMutatorTheorem: THEOREM
   temporal[Collection].isValid(temporal[Collection].ALWAYS(
           correctMutator(Collection.preInsert(o),
           Bag.insert(o), Collection.postInsert(o))))

  deleteMutatorTheorem: THEOREM temporal[Collection].isValid(
       temporal[Collection].ALWAYS(correctMutator(Collection.preDelete(o),
         Bag.delete(o), Collection.postDelete(o))))
END BagBehavior
```

An attempt to verify the theorems of this theory by PVS succeeds. Of course, this is obvious in this example because of the way it was constructed. The `insertMutatorTheorem` and `deleteMutatorTheorem` are implications of the previously proved theorems and correspond precisely to the notion of behavioral subtyping: if the precondition for `insert` in `Collection` holds for the receiver `Bag` object and the `Bag` `insert` is executed, the postcondition for `insert` in `Collection` will hold for the receiver `Bag` object. Likewise for `delete`. Note that all the theorems of the `Collection` theory: `insertTheorem`, `deleteTheorem` and `countTheorem` have already been proved to hold for `Bag` theory. This is in accordance to the view of behavioral subtyping [26, 2] which requires that all theorems of the supertype must hold for its subtypes. This allows behaviorally compatible substitution of a `Bag` object for a `Collection` object.

Consider now a PVS theory `Set`, given below, constructed in such a way that the abstract type `Set` is a subtype of the abstract type `Collection`.

```
Set[T: TYPE+] : THEORY
BEGIN  IMPORTING Collection[T]
       Set: TYPE+ FROM Collection[T]
       IMPORTING temporal[Set]
 s,s1,s2: VAR Set
 o, o1: VAR T
 n: VAR nat
 seq: VAR temporal[Set].stateSequence
 t: VAR temporal[Set].TIME
 member(s,o): bool =count(s,o)=1

 preInsert(o):  pred[Set] = (LAMBDA (s): NOT member(s,o))
 postInsert(o): pred[Set] = (LAMBDA (s): member(s,o))
 insert(o): MUTATOR[Set] = (LAMBDA s1,s2: member(s2,o) AND
            (FORALL (o1 | o1 /= o): member(s1,o1) IFF member(s2,o1)))

 preDelete(o):  pred[Set] =  (LAMBDA(s): member(s,o))
 postDelete(o): pred[Set] = (LAMBDA (s): NOT member(s, o))

 delete(o): MUTATOR[Set] = (LAMBDA s1,s2: NOT member(s2,o) AND
```

```
            (FORALL (o1 | o1 /= o): member(s1,o1) IFF member(s2,o1)))

 union(s1,s2): pred[Set] = (LAMBDA (s): (FORALL (o): member(s,o)
                                  IFF (member(s1,o) OR member(s2,o))))
 intersect(s1,s2): pred[Set] = (LAMBDA (s):(FORALL (o):  member(s,o)
                                  IFF member(s1,o) AND member(s2,o)))
% insertTheorem, deleteTheorem, nextStateTheorem
END Set
```

An attempt to prove that `Set` is a behavioral subtype of `Collection` using the methodology described above would fail because `Set` naturally strengthens the `insert` precondition. The `nextStateTheorem` relating the history properties of `Set` and `Collection` will also fail and so will `insertMutatorTheorem` and `deleteMutatorTheorem` in the `SetBehavior` theory.

7 The constraint rule

The rules for behavioral subtyping defined in [26] include particularly strong requirements for compatibility of history properties. History properties are properties of sequences of object states. A pair of states where the second succeeds the first is a particular case. The notion of a constraint as defined in [26] is thus naturally expressed as a temporal predicate:

$Constraint : [TIME \rightarrow state] \times TIME \rightarrow bool$

For a particular theory A the constraint is proved based on the assumption that state transitions are restricted by the *nextState* predicate. The *initialState* axiom is also needed to establish:

$ALWAYS(Constraint[A])$

Since an invariant is just a predicate on a single object state, by the predicate to temporal conversion `predicateToTemporal` specified in the PVS `temporal` theory, an invariant may be viewed as a constraint. However, a constraint may be much more general, and it typically expresses properties of a sequence of two or more object states as in [26].

Consider now a theory B which imports the theory A and defines B as a PVS subtype of A. This theory import will be denoted as $import_B^A$. The corresponding abstraction function $Mod(import_B^A)$ acts on the state B to produce a state A:

$Mod(import_B^A)_{state} : state_B \rightarrow state_A$.

An implication of the above relationship between the theories A and B is that a state sequence of B produces a state sequence of A. Indeed, $stateSequence[B] : TIME \rightarrow state_B$ and $Mod(import_B^A)_{state}$ produce $stateSequence[A] : TIME \rightarrow state_A$ by composition so that we have

$stateSequence[A](t) = Mod(import_B^A)_{state}(stateSequence[B](t))$.

In order to establish that the theory B specifies a behavioral subtype of the theory A we assume that their respective constraints are proved in each theory separately:

$ALWAYS(Constraint[A])$ and

$ALWAYS(Constraint[B])$.

In addition we prove the following property:

$nextState[B] \Rightarrow nextState[A]$.

We take the assumptions about initial object states as axioms, but in fact the idea is that the following should also hold:

$initialState[B] \Rightarrow initialState[A]$.

The implication is:

$ALWAYS(Constraint[B] \Rightarrow Constraint[A])$.

In other words, if the constraint for B holds then under the above assumptions, the constraint for A will hold as well under the same assumptions as illustrated by the following diagram:

$$\begin{array}{ccc} state_B \times TIME & \overset{Constraint[B]}{\longrightarrow} & true \\ Mod(import_B^A)_{state} \times 1_{TIME}\downarrow & & \downarrow = \\ state_A \times TIME & \overset{Constraint[A]}{\longrightarrow} & true \end{array}$$

where 1 denotes the identity function. Following the above reasoning, the provable properties of the relationship between the theories *Collection* and *Bag* as indicated in the `Bag` theory are:

$initialState[Bag] \Rightarrow initialState[Collection]$

$nextState[Bag] \Rightarrow nextState[Collection]$

leading to the desired result

$ALWAYS(Constraint[Bag] \Rightarrow Constraint[Collection])$.

The corresponding theorem for the `Set` and `Collection` theories fails.

8 Related research

A number of projects are addressing the problem of extending object-oriented languages, Java in particular, with assertions [5]. ESC/Java [13] statically detects some programming errors. Nice [11] is a functional Java like language with assertions that are enforced at run-time, and Spec# [8] is a superset of C# equipped with assertions. JML [24, 25] annotates Java programs with behavioral specifications that are compiled and enforced at run-time, and LOOP [10] generates theories (PVS in particular) representing the semantics of Java classes so that they can be verified by a theorem prover. A related work is [18].

The rules for behavioral subtyping as specified in [26] were accompanied with a hint that the problem of behavioral subtyping should be viewed in terms of subtheories. The related work includes extensive research reported in [21, 22, 24] where the results [30, 23] are in fact of model theoretic nature. The notion of specification as used in [1] corresponds to the notion of a theory.

Integrating classes and algebraic specifications has been studied by [12]. Our view of classes as theories is based on [15]. The model theory is based on [15] and [14]. Its applications to module-based component software systems are given in [16]. The implications of this model theory in the object-oriented paradigm were investigated in [2] developing a model theory of behavioral compatibility for an object-oriented paradigm based on self-typing. In a coalgebraic view of

the problem of behavioral compatibility [32] the notion of behavioral compatibility and the results on soundness and completeness are closely related to the Satisfaction Condition in [14, 15, 2]. The same remark applies to the notion of behaviorally correct subtyping [23].

In comparison with our earlier work reported in [2], the main contribution of this paper is in matching this theory of behavioral compatibility with the tools and techniques based on the PVS system [28]. Both this theory and PVS are attractive for the object-oriented paradigm because they are general enough to apply to a variety of logics [15, 14, 2].

Classes as temporal theories were studied in [4, 6] with an implementation of a declarative object-oriented temporal language on top of the Java Virtual Machine [5]. The temporal PVS view presented in this paper builds upon the work reported in [31] which considers sequences of states and temporal operators using the apparatus available in PVS. Our earlier paper applies PVS technology to the problem of verification of object-oriented database transactions [3].

PVS is a programmable verification system. It has a small language (with a type system and even formal semantics [19]) for specifying proof strategies. This makes it possible to customize the proof checking capabilities of PVS to the specific problem of verification of behavioral subtyping requirements. The strategy language and a variety of already available proof strategies are given in [29, 7].

9 Conclusions

In this paper we addressed the following problems in the formal foundations for the applications of modern program verification systems to Java-like parametric classes equipped with logic-based constraints:

- Representing classes equipped with assertions as theories of a modern program verification system.
- Matching an object-oriented type system with a type system of a modern program verification system equipped with subtyping and parametric theories.
- Expressing the effects of methods that change the underlying object state (mutators) within the abstract data type, functional program verification system paradigm.
- A model-theoretic approach to the problem of behavioral compatibility of a subclass with respect to its superclass, which makes it possible to apply modern program verification tools to verify the behavioral compatibility requirements.
- A technique to fit the notion of behavioral subtyping to the types as theories view, and using program verification tools based on that view to verify behavioral subtyping requirements.
- Developing a temporal logic view of the object-oriented paradigm based on suitable program verification system theories, and using temporal theories to represent classes and verify or disprove behavioral properties such as behavioral compatibility.

The current representation does not address the issues related to object identity. This extension is left for future work. It would require specification of an `Object` theory that would include functions for identity comparisons like the default method `equals` in Java, and the associated constraints. This would produce a more complex but more realistic object-oriented modeling framework capable of object aliasing and expressing properties implemented by the heap model.

Our experiences show that using PVS comes with nontrivial subtleties. One of them is that, contrary to our natural expectations, PVS does not check the soundness of a set of axioms. The other subtlety is that even if the proofs succeed, one has to be very cautious about what PVS actually proved. That requires looking into the actual proofs which are by no means easy to read. On the positive side, using PVS requires explicit specification of the assumptions that make the proofs go through. This makes both specifications and proofs much more carefully defined. This paper reports the results attained thus far, with a number of issues still to be resolved in future work. Perhaps the most important area that is still under development is the proof strategies suited for classes as temporal verification theories.

References

1. M. Abadi and K. R. M. Leino, A logic of object-oriented programs, Proceedings of TAPSOFT '97, *Lecture Notes in Computer Science 1214*, pp. 682-696, Springer, 1997.
2. S. Alagić, S. Kouznetsova, Behavioral compatibility of self-typed theories. Proceedings of ECOOP 2002, *Lecture Notes in Computer Science 2374*, pp. 585-608, Springer, 2002.
3. S. Alagić and J. Logan, Consistency of Java transactions, Proceedings of DBPL 2003, *Lecture Notes in Computer Science 2921*, pp. 71-89, Springer, 2004.
4. S. Alagić, Semantics of temporal classes, *Information and Computation*, *163* pp. 60-102, 2000.
5. S. Alagić, J. Solorzano, and D. Gitchell, Orthogonal to the Java imperative, Proceedings of ECOOP '98, *Lecture Notes in Computer Science 1445*, pp. 212 - 233, Springer, 1998.
6. S. Alagić and M. Alagić, Order-sorted model theory for temporal executable specifications, *Theoretical Computer Science 179*, pp. 273-299, 1997.
7. M. Archer, B. Di Vito, and C. Munoz, Developing user strategies in PVS: A tutorial, Proceedings of STRATA 2003.
8. M. Barnett, K. R. M. Leino, and W. Schulte, The Spec# programming system: an overview, Microsoft Research 2004. Also in Proceedings of CASSIS 2004.
9. V. Benzanken and X. Schaefer, Static integrity constraint management in object-oriented database programming languages via predicate transformers, Proceedings of ECOOP '97, *Lecture Notes in Computer Science 1241*, pp. 60-84, 1997.
10. J. van den Berg and B. Jacobs, The LOOP compiler for Java, *Lecture Notes in Computer Science 2031*, Springer, 2001, pp. 299 - 312.
11. D. Bonniot, The Nice programming language, http://nice.sourceforge.net/.
12. R. Breu, *Algebraic specifications in Object-Oriented Programming Environments*, *Lecture Notes in Computer Science 562*, Springer, 1991.

13. C. Flanagan, K. R. M. Leino, G. Nelson, J. B. Saxes, and R. Stata, Extended static checking for Java, Proceedings of PLDI, ACM, 2002, pp. 234-245.
14. J. Goguen and R. Burstall, Institutions: Abstract model theory for specification and programming, *Journal of the ACM*, *39*, pp. 92-146, 1992.
15. J. Goguen, Types as theories, in: G. M. Reed, A. W. Roscoe and R. F. Wachter, *Topology and Category Theory in Computer Science*, pp. 357-390, Clarendon Press, Oxford, 1991.
16. J. Goguen and W. Trace, An implementation oriented semantics for module composition, in: G. Leavens and M. Sitaraman, *Foundations of Component-Based Systems*, pp. 231 - 263, Cambridge University Press, 2000.
17. B. Jacobs, Objects and classes, co-algebraically, in B. Freitag, C.B. Jones, C. Lengauer, and H.-J. Schek, eds., *Object Orientation with Parallelism and Persistence*, pp. 83–103, Kluwer, 1996.
18. B. Jacobs, L. van den Berg, M. Husiman and M. van Berkum, Reasoning about Java classes, Proceedings of OOPSLA '98, pp. 329-340, ACM, 1998.
19. F. Kirchner, Coq tacticals and PVS strategies: A small step semantics, Proceedings of STRATA 2003.
20. G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cook, P. Muller, and J. Kiniry, JML Reference Manual (draft), July 2005, http://www.cs.iastate.edu/ leavens/JML/.
21. G. Leavens and D. Pigozzi, The behavior-realization adjunction and generalized homomorphic relations, *Theoretical Computer Science 177*, pp. 183-216, 1997.
22. G. T. Leavens and K. K. Dhara, Concepts of behavioral subtyping and a sketch of their extension to component-based systems, in: G. T. Leavens and M. Sitaraman, *Foundations of Component-Based Systems*, Cambridge University Press, 2000.
23. G. Leavens and D. Pigozzi, Equational reasoning with subtypes, TR #02-07, Department of Computer Science, Iowa State University, 2002.
24. G. Leavens and Y. Cheon, Design by contract with JML, Iowa State University, 2004.
25. G. T. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. R. Cok, How the design of JML accommodates both runtime assertion checking and formal verification, *Science of Computer Programming*, Vol. 55, pp. 185-205, Elsevier, 2005.
26. B. Liskov and J. M. Wing, A behavioral notion of subtyping, *ACM Transactions on Programming Languages and Systems*, *16*, pp. 1811-1841, 1994.
27. B. Meyer, *Object-Oriented Software Construction*, Prentice Hall, 1997.
28. S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Clavert: PVS Language Reference, SRI International, Computer Science Laboratory, Menlo Park, California.
29. S. Owre and N. Shankar, Writing PVS proof strategies, Computer Science Laboratory, SRI International, http://www.csl.sri.com.
30. D. Pigozzi and G. Leavens, A complete algebraic characterization of behavioral subtyping, *Acta Informatica 36*, pp. 617-663, 2000.
31. A. Pnueli and T. Arons, TLPVS: A PVS-based LTL verification system, In: Verification: theory and Practice, *Lecture Notes In Computer Science Vol 2772*, Springer, 2004.
32. E. Poll, A coalgebraic semantics of subtyping, *Theoretical Informatics and Applications* Vol. 35(1), pp. 61-82, 2001.
33. J. Warmer and A. Kleppe, *The Object Constraint Language: Getting your Models Ready for MDA*, Addison-Wesley, 2003.