A Formal Bridge between Runtime Assertion Checking and Static Verification of Inheritance

ABSTRACT

In object-oriented programming languages, the relationship that should hold between the specifications of a class and its superclass is called behavioural subtyping. In this paper we analyse the conditions that a behavioural subtype should meet during runtime assertion checking, that is, during dynamic verification. Our exploration relates such conditions for runtime assertion checking to the conditions that should be met in static verification of correctness under the principles of modular reasoning. As a result, we state and prove a theorem that connects dynamic and static verification of method calls in the presence of inheritance. The novelty of this theorem lies on the fact that the connection is an equivalence, where the implication from static to dynamic verification has been explored before but not the opposite one. This new exploration then poses that a hypothetical exhaustive testing through runtime assertion checking would be equivalent to the corresponding static verification of definite correctness, which adds solidity to runtime assertion checking. None but one of the runtime assertion checking tools that we know of can effectively detect all possible problems in class inheritance; the one exception is the tool used for Spec#, but their strategy relies on both specification inheritance and a rather substantial restriction on preconditions, requirements we could dispose of when taking our the-oretical results to a practical implementation.

■ Gabriela Montoya and Jesús N. Ravelo
e-mail: gmontoya,jravelo@ldc.usb.ve
Departamento de Computación y Tecnología de la Información
Universidad Simón Bolívar
Caracas, Venezuela

Fecha de Recepción: 31 de Marzo de 2011
Fecha de Aceptación: 03 de Junio de 2011

Key words. inheritance, data refinement, data abstraction, dynamic verification, method calls.

Un puente formal entre el chequeo de aserciones en tiempo de ejecución y verificación estática de la herencia

RESUMEN

En lenguajes de programación orientados por objetos, la correcta relación que debe ser satisfecha entre las especificaciones de una clase y su superclase es llamada de "subtipo por comportamiento" (del término en inglés "*behavioural sub-typing*"). En este artículo se analizan las condiciones que un subtipo por comportamiento debe cumplir durante el proceso de verificación de aserciones durante ejecución (del inglés "*runtime assertion checking* "), conocido también como verificación dinámica a secas. Nuestra exploración relaciona tales condiciones de verificación dinámica a las condiciones que deben ser satisfechas durante el proceso de verificación estática de correctitud bajo principios de razonamiento modular. Como resultado, se presenta y se demuestra un teorema que conecta la verificación dinámica y la estática de llamadas a métodos en presencia de herencia. Lo novedoso de este teorema es que la conexión planteada es una equivalencia, de la cual ha sido explorada con anterioridad la implicación desde la verificación estática hacia la dinámica pero no la implicación contraria. Esta nueva exploración plantea entonces que un proceso hipotético de prueba ("testing") exhaustiva a través de verificacion dinámica de aserciones sería equivalente a la correspondiente verificación estática de correctitud definitiva. Con una sola excepción, ninguna de las herramientas de verificación dinámica de aserciones conocidas por los autores es capaz de detectar todos los posibles problemas que pueden presentarse con la herencia entre clases; la excepción corresponde a la herramienta usada para el lenguaje Spec#, pero la estrategia de esta se apoya tanto en el uso de herencia de especificaciones como en una considerable restricción que debe ser impuesta a las precondiciones, requerimientos ambos que pueden ser eliminados al llevar los resultados teóricos del presente trabajo a una implementación práctica.

Palabras Claves: herencia, refinamiento de datos, abstracción de datos, verificación dinámica, llamadas a métodos.

1. INTRODUCTION

In this article, we analyse how the correctness of object-oriented method calls in the presence of inheritance should be dealt with through runtime assertion checking. More importantly, we explore this in connection with the corresponding static verification of such calls. We thus build a formal bridge linking dynamic and static verification through a theorem that states an equivalence between them. One direction of this bridge, the fact that static correctness through modular reasoning guarantees that runtime assertion checking of dynamically-bound method calls will succeed, has already been explored and proved [1, 2]. The other direction of the bridge is, to the best of our knowledge, a novel result. It amounts to stating that a hypothetical exhaustive testing through runtime assertion checking would be equivalent to the corresponding static verification, which gives a more solid foundation to such a dynamic verification. Without the guarantee provided by the novel implication of our theorem, one might suspect that testing via runtime assertion checking could succeed indefinitely without ever detecting present flaws. This is the added value which we argue our theorem provides to runtime assertion checking: if there is an error, that is, static correctness does not hold, testing can "eventually" detect it. Therefore, modulo the wellknown shortcomings of testing, particularly the fact that exhaustive testing is usually unattainable, runtime assertion checking backed by our result is more dependable.

We present this exploration using a small simplified object-oriented language extended with behavioural specifications such as class invariants and method pre/postconditions, in the spirit of Eiffel [3], Java Modeling Language (JML) [4,5] and Spec# [6]. With the reassurance of the aforementioned theorem, we put forward detailed conditions to be verified through runtime assertion checking. Such conditions can be simplified in the presence of specification inheritance [7], as used in JML and Spec#. Finally, as the practical counterpart of our theoretical exploration, we also present Java [8] code to do dynamic verification of method calls, both using the fully detailed conditions and their simplified version, in an extended version of this paper [9]. We also show that all but one of the existing runtime assertion checking tools do not capture appropriately all the verification required by inheritance and behavioural subtyping. The one exception is the runtime assertion checker of Spec#, but it relies on the fact that specification inheritance guarantees behavioural subtyping and also relies on imposing a substantial restriction on preconditions; an implementation based on our results would require neither of these conditions, although we could optionally use the first.

Inheritance or specialisation is a hierarchical relationship between classes under which subclasses offer the same methods as their superclasses, possibly refining the behaviour of some of them and possibly offering additional methods. It is imperative that all subclasses preserve the behaviour of the superclass, and thus be able to fulfill what is expected of the latter. This is known as the *substitution principle* [10], and subclasses that meet this property are known as *behavioural subtypes*. This notion was first presented by Liskov and Wing in [11] and various other authors, some of them working over different formal foundations, have insisted on the importance of this behavioural property. Among these authors, we have Back, Mikhajlova, von Wright, Mikhajlov and Sekerinski, who use the refinement calculus to formalise the notion of a subtype being a refinement of its supertype. They thus ensure that the substitution principle is met [12–14]. Also, Leavens and Dhara [7] have put forward a notion of weak behavioural subtypes, which impose fewer restrictions than those given by Liskov and Wing, but its use is limited to programs in which there is no aliasing between variables of different types. Leavens and Dhara also introduce strong behavioural subtypes as a modified version of the notion of behavioural subtyping initially proposed by Liskov and Wing. In both cases, weak and strong, their subtypes meet the substitution principle.

We will use the functional language Haskell [15, 16] to present the structure of the simplified object-oriented language of our formalisation. Also, we will use Haskell as the vehicle to formalise the conditions to be used in runtime assertion checking, and the theorem that connects this dynamic checking to the corresponding static verification conditions. We decided to use Haskell mainly for two reasons: (i) it is a language that facilitates the expression of all this information in a clear and precise way through its type system and function definition mechanisms; and (ii) formalisations of theorems expressed in Haskell have been proven both manually and semi-automatically in simple successful ways [17].

The rest of this article is organised as follows. Section 2 presents our simplified object-oriented language extended with behavioural specifications of classes and methods. Also, several auxiliary functions over the structure of the language are defined for later use. Section 3 then presents, first, the rules that must be met by behavioural subtypes, with which we then proceed to present the conditions to be used during runtime assertion checking for method calls in the presence of inheritance. Such dynamic conditions are deduced from the corresponding conditions for static verification. It is then in Section 4 where we put forward the above mentioned theorem that relates the conditions for runtime assertion checking with the conditions for static verification of correctness. Section 5 reviews related work, specially existing runtime assertion checking tools for object-oriented languages, analysing these tools under the light of our exploration. Section 6 closes with some conclusions and possible ways to extend the work presented in this paper.

2. A SIMPLIFIED EXTENDED OBJECT-ORIENTED LANGUAGE

We consider the following extension of a simplified object-oriented language: every class has a name, possibly a superclass, an invariant, and a list of methods; each method has a name, a precondition, a postcondition, and its body. We only consider methods without parameters and that are not functional, that is, that do not return results. This allows us to simplify the presentation without losing generality, as these restrictions will not affect the validity of the results we will present.

```
data Class = Class Id (Maybe Class)
                        Invariant Method]
data Method = Meth Id Precondition
                Postcondition  Instruction
type Invariant = BoolExpr
type Precondition = BoolExpr
type Postcondition = BoolExpr
```

Fig. 1. The structure of the extended object-oriented language.

To represent the structures that make up our simplified object-oriented language, we use the type definitions written in Haskell shown in Fig. 1. A class has a name of type `Id`, possibly a superclass, an invariant and a list of methods. A method has a name of type `Id`, a precondition, a postcondition, and an instruction that corresponds to its body. Type `Id` is the restriction of the `String` type to valid names. All three types `Invariant`, `Precondition` and `Postcondition` are just synonyms of the type BoolExpr. The type `BoolExpr` corresponds to boolean expressions, which may contain variables, constants, operators, et cetera; we will have more to say about this type later. The type `Instruction` should include representations for typical instructions of an imperative object-oriented language; however, we will not make use of internal structural details of this type.

2.1 Extraction Functions and Others

This subsection presents Haskell functions that extract relevant information from the structures of the simplified extended object-oriented language just defined.

```
super :: Class -> Maybe Class
super (Class _ sc _ _) = sc
inv :: Class -> BoolExpr inv
(Class _ _ i _) = i
methods :: Class -> [Method]
methods (Class _ _ _ ms) = ms
annotPre :: Method -> BoolExpr
annotPre (Meth _ p _ _) = p
annotPost :: Method -> BoolExpr
annotPost (Meth _ _ q _) = q
```

Fig. 2 Basic extraction functions.

Figure 2 shows the five most basic extraction functions. The first three functions simply extract the main components of a class, respectively, its potential superclass, its invariant and its list of methods. Since function `super` only extracts the potential superclass of a class, of type `Maybe Class` , if one is confident that some class c has an actual superclass and wants to extract it, Haskell function fromJust can be used to get it with the expression `fromJust (super c)` , of type Class. The other two functions in Fig. 2, in turn, extract information from methods: the precondition and the postcondition, respectively, with which a method is locally annotated in its declaration.

```
localMethod  :: Id -> Class -> Maybe Method
localMethod nm c
    = find (\(Meth n _ _ _) -> n == nm)
                (methods c)
method :: Id -> Class -> Method method nm c
    | isNothing local = method nm (super c)
    | isJust local = fromJust local
    where
             local = localMethod nm c
declare :: Class -> Method -> Bool
declare c (Meth n _ _ _)
                                            = isJust
(localMethod n c)
pre :: Method -> Class -> BoolExpr
pre (Meth nm _ _ _) c = annotPre (method
nm c)
post :: Method -> Class -> BoolExpr
post (Meth nm _ _ _) c = annotPost (method
nm c)
```

Fig. 3 Other extraction functions.

As we have to deal with inherited, not redefined, methods, we will have to define functions that deal with extracting a method from a class, whether it is directly declared in the class or it is inherited. These functions are presented in Fig. 3. Function `localMethod` extracts he local method with a given name from a given class, if it exists, which is why its return type is `Maybe Method`. Function method is its counterpart both for locally declared `methods` and inherited ones. Note that function `method` requires that the soughtafter method will eventually be found along the inheritance chain. The third function, `declare`, determines whether a class locally declares a method with a certain name. Armed with the first three functions of Fig. 3, the other two functions can deal with the general case of pre/postconditions with which any method can be offered by a class. A method is offered by a class if it is locally declared in the class or if it is inherited from its superclass. Functions `pre` and `post` obtain he precondition and the postcondition, respectively, that a `method` has in a certain class. Note that function `method` deals with the analysis of whether the annotations are local or inherited.

There is one last function to be presented in this section, but its nature is different to the nature of the previous ones: it is actually an "expression constructor" instead of an "evaluating function". The discussion on this function will also bring up other issues related to the type `BoolExpr`, which we use to represent boolean expressions.

```
old :: BoolExpr -> BoolExpr
```

Fig. 4 A BoolExpr constructor.

Function old in Fig. 4 is meant to construct an expression that denotes the "initial value" of its expression argument, that is, the value of its expression argument *before* the execution of a method, but used in a postcondition, the value of which is meant to be analysed *after* the execution of the method at issue.

We do not provide a definition body for this function, as it must be considered as a unary constructor of the type `BoolExpr`.

Likewise, `BoolExpr` should have constructors that correspond to all the typical boolean operators: binary constructors for conjunction, disjunction and so on; a unary constructor for negation; et cetera.

To avoid an unnecessary proliferation of names, we will use regular boolean operators as constructors for our type `BoolExpr`. Actually, the only other constructors we need for `BoolExpr` in what follows are conjunction, implication, equivalence and the constant true: we will use Haskell notation `&&` for the first, ==> for the second, and Haskell notation == and `True` for the third and the fourth. Our subtle use of the same syntax for both boolean operators and `BoolExpr` constructors thus corresponds to the following: for Haskell boolean ex pressions `a` and `b`, expression `a && b` evaluates to the conjunction of them; for expressions `c` and `d` of type `BoolExpr`, expression `c && d` is used to construct a new expression of type `BoolExpr` that denotes the conjunction of them. The same goes for ==> and == as binary constructors for `BoolExpr`, for True as a constant (0-ary) constructor for `BoolExpr`, and also for old as a unary constructor for `BoolExpr`.

All the functions presented in this section will be used in the rest of the paper to formalise rules, conditions, lemmas and theorems.

3. RUNTIME ASSERTION CHECKING

In this section, we explore what conditions should be runtime assertion checked around a method call in the presence of inheritance. Given that inheritance should correspond to behavioural subtyping, as mentioned in the introduction, we first re call in Subsect. 3.1 the conditions that behavioural subtypes must meet. Then, in Subsect. 3.2, modular reasoning is used to analyse the correctness conditions of a method call in the presence of inheritance, that is, the corresponding *static* verification conditions, and deduce from them the conditions that should be used during runtime assertion checking, that is, the *dynamic* verification conditions for the method call. Finally, subsection 3.3 presents stronger conditions to be used for runtime assertion checking that result from combining the conditions deduced in Subsect. 3.2 with the rules for behavioural subtyping of Subsect. 3.1.

3.1 Rules of Behavioural Subtyping

For a class to be a behavioural subtype of another class, it must satisfy a num ber of conditions that we enumerate below. In this subsection, we follow Dijkstra and Scholten [18] in the use of square brackets [...] to state that a logical formula is a theorem, and we start using the functions defined in Sect. 2.

Let `c0` and `c1` be two classes. According to Leavens and Naumann [1], `c0` is a behavioural subtype of `c1` if the following conditions hold:

(i). Invariants rule:

```
[ inv c0 ==> inv c1 ] .
```

(ii). Preconditions rule: for every method m defined in both types,

```
[ pre m c1 ==> pre m c0 ] .
```

(iii). Postconditions rule: for every method m defined in both types,

```
[ post m c0 && old (pre m c1)
==> post m c1 ] .
```

Rules (ii) and (iii) correspond to standard method refinement [19, Laws 5.1 and 1.2], and this is an improvement over the behavioural subtyping rules presented by Liskov and Wing in [11], where a less general form of method refinement is used [19, Laws 1.1 and 1.2]. However, rules (ii) and (iii) above omit explicit mention of invariants and, as said in [11], the invariant of a type can be included in the antecedent of any of these rules, because such invariants can always be assumed. Rewriting the rules with explicit mention of the invariants, we get:

(ii). `[ pre m c1 && inv c0 ==> pre m c0 ]  .`

(iii). `[ post m c0 && inv c0`
```
&& old (pre m c1) && old (inv c0)
==> post m c1 ] .
```

Only the invariant of `c0` is included since, by rule (i), if the invariant of `c0` is met then so is also the invariant of `c1`.

3.2 Modular Reasoning, Static Verification, and Runtime Assertion Checking

Let us analyse the conditions that should be statically verified in a method call according to modular reasoning. From the point of view of the *caller*, before execution of the method call the precondition must be checked as an assertion, and after execution of the call the postcondition can be assumed. Now, from the point of view of the *callee*, before execution of the body of the method it can be assumed that the precondition is satisfied, and at the end it must be verified as an assertion that the postcondition is met. Note that we are using the assertion/assumption terminology of JML [5] (which is assertion/coercion in the jargon of the refinement calculus of Morgan [19]). In the presence of inheritance, due to the dynamic method binding philosophy of object-orientation, these assumptions and assertions should be verified in accordance with the types involved in the binding variable-object of the method call: from the point of view of the *caller*, the static type of the variable (or, more generally, expression) used in the call should be used, and from the point of view of the *callee*, the body of the method to be executed depends on the dynamic type of the object bound to the variable/expression of the call.

The previous analysis corresponds to (modular) static verification of method calls and method bodies. However, we are interested in (modular) dynamic verification, that is, runtime assertion checking, in which case both assumptions and assertions are dealt with in the same way: checking if the predicate holds at that point of the execution of the program, as done in JML [5, 20].

Let us now summarise which are the conditions to be dynamically checked, that is, used in runtime assertion checking, in a method call according to the binding (static)variable-(dynamic)ob ject of the call, and remembering that both assumptions and assertions become assertions in dynamic verification:

– Before the call, check the precondition and invariant according to the static type of the variable, because this corresponds to what is expected statically in relation to the correctness of the call.

- Before executing the method body, check the precondition and invariant according to the dynamic type of the object, as this corresponds to the correctness of the method body that must be used according to the dynamic method binding semantics of object- oriented languages.
- Execute the method body according to the dynamic type.

- At the end of the execution of the body of the method, check the postcondition and invariant according to the dynamic type, as this corresponds to the correctness of the method that was actually executed in accordance with the dynamic method binding semantics.
- When the call returns, check the postcondition and invariant depending on the static type, as this is what is statically expected in relation to the correctness of the call.

3.3 Stronger Conditions for Runtime Assertion Checking

Combining the rules of behavioural subtyping presented in section 3.1 with the conditions deduced in the previous subsection 3.2 for runtime assertion checking, we can obtain stronger conditions for runtime assertion checking. These new conditions are equivalent to the previous ones provided behavioural subtyping holds, but without this assumption they are formally stronger. The fact that these new conditions are stronger, when behavioural subtyping is not guaranteed, makes it possible to detect dynamically a greater number of failures, both in relation to the conditions related to the inheritance chain and in relation to the conditions that must locally be met by a method when it is called.

We will present the new stronger conditions, first, for a simple hierarchy of just two classes, and then we will generalise this to an arbitrary hierarchy (of simple inheritance, as we do not consider multiple inheritance).

Let `c0` be a subclass of `c1`, and let both classes declare a method `m`. Let us see the conditions that must be checked in a call to `m`, according to the possible combinations of static/dynamic type that can be involved in the binding variable-object of the call:

- Both static and dynamic type `c1`: from the point of view of the caller and from the point of view of the callee the same conditions should be checked:

```
{  inv c1 && pre m c1 }
and {  post m c1 && inv c1 }  .
```

- Both static and dynamic type `c0`, from the point of view of the caller and from the point of view of the callee, again, the same conditions should be checked:

```
{  inv c0 && pre m c0 }
   and {  post m c0 && inv c0 }  ,
```

but, using rules (i) and (iii) of behavioural subtyping, these conditions can be strengthened to:

```
{  inv c0 && inv c1 && pre m c0 }   and
{  post m c0
&& (old (pre m c1) ==> post m c1)
&& inv c0 && inv c1 }  .
```

- Static type `c1` and dynamic type `c0`: both the point of view of the caller and the point of view of the callee should be checked:

```
{  inv c1 && inv c0 && pre m c1
&& pre m c0 }   and
{  post m c0 && post m c1 && inv m c0
&& inv c1 }  .
```

This analysis of the possible situations in a two-classes hierarchy can be generalised to an arbitrary hierarchy (without multiple inheritance) as follows:

– Before executing the method body:

- check the invariant of all the types in the hierarchy between the dynamic type and the root type of the whole hierarchy, both included, as the invariant of the dynamic type should hold, and rule (i) of behavioural subtyping implies that the invariants of all the other classes higher up in the hierarchy should also hold; and
- check the precondition of all the types in the hierarchy between the static type and the dynamic type, both included, as the one in the static type should be checked from the point of view of the caller, and rule (ii) of behavioural subtyping implies that all the preconditions in classes lower in the hierarchy all the way down to the dynamic type should then also hold.

– After executing the method body:

- as before, check the invariant of all the types in the hierarchy between the dynamic type and the root type, both included, as the one of the dynamic type should hold, and rule (i) of behavioural subtyping implies that the invariants of all the classes higher up in the hierarchy should hold too;
- check the postconditions in all the types in the hierarchy between the dynamic type and the static type, both included, as the one in the dynamic type should be checked from the point of view of the callee, and rule (iii) of behavioural subtyping then implies that all postconditions higher in the hierarchy up to the static type should also be checked, noting that the precondition before executing the method body and the invariant both before and after executing the method body hold in all this section of the hierarchy; and, finally,
- check the implication between the precondition at the beginning of the method body and the postcondition at the end, in all the classes higher in the hierarchy between the static type, not included, and all the way up to the root type, which is again a consequence of rule (iii) of behavioural subtyping, except that in this section of the hierarchy the preconditions did not necessarily hold at the beginning of the method body.

As stated before, these stronger conditions would facilitate the dynamic detection of more failures in pre/postconditions of method calls and also of failures in the conditions related to behavioural subtyping. We will call the stronger condition associated with invariants the *augmented invariant*, the stronger condition associated with preconditions the *augmented precondition*, and the stronger condition associated with postconditions the *augmented postcondition*. All of these augmented conditions will be formalised in a precise way using Haskell in the following section.

4. THE BRIDGE BETWEEN DYNAMIC AND STATIC VERIFICATION

In this section, we first formalise the augmented conditions that were put forward for runtime assertion checking in the previ ous section. Then, we state a theorem that formally links such a dynamic verification of all methods of a class to the corresponding static verification of correctness of all those methods. As said in the introduction, this formal bridge amounts to proving that a hypothetical exhaustive testing through runtime assertion checking would be equivalent to the corresponding static verification, and we believe that this gives a more solid foundation to dynamic verification through runtime assertion checking. Function dvt, for dynamic verification triple, gives the Hoare triple to be dy namically checked for dynamic type dt, in a call from an object-variable or object- expression with static type st, for method m in class dt.

```
data HoareTriple = HT Precondition
                      Instruction
                      Postcondition
```

Fig. 5 A type for Hoare triples.

First and foremost, we introduce a new type that represents Hoare triples, which will be our building blocks to formalise what is verified both dynamically and statically. The definition is in Fig. 5 and it states that a Hoare triple has a precondition, an instruction and a postcondition.

The three basic types used are as already defined in Sect. 2.

```
dvt :: Class -> Class -> Method
      -> HoareTriple
dvt dt st m
       = HT (augmPre dt st m && augmInv dt)
       (instruction m)
       (augmPost dt st m && augmInv dt)
```

Fig. 6 A Hoare triple for dynamic verification.

In the previous section, we proposed conditions for the runtime assertion checking of a method call for any given combination of static/dynamic types with which such a method can be called. Those conditions correspond to a Hoare triple with the augmented invariant and augmented precondition of the method as precondition, and the augmented invariant and augmented postcondition of the method as postcondition. The construction of such a triple, completed with the method body associated with the dynamic type, is formalised in Fig. 6.

Function `dvt`, for *dymamic verification triple*, gives the Hoare triple to be dynamically checked for dynamic type `dt`, in a call from an object-variable or object-expression with static type st, for method `m` in class `dt`.

The *augmented* conditions are then formalised in Fig. 7.

```
augmInv :: Class -> BoolExpr
augmInv c
  | isNothing (super c) = inv c
  | isJust (super c)
    = inv c && augmInv (fromJust (super c))
augmPre :: Class -> Class -> Method
    -> BoolExpr augmPre dt st m
  | dt == st = pre m st
  | dt /= st
    = augmPre (fromJust (super dt)) st m
    && pre m dt
augmPost :: Class -> Class -> Method
  -> BoolExpr augmPost dt st m
  | dt == st && isNothing (super st)
  = post m st
  | dt == st && isJust (super st)
  = post m st
  && augmPost' (fromJust (super st)) m
  | dt /= st
  = post m dt
  && augmPost (fromJust (super dt)) st m
```

Fig. 7 Augmented invariant, precondition and postcondition.

Function `augmInv` gives the augmented invariant for a certain dynamic type, moving through the hierarchy all the way up to the root class. Function `augmPre` gives the augmented precondition for a given combination of dynamic and static type for a method call. To obtain the augmented condition, the hierarchy is examined from the static type all the way down to the dynamic type (although the recursion travels in the opposite direction). Function `augmPost` gives the augmented postcondition for, again, a given combination of dynamic and static type for a method call. To obtain the augmented condition, the hierarchy is examined from the dynamic type all the way up to the static type, and, above the static type, the rest of the hierarchy must be considered using function `augmPost`'. Function `augmPost`', in Fig. 8, obtains the conditions that correspond to each of the classes where the method is declared; for each of them, the implication between the precondition before execution of the method body and the postcondition afterwards must be satisfied.

```
augmPost' :: Class -> Method -> BoolExpr augm-
Post' c m
| isNothing (super c) && declare c m
= (old (pre m c) ==> post m c)
| isNothing (super c)
&& not (declare c m) = True
| isJust (super c) && declare c m
= (old (pre m c) ==> post m c)
&& augmPost' (fromJust (super c)) m
| isJust (super c) && not (declare c m)
= augmPost' (fromJust (super c)) m
```

Fig. 8 Extra augmentation for postconditions.

Note that, unlike `augmPre` and `augmPost`, function augmPost' needs to ask explicitly about local declarations of the method at issue. In the case of `augmPre` and `augmPost`, all classes in the inspected hierarchy must have a declaration of the sought-after method, whether local or inherited, which renders it unnecessary to ask about such declarations.

Inherited methods might make the same pre/post-condition to appear more than one in the final augmented pre/postcondition, which is harmless due to idempotence of conjunction. In the case of `augmPost`', the inspected hierarchy above the static type all the way up to the root might not necessarily have declarations of the method at issue, which is why declarations must be inquired about. All auxiliary functions of function dvt have already been defined. It has thus been fully formalised how to obtain, through `dvt`, the Hoare triple that corresponds to the dynamic verification of a method call for any combination of dynamic/static type with which such a call can be made. We now need to define a function that, given any class, generates all the Hoare triples of all the methods of the class, considering all possible static types from which calls can be made with the given class as dynamic type. Such a function is presented in Fig. 9 under the name `dvts`. Auxiliary recursive function `dvts`' considers all possible static types for a fixed dynamic type; these potential static types are just all the types higher up in the hierarchy between the given dynamic type and the root.

```
dvts :: Class -> [HoareTriple]
dvts c = dvts' c c
dvts' :: Class -> Class -> [HoareTriple]
dvts' c0 c1
| isNothing (super c1) = dvt'
| isJust (super c1)
= dvt'
++ dvts' c0 (fromJust (super c1))
where
dvt' = map (dvt c0 c1) (declaredMethods
c0 c1)
```

Fig. 9 All the Hoare triples of a class for dynamic verification.

In function dvts', to obtain all the Hoare triples through function dvt, we use the list of methods that are offered in both the dynamic type and the static type, and apply dvt to each of them using Haskell function map. Functions for this methods extraction purpose are defined in Fig. 10. Function declaredMethods obtains the list of methods that are offered both in class c and class sc. It uses function methods', which, given a class c, returns all the methods that c offers: either locally declared or inherited from its superclass without redefinition. Function *offers* determines if a class *offers* a certain method, either directly or indirectly.

```
declaredMethods :: Class -> Class
-> [Method]
declaredMethods c sc = filter (offers
sc) (methods' c)

methods' :: Class -> [Method]
methods' c
| isNothing (super c) = methods c
| isJust (super c)
= methods c
++ filter (\m -> not (declare c m)) (methods'
(fromJust (super c)))

offers :: Class -> Method -> Bool offers c
(Meth n _ _ _)
= isJust
(find (\(Meth n' _ _ _) -> n' == n) (meth-
ods' c))
```

Fig. 10 Methods extraction.

Now that methods indirectly offered by a class came up, meaning methods inherited by a class without redefinition, an important subtlety about them must be mentioned. Methods inherited by a class c must meet their specifications from the standpoint of c, even though they are not defined in c. Thus, they must establish the local invariant of c as part of their postcondition, but this cannot be foreseen by the designers/implementors of the superclass of c if such an invariant is allowed to be arbitrary. This problem can be avoided methodologically by restricting invariants so that they refer only to local attributes, that is, declared in the same class and not inherited. Otherwise, inherited methods without redefinition should be re-verified in relation to the invariant of c, but this is inconvenient both practically and methodologically. In practice, the source code of the superclass of c might not be available to the designers/implementors of c, and, from a methodological point of view, such a re-verification would go against principles of object-oriented design and modular reasoning. In-depth explorations of this problem, that is, the problem of verifying invariants locally through modular reasoning, have been made in the context of the so-called Boogie methodology for object invariants [21, 22]. The Boogie methodology proposes much more elaborate solutions to this problem than the simple one we just mentioned.

So far we have already formalised the Hoare triples that correspond to dynamic verification of a class. We still need to formalise the Hoare triples that correspond to the static verification of a class, which we do with the functions in Fig. 11.

```
 svt :: Class -> Method -> HoareTriple svt c
m = HT (pre m c && inv c)
 (instruction m) (post m c && inv c)

 svts :: Class -> [HoareTriple]
 svts c = map (svt c) (methods' c)
```

Fig. 11 All the Hoare triples of a class for static verification.

Function svt, for *static verification triple*, gives the Hoare triple that corresponds to the static verification of a method m in a class c that offers it. This triple has as precondition the conjunction of the precondition and the invariant in class c, as instruction the method body, and as postcondition the conjunction of the postcondition and the invariant in class c. Function svts gives all the Hoare triples of a class, one for each method offered by the class.

```
behaviouralSubtype :: Class -> Bool behav-
iouralSubtype c
 | isNothing (super c) = True
 | isJust (super c)
 = directBSubtype c (fromJust (super c))
 &&
 behaviouralSubtype
 (fromJust (super c))

 directBSubtype :: Class -> Class -> Bool di-
rectBSubtype c0 c1
 = isTheorem (inv c0 ==> inv c1)
 &&
 and (map (directBSubtypeM c0 c1) (declared-
Methods c0 c1))

 directBSubtypeM :: Class -> Class -> Method
 -> Bool directBSubtypeM c0 c1 m
 = isTheorem (pre m c1 && inv c0
 ==> pre m c0)
 &&
 isTheorem (post m c0 && inv c0
 && old (pre m c1)
 && old (inv c0)
 ==> post m c1)
```

Fig. 12 Behavioural subtyping conditions.

Having formalised the Hoare triples of both dynamic and static verification, the only thing we have left to formalise is the notion of behavioural subtyping, which is done with the functions in Fig. 12. Function `behav-iouralSubtype` determines if a given class `c` is a proper behavioural subtype, inspecting the whole hierarchy between `c` and the root class, and verifying that each of these classes satisfies the rules of be havioural subtyping in relation to its direct superclass. Behavioural subtyping is transitive and, hence, it suffices to check only direct superclasses. Auxiliary function `directBSubtype` does the job for two given classes `c0` and `c1`, determining if `c0` meets the criteria for being a direct behavioural subtype of `c1`; that is, rule (i) of behavioural subtyping is satisfied, and each method offered in both classes satisfies rules (ii) and (iii) of behavioural subtyping.

Auxiliary function isTheorem is meant to determine whether a given boolean expression is a theorem. We do not provide a body for it, as its inner workings would not be an important concern to us. It would, instead, be implemented by a theorem prover. We will only use it for reasoning at a higher level, with the signature shown in Fig. 13.

```
   isTheorem :: BoolExpr -> Bool
```

Fig. 13 A function for reasoning.

This function corresponds to the square brackets [...] of Dijkstra et al. [18] that we used previously in Subsect. 3.1.

Finally, we are ready to state our main theorem, that establishes the promised formal bridge between the Hoare triples used for runtime assertion checking, or dynamic verification, and the Hoare triples that correspond to static verification:

Theorem 4.1 (The Formal Bridge).

```
bridge :: Class -> Bool
bridge c = behaviouralSubtype c
           ==> (allValid (dvts c)
           ==
           allValid (svts c))
```

The theorem states that, for every class that is a proper behavioural subtype, the Hoare triples proposed for dynamic verification of the class are all valid if and only if the Hoare triples that correspond to its static verification are `allvalid`. Auxiliary function allValid, presented in Fig. 14, determines whether a list of Hoare triples are all valid.

```
 allValid :: [HoareTriple] -> Bool all-
Valid
 hts = and (map valid hts)
 valid :: HoareTriple -> Bool
```

Fig. 14 Validity of Hoare triples.

This function has a similar purpose to isTheorem above, but we do define it in terms of a more basic one, `valid`, which could again be implemented by a theorem prover and we will use for reasoning at a higher level.

4.1 Simplifying the Main Theorem

Now that we have stated our main theorem, it so happens that it can be simplified due to hypothesis `be-haviouralSubtype` c. Specifically, the Hoare triples for dynamic verification can be simplified, as both their precondition and postcondition are equivalent to simpler expressions under the hypothesis of behavioural subtyping. The formalisation of these simplified dynamic verification Hoare triples, in the form of a new version of function dvt, defined in Fig.15.

The precondition in these new Hoare triples is just the conjunction of the precondition at the static type and the invariant at the dynamic type, and the postcondition is just the conjunction of the postcondition and the invariant at the dynamic type.

```
dvt :: Class -> Class -> Method
         -> HoareTriple
dvt dt st m = HT (pre m st && inv dt)
                 (instruction m)
                 (post m dt && inv dt)
```

Fig. 15 A simplified Hoare triple for dynamic verification.

In languages as JML [4, 5] and Spec# [6], where every subclass is always a behavioural subtype due to the use of specification inheritance [7], this simplified version of the triples can and should be used for dynamic verification, that is, for runtime assertion checking. On the other hand, in languages where behavioural subtyping is not guaranteed, the original version of the triples should be used. Their pre/post-conditions are formally stronger and, thus, they fail in more cases during runtime assertion checking and facilitate the detection of more errors.

To prove our main theorem, it is of course better to use the simpler version of it, but for that we would need to show that our two versions of dynamic verification triples are indeed equivalent. It suffices to prove the following:

Lemmas 4.2.

```
(i).  isTheorem (augmInv dt == inv dt)
(ii). isTheorem (augmPre dt st m &&
augmInv dt == pre m st && inv dt)
(iii). isTheorem (augmPost dt st m &&
augmInv dt == post m dt && inv dt)
```

Provided `dt` is a behavioural subtype and st is a superclass of dt, and (iii) also requires assumption `old (augmPre dt st m && augmInv dt)`, that is, that `augmPre dt st m && augmInv dt` is satisfied in the pre-state.

Note that the first hypothesis corresponds directly to the hypothesis of the main theorem, and the second hypothesis is a consequence of the way Hoare triples are built within `dvts`. The third hypothesis has to do with the way Hoare triples are reasoned about: when reasoning about the postcondition, it is valid to assume that the precondition held in the pre-state, and this third hypothesis is precisely the precondition of our first version of the `dvt-triples`.

A proof for all Lemmas 4.2 and for Theorem 4.1 can be found in the appendix of [9].

5. RELATED WORK

The static-to-dynamic implication of our main theorem is related to the exploration of Leavens and Naumann of supertype abstraction [1, 2]. However, their semantic characterisation of the relevant concepts is much more detailed and, also, their results are much richer than just our staticto-dynamic implication. As mentioned in the introduction, it is our dynamic-tostatic implication what we believe to be a novel exploration. In any case, the connection between our work and these results of Leavens and Naumann regards only a purely theoretical view of our result. Most of the work that we relate to ours has to do with the practical consequences of our theorem in the construction of runtime assertion checking tools. The rest of this section reviews such tools.

For Contract Java [23, 24], its designers propose a scheme for runtime assertion checking very similar to the verifying code we can derive from our main theorem (presented in the extended ver sion of the present article [9] that includes the practical counterpart of our theoretical exploration). Their checks aim at verifying that every subclass is actually a behavioural subtype and, if not, properly inform the user of where the problem is. For this, they take the rules of Liskov and Wing [11] for pre/postconditions and evaluate, after checking the local pre/post-condition, that the hierarchy satisfies Liskov and Wing's rules. A subtle difference with our code is that their design of checks produces several unnecessary re-evaluation of conditions, even at points where the outcome is irrelevant. For example, when the static type matches the dynamic type, and the precondition has already been checked to be true, it is irrelevant to check the inheritance-precondition rule and, yet, they do it. Other differences with our approach include the absence of invariants in their proposal and the use of the older and stronger inheritancepostcondition rule that does not take into account that the superclass-precondition is satisfied before the method execution (recall that our postcondition rule (iii) of behavioural subtyping in subsection 3.1, as presented in [1] and which also corresponds to [25, 19], is a weaker extension of the original one of Liskov and Wing [11]). Additionally, in the case where the static type does not match the dynamic type and the postcondition fails, the error they report to the user is imprecise and different from ours. They report that the postcondition is not met; however, given that the executed code is the one of the dynamic type and the postcondition checked is the one of the static type, this message is imprecise for the user: the fault may lie with the implementation of the dynamic type that does not ensure its postcondition, or with the hierarchy between the dynamic type and the static type that has a class that is not a behavioural subtype. To get as much detail as possible regarding the failure of postconditions, in our verifying code we first check the postcondition of the dynamic type and later proceed with the rest of the postconditions higher in the inheritance hierarchy; if any of the postconditions fails, we can precisely report to the user whether the method code in the dynamic type does not meet its postcondition, or whether some other postcondition higher in the hierarchy fails, which makes the class immediately below (whose postcondition did succeed) an incorrect behavioural subtype.

Regarding iContract [26], our proposal differs a lot from the conditions they verify. For each method defined in both classes of a two-classes hierarchy, iContract checks as precondition the disjunction of the preconditions annotated in the class and in the superclass, as postcondition the conjunction of the postconditions annotated in the class and the superclass, and as invariant the conjunction of the invariants of the class and the superclass. These conditions do not correspond to the contracts that were written by the programmer. With these conditions, a method could even be executed starting in a state that does not meet its own precondition, if its superclass-precondition holds but its own does not.

jContractor [27] verifies the same conditions as iContract. Therefore, it suffers from the same problems just pointed out.

Jass [28] gives programmers the possibility to decide whether a subclass should be verified as a behavioural subtype or not. This is a possibility offered by Jass that we do not consider, as we believe that semantic cleanness must be a part of a good object-oriented programming language. The rules they check on behavioural subtyping correspond to those of Liskov and Wing [11], with the modification of Leavens and Naumann [1]. However, they do not take into account the static types associated with method calls and, therefore, their verification is not consistent with the static verification of the call from the point of view of the client.

JML [4, 20, 5] includes specification inheritance [7] for all subclasses and, therefore, every subtype is always a behavioural subtype. As mentioned towards the end of Sect. 4 when we presented the simplified version of our theorem, a runtime assertion checking tool based on our results could use the simpler weaker conditions when behavioural subtype is guaranteed, or otherwise use the stronger conditions so that it is possible to detect design problems in the class hierarchy at runtime. Our proposal can thus be seen as giving more freedom to the specifiers of subtypes, both allowing that every subclass is ensured to be a behavioural subtype, through specification inheritance or any other mechanism, and also allowing that such a guarantee is not given.

Our proposal also differs from the conditions verified in JML when the binding variable-object corresponds statically to a class and dynamically to one of its subclasses. We explain this in detail with a small example. Take a hierarchy of two classes, with superclass A and subclass B; they both define method m, with specifications given by the user [preA, postA] and [preB, postB] , respectively. Due to specification inheritance, the real specification in the subclass ends up being `[ preA || preB, (old preA ==> postA) && (old preB ==> postB) ]`.

The runtime assertion checker of JML verifies, in the case that the object is dynamically from the subclass, the real specification in the subtype without taking into account the static type associated with the call. However, for the dynamic verification to match the static verification, this is not what should be checked. The dynamic verification should check `[ preA, postA && (old preB ==> postB) ]`. Statically, it is only known the static type associated with the call, and so the specification of A should be met. This corresponds to the object of the subclass satisfying the contract of the superclass: it must abort the program if `preA` is not met at the beginning and ensure that `postA` is satisfied at the end. Note that, provided preA holds at the beginning, the postconditions checked by JML and by us are equivalent but, when `preA` does not hold and `preB` does, JML does not announce the error (the caller did not guarantee the required static precondition) but our proposed verification does.

Modern Jass [29] includes specification inheritance and verifies precisely the same conditions as JML [4]. Therefore, it also differs from our proposal on the verification performed on the precondition when the call is done with a dynamic type that does not match the static type.

Spec# [6] restricts subtypes in a way that they are behavioural subtypes, offering, as well as JML, specification inheritance. However, it is more restrictive with respect to the preconditions that can present in a subtype: preconditions must remain the same. Although it is more restrictive with respect to the potential subtypes, the checks are appropriate in any situation. Comparing this to our proposal, note that, with the precondition restriction of Spec#, the expression `preA || preB` of our JML example, ends up being just `preA` and, thus, the proposal of Spec# ends up being equivalent to ours. However, we consider the possibility of not forcing specification inheritance, and even in the case that it is forced, we do not force the precondition in the subclass to be the same of the superclass. It suffices to take into account the static type in the conditions to be verified.

6. CONCLUSIONS

We have established a formal theoretical connection between runtime assertion checking of method calls in the presence of inheritance and the corresponding correctness static verification of such calls. This was formalised through Theorem 4.1, a proof of which is presented in the appendix of [9]. We believe this formal connection to be important, as it provides a more solid foundation to runtime assertion checking, and we have not found in the literature of this subject any such formal relationship to have been established previously.

Also, our theoretical result allowed us to determine precise conditions to be used in runtime assertion checking, making it possible to dynamically detect all kinds of failures in an inheritance relationship (we present code in the extended version of this article [9]). Our proposed conditions also allow programmers to design classes without restricting the specifications of redefined methods in subclasses, that is, without forcing specification inheritance; testing and dynamic verification of specifications would then be used to find errors or to obtain a high degree of certainty about the correctness of the design of a subclass as a behavioural subtype.

7. FUTURE WORK

Other aspects related to this work that might be studied in more depth in the future are the following:

– Strengthening our main theorem so that it does not depend on the behavioural subtyping hypothesis; that is, establishing a relationship between the dynamic conditions and all the static conditions including behavioural subtyping. Recall our main theorem:

```
behaviouralSubtype c
==>  (allValid (dvts c)
==
allValid (svts c)) .
```

Considering the satisfaction of behavioural subtyping as one more static condition, a stronger theorem would be:

```
allValid (dvts c)
==
behaviouralSubtype c
&&
allValid (svts c) ,
```

with conjunction binding stronger than equivalence. This new proposition would really show dynamic verification to be equivalent to all the corresponding static verification. However, with our formalisation, this proposition is not a theorem. Nevertheless, we believe that under a more detailed formalisation this proposition can be proved to be a theorem. An extra detail we believe to be missing is the explicit formalisation of execution states as done by, for example, Leavens and Naumann in [1].

– Stating and proving a theorem that combines inheritance and data refinement.

8. REFERENCES

[1] G. T. Leavens and D. A. Naumann, “Behavioral subtyping, specification inheritance, and modular reasoning,” Tech. Rep. 06-20b, Department of Computer Science, Iowa State University, Sept. 2006.

[2] G. T. Leavens and D. A. Naumann, “Behavioral subtyping is equivalent to modular reasoning for object-oriented programs,” Tech. Rep. 06-36, Department of Computer Science, Iowa State University, Dec. 2006.

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

[4] G. T. Leavens and Y. Cheon, “Design by contract with JML.” Available from www.jmlspecs.org (The Java Modeling Language (JML) home page), 2005.

[5] G. T. Leavens, E. Poll, C. Clifton, Y. Cheon, C. Ruby, D. Cok, P. Mu¨ller, J. Kiniry, and P. Chalin, JML Reference Manual, Nov. 2007.

[6] M. Barnett, K. R. M. Leino, and W. Schulte, “The Spec# programming system: An overview,” in Construction and Analysis of Safe, Secure, and Interoperable Smart Devices, CASSIS 2004, vol. 3362 of Lecture Notes in Computer Science, pp. 49–69, Springer, 2004.

[7] K. K. Dhara and G. T. Leavens, “Forcing behavioral subtyping through specification inheritance,” in Proceedings of the 18th International Conference on Software Engineering, Berlin, Germany, pp. 258–267, IEEE Computer Society Press, 1996.

[8] K. Arnold, J. Gosling, and D. Holmes, The Java Programming Language. Addison Wesley Professional, fourth ed., 2005.

[9] G. Montoya and J. Ravelo, “A formal bridge between runtime assertion checking and static verification of object-oriented programs in the presence of inheritance –theory and practice–,” tech. rep., Departamento de Computacion y Tecnolog´ia de la Informacion, Universidad Simon Bol´ivar, Caracas, Venezuela, October 2009.

[10] B. Liskov and J. Guttag, Program Development in Java: Abstraction, Specification and ObjectOriented Design. Addison-Wesley, 2001.

[11] B. H. Liskov and J. M. Wing, “A behavioral notion of subtyping,” ACM Transactions on Programming Languages and Systems, vol. 16, pp. 1811–1841, Nov. 1994.

[12] A. Mikhajlova and E. Sekerinski, “Class refinement and interface refinement in object-oriented programs,” in FME’97: Industrial Applications and Strengthened Foundations of Formal Methods (Proc. 4th Intl. Symposium of Formal Methods Europe, Graz, Austria, September 1997) (J. Fitzgerald, C. B. Jones, and P. Lucas, eds.), vol. 1313, pp. 82–101, Springer-Verlag, 1997.

[13] R.-J. Back, A. Mikhajlova, and J. von Wright, “Class refinement as semantics of correct object substitutability,” Formal Aspects of Computing, vol. 12, pp. 18–40, Oct. 2000.

[14] R.-J. Back, L. Mikhajlov, and J. von Wright, “Formal semantics of inheritance and object substitutability,” Tech. Rep. TUCS-TR-337, Turku Centre for Computer Science, 27, 2000.

[15] S. Thompson, Haskell: The Craft of Functional Programming. Addison Wesley, Mar. 1999.

[16] R. Bird, Introduction to Functional Programming using Haskell. Prentice Hall, second ed., 1998. [17] G. Hutton and J. Wright, “Compiling excep tions correctly,” in Proceedings of the 7th International Conference on Mathematics of Program Construction, pp. 211–227, Springer, 2004.

[18] E. Dijkstra and C. Scholten, Predicate Calculus and Program Semantics. Texts and Monographs in Computer Science, Springer-Verlag, 1990.

[19] C. Morgan, Programming from Specifications. In ternational Series in Computer Science, Prentice Hall, 2nd ed., 1994.

[20] Y. Cheon, “A runtime assertion checker for the Java Modeling Language,” Tech. Rep. 03-09, Department of Computer Science, Iowa State University, Apr. 2003.

[21] M. Barnett, R. DeLine, M. Fahndrich, K. R. M. Leino, and W. Schulte, “Verification of object-oriented programs with invariants,” Journal of Object Technology, vol. 3, no. 6, pp. 27–56, 2004. [22] K. R. M. Leino and P. Mu¨ller, “Object invariants in dynamic contexts,” in ECOOP 2004 – Object-Oriented Programming,

vol. 3086 of Lec ture Notes in Computer Science, pp. 491–515, Springer, 2004.

[23] R. B. Findler, M. Latendresse, and M. Felleisen, "Behavioral contracts and behavioral subtyping," SIGSOFT Software Engineering Notes, vol. 26, no. 5, pp. 229–236, 2001.

[24] R. B. Findler and M. Felleisen, "Contract soundness for object-oriented languages," in ACM Conference on Object Oriented Programming, Systems, Languages and Applications (OOP-SLA'01), pp. 1–15, 2001.

[25] C. Morgan and K. Robinson, "Specification statements and refinement," IBM Journal of Research and Development, vol. 31, no. 5, pp. 546–555, 1987.

[26] R. Kramer, "iContract - the Java design by contract tool," in Proceedings of Technology of Object-Oriented Languages, pp. 295–307, 1998.

[27] M. Karaorman, U. Hölzle, and J. Bruno, "jContractor: A reflective Java library to support design by contract," Tech. Rep. TRCS98-31, University of California at Santa Barbara, 1998.

[28] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim, "Jass – Java with assertions," in Runtime Verification (K. Havelund and G. Ro¸su, eds.), vol. 55 of Electronic Notes in Theoretical Computer Science, Elsevier, July 2001.

[29] J. Rieken, "Design by contract for Java revised," Master's thesis, Carl von Ossietzky Universität Oldenburg, Apr. 2007.