Polymorphism in Object-Oriented Contract Verification

I.Nunes
Lisbon University, Faculty of Sciences, Informatics Department
Bloco C5, Piso 1, Campo Grande, 1749-016 Lisbon Portugal
Tel: +351 21 7500602 Fax: +351 21 7500084 e-mail: in@di.fc.ul.pt

Abstract. The role that design by contract plays in the specification, monitoring, and reuse of classes is of increasing importance in the OO community. Although recognizably useful, nevertheless, because of lack of expressiveness of existing assertion languages, contracts can bring undesirable effects such as the increasing in class coupling when we deal with complex classes. A means of surpassing those problems was proposed in [7] – the concept of meta-assertion. These are assertions that allow to refer to other methods' assertions. The semantics that was given in [7] for meta-assertions, used static binding when deciding which assertions to verify. In that paper it was pointed out, as an important goal, that the dynamic type of the target object should be used when deciding which assertion to verify, in order to have, at the level of assertions, the same dynamic binding mechanism that we have in OO method calls. In this paper we present a semantics for meta-assertions and a process of translating them into monitorable assertions, that dinamically defines the assertions that are to be verified, allowing the polymorphism in meta-contract verification.

KeyWords: OO program verification, Design by contract, meta-assertions, contract monitoring, polymorphism.

1 Introduction

The specification of class invariants and pre- and post-conditions for methods in a class that supplies services to other classes – call it the supplier class – is a means of establishing the rules of business with its clients – the classes that use it. In this way we minimize the errors that are due to the lack of information about classes and the services they supply.

There are several assertion languages that allow the specification of assertions in different, but yet similar, ways. Some, like Eiffel [6], are included in the programming language itself; others, like iContract [3], have Java boolean expressions as its base and allow to write assertions for Java programs that are preprocessed into Java commands. There are many others as for example COLD-1 [4], Jass [1], ContractJava [2], Larch family [Guttag85], JML [5] among them. Some of these – Jass, iContract, Eiffel – as well as others, allow the monitoring of contracts at runtime.

In [7, 8] we claimed that these languages are not expressive enough to allow to write more complex contracts without bringing undesirable effects like, e.g., side-effects in contract monitoring, or class coupling increasing. We introduced meta-assertions as a means of writing simple, clean, and yet very expressive, contracts that do not suffer from those deffects. We proposed a process of expansion of meta-assertions that allowed for the use of existing tools that generate monitoring code from assertions. This process, as well as the semantics we gave for meta-assertions, used static binding in what concerns the choice of the assertions to be verified. The dynamic binding in meta-assertion semantics and expansion process was referred in [7] as an important goal.

In [2] the assertions that are monitored in each method call are the ones that accord to the static type of the target object. Although the hierarchy of pre- and post-conditions is checked for violation of the "don't ask for more, don't offer less" principle [6], the fact is that one cannot bennefit from stronger/weaker post/pre-conditions in subclasses when polymorphic variables are at stake. Unlike this approach, we claim that the binding of invoked methods assertions with their target objects, when monitoring assertions, should be a dynamic one, that is, it should regard the dynamic type of the target object. Obviously this extends to meta-assertions as well. We present the semantics and expansion process for meta-assertions according to this claim.

The paper consists of four sections. In the next section we motivate the need for meta-assertions and discuss about which version – static vs dynamic – of assertions to monitorize. We also present the formal syntax and semantics of meta-assertions in what they differ from the ones presented in [7]: the assertions that are monitored are the ones that conform to the dynamic type of the target object. In section 3 we present the rules that define how to expand and modify a class in order to allow dynamic meta-contract monitoring. Section 4 presents the conclusions.

2 Meta-Assertions

The constructs we introduced in [7] for meta-assertions are `»pre` and `»post`, that are used to represent, respectively, the pre-condition and the post-condition of the method to which they are applied.

2.1 Why Meta-Assertions

Let us recall the example in [7] of a class `Drawing` that represents a structured collection of polygons – objects of class `Polygon`. A polygon, in turn, represents a structured collection of points – its vertices – that are objects

of class `Point`. All three classes define a method `move(int dh, int dv)` that moves the object to which it is applied by a horizontal distance `dh` and a vertical distance `dv`.

With the expressiveness of the languages of assertions that we have at our disposal, we would have to follow one of three undesirable ways if we wanted to write the `Drawing`'s `move(...)` post-condition – "the result of moving a drawing is the same as moving all its polygons":

- create several and otherwise useless methods that would, in this case for example, i) reveal all about the class's internal objects and the internal objects of these ones or ii) that would inform whether a polygon had been moved for given distances;
- introduce side effects in the assertion, that would entail changes to the objects at the time the assertion was monitored:

```
ensures forall int i in 1..polies() |
           poly(i).equals((old poly(i)).move(dh,dv))
```

- increase class coupling by testing the polygons vertices:

```
ensures forall int i in 1..polies() |
          forall int j in 1..poly(i).vertices() |
            (poly(i).vertex(j).x() == (old poly(i)).vertex(j).x()+dh) &&
            (poly(i).vertex(j).y() == (old poly(i)).vertex(j).y()+dv)
```

When we introduce meta-assertions we avoid going any of these ways. Instead, we can write a simple, clean, and low coupling preserving post-condition for the `Drawing` `move` method:

```
ensures forall int i in 1..polies() | poly(i).move(dh,dv)»post
```

The intended meaning is: the post-conditions of all commands `move` applied to all the drawing polygons are true in the resulting state. These meta-assertions refer to assertions, not to methods. So, when they are monitored, there is no execution of methods `move` but, instead, the evaluation of their post-conditions.

In this way, we are able to represent the result of an operation by writing the conditions that are of the direct responsibility of the enclosing class only. We do this without creating unnecessary query methods for querying objects that are "strangers" to client classes.

2.2 Dynamic vs Static Binding in Meta-Assertion Monitoring

When we test the pre- and post-conditions regarding the static type of the target objects, there can be method invocations that are forbidden (due to pre-condition violation) and that should not be forbidden, and others that do not cause post-condition violation but they should. The following classes exemplify this problem:

```
public class Account {
/** requires x ≤ balance()
*   ensures balance() == old balance() - x
*/
public void withdraw (int x) {...}
....
} // end class Account

public class RevenueAccount extends Account{
/** requires x ≤ balance() + owner().revenue()
*   ensures balance() == old balance() - x
*/
public void withdraw (int x) {...}
...
} // end class RevenueAccount

public class SpecialAccount extends Account{
/** requires x ≤ balance() + credit()
*   ensures balance() == old balance() - x
*   ensures x == old balance() + old credit() implies newPenalty()»post
*/
public void withdraw (int x) {...}
...
} // end class SpecialAccount
```

Suppose we have an array of objects of type `Account` that contains `Account` objects as well as `RevenueAccount` and `SpecialAccount` ones. Suppose we call the method `withdraw(amount)` over an element of that array which dynamic type is `SpecialAccount`, and where the value of `amount` is bigger than the account's `balance()` but smaller than the account's `balance()` plus the account's `credit()`. If the pre-condition that is tested is the `Account`'s `withdraw` one (as happens in [2]) that call will be rejected. This should not happen because in what the `SpecialAccount` object is concerned, the withdrawal of `amount` is perfectly legal.

Suppose now that the `withdraw()` method in class `SpecialAccount` is ill-implemented, that is, it does not establish its post-condition. If the post-condition that is monitored is the `Account's` `withdraw()` one (the one

that conforms the static type of the target object), it becomes impossible to trap the cases which violate the `SpecialAccount`'s `withdraw()` post-condition.

When we deal with meta-assertions the concern is the same: we want to monitor the versions of the pre- and post-conditions that conform to the dynamic type of the target object. Consider the example:

```
public class MyClass {
// The accounts of the clients
private Account [] clientAccounts

/** Withdrawal of a given amount from a given account
*   requires acc!=null && acc.withdraw(amount)»pre
*   ensures acc.withdraw(amount)»post
*/
public void withdraw (double amount, Account acc) {...}
...
} // end class MyClass
```

Suppose now that the array `clientAccounts` has `howMany()` accounts, and that these accounts are instances of types `Account`, `SpecialAccount` and `RevenueAccount`. If, in `MyClass`, we have

```
for (int i=0; i<howMany(); i++)
       clientAccounts[i].withdraw(clientAccounts[i].tax())
```

we want to be able to withdraw the necessary amount from a `RevenueAccount` even if that amount exceeds the account's balance. As long as it does not exceed the balance plus the owner's revenue, all is well. If the pre-condition of the `withdraw` method that is chosen to be verified from `acc.withdraw(amount)»pre` is the one conforming to the `acc` static type – `Account` – this is not possible. The one conforming with its dynamic type should be chosen to be monitored.

We also want that, in case the amount withdrawn from a `SpecialAccount` is equal to the account's available amount, a new penalty is issued to that account. Again, if the post-condition of the `withdraw` method that is chosen to be verified from `acc.withdraw(amount)»post` is the one conforming to the `acc` static type, this is not guaranteed. In a situation where a new penalty should be issued but it is not, the monitoring does not detect any violation. The post-condition conforming with the target object dynamic type should be chosen to be monitored.

2.3 Syntax and Semantics of Meta-Assertions

By limitations of space we refer the reader to [7] where the syntax of the meta-assertion language MAL, which is an existing assertion language AL (for programming language PL) extended with meta-constructs, is defined. In what respects the semantics, we will only present the semantics of MAL in what it differs from the one presented in [7]. The difference lies in the Type rules: here they give us the dynamic type of references (while in [7] they dealt with static types).

The semantics was given operationally through a set of rules that, given a configuration that includes, among other elements, a meta-assertion and a state, gives a boolean value. The type rules are now:

[Type1]:

$$\frac{ty,mth \vdash s\ p \longrightarrow_{type} ty_1 \qquad Meth(ty_1,mc)=mth_1}{ty,mth \vdash s\ applyP(p,mc) \longrightarrow_{type} TypeOfMeth(ty_1,mth_1)}$$

[Type2]:

$$\frac{ty,mth \vdash s\ p \longrightarrow_{type} ty_1 \qquad Attr(ty_1,atc)=attr_1}{ty,mth \vdash s\ applyP(p,atc) \longrightarrow_{type} TypeOfAttr(ty_1,attr_1)}$$

[Type3]:

$$\frac{}{ty,mth \vdash s\ ref \longrightarrow_{type} first(s(yo)[ref])}$$

Where yo is *old* if ref is applied the old construct of AL and *young* if that is not the case.

These rules give us the type of a path. They establish a relation between elements of Path and TYPEID given a context composed of a pair of elements of TYPEID and METHID which represent the type or class and the method within which the path appears. The type of a path is the type of the tail method/attribute of the path. The form *ref* for the path applies to the other possibilities, such as the current object, or a formal parameter: that is why the state, s, is needed. States, as defined in [7], give us the information about the dynamic type of an object. `first(s(yo)[ref])` gives us the type of the object denoted by `ref`. The possible values for `yo` - *old* and *young* - are the state components that refer to objects in the pre-execution and post-execution states respectively.

Suppose we want to evaluate the basic-meta of the post-condition of the `withdraw` method of class `MyClass` which is `acc.withdraw(amount)»post`

<MyClass, withdraw, applyP(acc,withdraw(amount))»post, s, Lm> ⟶ ?

where *s* has information about the instance of class `MyClass` to which was applied the method `withdraw(amount,acc)`, and also about object `acc` of dynamic type `SpecialAccount` (and `Account` as static type). The result of this evaluation, by rule [basicM1] (see [7]), is the same as we get by evaluating the meta-assertion (notice the substitution of parameters – `amount` by `x`):

```
amount == old balance() + old credit() implies newPenalty()»post
```

in a state where the current object is not a `MyClass` object as it was before, in [7], but, instead, the `SpecialAccount` object referred to by `acc` (the target object of the basic-meta `acc.withdraw(amount)»post`).

3 Monitoring Meta-Contracts

If contracts are to be checked at runtime, we must free meta-assertions from basic-metas, so that the existing monitoring code generation tool for AL/PL that is to be used, can generate runtime checking code from simple assertions in the base assertion language AL. We use the functions and sets of identifiers defined in [7] for the semantics of meta-assertions – in order to abstract away the most from the details of the base assertion and programming languages – plus the syntactic category Prog that stands for programs in the base programming language for AL, and function *Body*: METHID → Prog that gives the body of a given method.

3.1 Augmenting classes

We first define the process of expanding a type in order to augment it with pre- and post-methods for each of the original methods that have non-empty pre- and/or post-conditions.

[Expand]:

$$\frac{\begin{array}{l}\text{MembList(ty)} = \{\text{mth}_0 \ldots \text{mth}_n, \text{attr}_0 \ldots \text{attr}_q\} \\ \text{MembList(ty')} = \{\text{mth}_0 \ldots \text{mth}_n, \text{mth\_post}_0 \ldots \text{mth\_post}_m, \text{mth\_pre}_0 \ldots \text{mth\_pre}_p, \text{attr}_0 \ldots \text{attr}_q\} \\ \text{Cond(ty,mth}_j\text{,post) non-empty} \qquad \text{ty} \vdash \text{mth}_j \rightarrow_{\text{post}} \text{mth\_post}_k \qquad \text{for } j \in [0,n], \text{ for } k \in [0,m] \\ \text{Cond(ty,mth}_j\text{,pre) non-empty} \qquad \text{ty} \vdash \text{mth}_j \rightarrow_{\text{pre}} \text{mth\_pre}_l \qquad \text{for } j \in [0,n], \text{ for } l \in [0,p]\end{array}}{\vdash \text{ty} \rightarrow_E \text{ty'}}$$

[Post]:

$$\frac{\begin{array}{lll}\text{Params(ty,mth)} = \{(\text{ty}_0, \text{par}_0) \ldots (\text{ty}_n, \text{par}_n)\} & \text{TypeOfMeth(ty,mth)} = \text{ty'} & \text{Cond(ty,mth,post)} = \text{P} \\ \text{Params(ty,mth\_post)} = \{(\text{ty}_0, \text{par}_0) \ldots (\text{ty}_n, \text{par}_n), \text{ty o}\} & & \text{TypeOfMeth(ty,mth\_post)} = \text{Boolean} \\ \text{Body(mth\_post)} = \texttt{return true} & & \text{Cond(ty,mth\_post,post)} = \text{P}[\text{o} \rightarrow \texttt{old cur()}]\end{array}}{\text{ty} \vdash \text{mth} \rightarrow_{\text{post}} \text{mth\_post}}$$

[Pre]:

$$\frac{\begin{array}{ll}\text{Params(ty,mth\_pre)} = \text{Params(ty,mth)} & \text{Cond(ty,mth\_pre,pre)} = \text{Cond(ty,mth,pre)} \\ \text{TypeOfMeth(ty,mth\_pre)} = \text{Boolean} & \text{Body(mth\_pre)} = \texttt{return true}\end{array}}{\text{ty} \vdash \text{mth} \rightarrow_{\text{post}} \text{mth\_pre}}$$

The rule [Expand] picks a type and adds it as many methods as there are non-empty pre- and post-conditions in its original methods. For each method *mth* of the type that has a non-empty pre-condition, a method *mth_pre* is added to the type. For each method *mth* of the type that has a non-empty post-condition, a method *mth_post* is added to the type.

Rule [Pre] defines the way in which the *mth_pre* is built from *mth*: the signature is the same except for the return type which is type boolean for *mth_pre*; its body is a sole statement that returns the value `true`.

Rule [Post] defines the way in which the *mth_post* is built from *mth*: it has an extra parameter – *o* – which type is the type that contains *mth*; its post-condition is the same as *mth*'s post-condition with *o* substituted by `old cur()`; its return type is boolean; its body is a sole statement that returns the value `true`.

The purpose of parameter *o* in *mth_post* is to be able to send to this method the old value of the path to which *mth* is applied. Because the post-condition of *mth_post* is the same as *mth*'s one, if we substitute *o* for `old cur()` in it, we obtain the reference to the old version of the object to which is applied the method that has, in its own post-condition, a basic-meta that refers to *mth*'s post-condition (type *ty* in the rule).

The reason why the body of the *mth_pre* method is simply the returning of the true value, has to do with the fact that the purpose of *mth_pre* is the evaluation of *mth*'s pre-condition. Because *mth_pre*'s pre-condition is exactly the same as *mth*'s one, and because this new expanded type will still suffer from the generation of code in the body of its methods to evaluate assertions, the final pre-processed code for *mth_pre* will be the evaluation of its pre-condition followed by the returning of true – if the pre-condition is true – or followed by an exception raising – if it evaluates to false. The same applies to *mth_post*.

In the `MyClass` example above, the types `MyClass`, `Account`, `SpecialAccount` and `RevenueAccount` would be added `m_pre` and `m_post` methods corresponding to all methods `m` with non-empty pre- or post-conditions. As an example, class `Account` would be added the following methods:

```
/** requires x ≤ balance()
*/
public boolean withdraw_pre(int x) {return true}
```

(where one can notice the body of the method that simply returns the value true, and the same pre-condition as the one in the method that originated it – the `withdraw` method of `Account`) and

```
/** ensures balance() == o.balance() - x
*/
public boolean withdraw_post(int x, Account o) {return true}
```

Notice the extra-parameter of the same type as the type being augmented (when augmenting type `SpecialAccount` for example, the type of `o` would be `SpecialAccount`); the body of the method that simply returns the value true; the same post-condition as the one in the method that originated it – the `withdraw` method of `Account`; the substitution of parameter *o* by `old cur()` in all places throughout the post-condition.

3.2 Producing monitorable assertions

Finally, the rules for freeing meta-assertions from basic-metas, that is, that define the transformation of a meta-assertion into a simple assertion:

[Un-meta]:

$$\frac{\text{BasicM}(ma)=\{bm_0 \ldots bm_n\} \qquad \text{where } bm_j = \text{applyP}(p_j, mc_j)\text{»prest}_j \quad \text{for } j \in [0,n] \qquad ty\ mth \vdash bm_j \longrightarrow_{\text{lowM}} a_j}{ty, mth \vdash ma \longrightarrow_{\text{Un-M}} ma[a_0/bm_0 \ldots a_n/bm_n]}$$

[LowerMeta]:

$$\frac{\begin{array}{l}\text{ActualPar}(mc)=\{exp_0 \ldots exp_m\} \\ \text{ActualPar}(mc\_prest)=\{exp_0 \ldots exp_{m.}, \text{old p}\} \text{ if prest=post} \qquad \text{ActualPar}(mc\_prest)=\{exp_0 \ldots exp_{m.}\} \text{ if prest=pre}\end{array}}{ty\ mth \vdash \text{applyP}(p,mc)\text{»prest} \longrightarrow_{\text{lowM}} \text{applyP}(p,mc\_prest)}$$

[CurPath]:

$$\frac{\vdash \text{applyP}(\texttt{cur()}, mc)\text{»prest} \longrightarrow_E a}{\vdash mc\text{»prest} \longrightarrow_E a}$$

For a given meta-assertion *ma*, the [Un-meta] rule gives a meta-assertion that is equal to *ma* with calls to pre- and post-methods substituting all basic-metas. This is done by applying rule [LowerMeta] to all *ma*'s basic-metas obtaining simple-assertions that are substituted by *ma*'s basic-metas in *ma*. Notice that the detailed syntax of a meta-assertion *ma* is not relevant here. The important thing here is that simple-assertions are substituted for basic-metas within a meta-assertion. The structure of the meta-assertion is kept the same.

Rule [LowerMeta] determines the substitution of a call, *mc_prest*, to a pre- or post-method (depending on whether »prest is pre or post) by a basic-meta envolving *mc*. The *mc_prest* method is called with the same arguments as *mc* is, (plus the extra argument that stands for the old version of the path *p* to which *mc* applies, if prest denotes post).

In the `MyClass` example above, when expanding the meta-post-condition of the `withdraw` method of class `MyClass`, we would apply the [LowerMeta] rule to the basic-meta: `acc.withdraw(amount)»post`. When applying the [LowerMeta] rule to it, the following meta-assertion would be obtained:

```
acc.withdraw_post(amount, old acc)
```

At runtime, when the withdraw method of `MyClass` is executed and its post-condition evaluated, the method `withdraw_post` of `acc`'s dynamic type is called with the arguments `amount` and the old version of `acc`. If `acc`'s dynamic type is, for example, `SpecialAccount`, then the method

```
/** ensures balance() == o.balance() - x
*   ensures x == o.balance() + o.credit() implies newPenalty()»post
*/
public void withdraw_post (int x, SpecialAccount o) {...}
```

will be executed and its own post-condition will be evaluated (which is exactly the same assertion as `SpecialAccount withdraw`'s post-condition with the substitution of `o` by `old cur()`). There, `(o.balance())` represents the value of the `balance` function of the account `acc` before having been modified by method `withdraw` of class `MyClass`. If we had not made the substitution `o→old cur()`, the `(old balance())` would be evaluated returning the value of the `balance()` of the new version of the `acc` account. This is because the `SpecialAccount withdraw` post-condition is only evaluated after the `MyClass withdraw` method has been executed and its `balance()` modified. The `SpecialAccount withdraw_post` method would be applied to this new version of `acc` and so, the `(old balance())` would test the new value of `balance()` instead of the old one. Remember that in post-methods, because they never change anything, the old

values (the ones before the execution of the method) always equal the new ones (the ones after the execution of the method).

By limitations of space, we do not present here the proof of soundness of the expansion. In a final version of the paper this proof should be included.

4 Conclusions

Meta-assertions were introduced in [7, 8] as a means to surpass the lack of expressiveness of existing assertion languages to write contracts for complex classes (tipically classes that are clients of other classes). The increasing in class coupling that the specification of contracts would bring to a system of classes is avoided by the use of meta-assertions. Moreover, very simple, and yet very powerful, contracts may be written using this construct. In [7] a semantics was presented for meta-assertions that used static binding when deciding which assertions to verify. In that paper it was pointed out, as an important goal, that the dynamic type of the target object should be used when deciding which assertion to verify, in order to have, at the level of assertions, the same dynamic binding mechanism that we have in OO method calls.

In this paper we presented the semantics of meta-assertions taking into account the dynamic verification of assertions, allowing polymorphic meta-assertions to be monitored at run-time according to the dynamic type of the target objects. We also proposed a mechanism, defined by a set of rules, for transforming classes with meta-contracts in classes with monitorable contracts.

References

1. D.Bartezko, C.Fischer, M.Moller and H.Wehrheim: Jass-Java with assertions, Workshop on RunTime Verification, 2001.
2. R.B.Findler and M.Felleisen, Contract Soundness for Object-Oriented Languages, OOPSLA 2001.
3. iContract HomePage. http://www.reliable-systems.com/tools/iContract/iContract.htm.
4. H. B. M. Jonkers, Upgrading the Pre- and Postcondition Technique, VDM Europe (1), pp.428-456, 1991.
5. G.T. Leavens, K.R.M. Leino, E. Poll, C. Ruby, and B. Jacobs, JML: notations and tools supporting detailed design in Java, OOPSLA 2000 Companion.
6. B.Meyer, Object-Oriented Software Construction, 2nd edition, , Prentice-Hall PTR, ISBN 0-13-629155-4, 1997.
7. I.Nunes, Design by Contract Using Meta-Assertions, in Journal of Object Technology, vol. 1, no. 3, special issue:TOOLS USA 2002 proceedings, pages 37-56. http://www.jot.fm/issues/issue_2002_08/article3.
8. I.Nunes, Design by Contract Using Meta-Assertions, Technical Report DI/FCUL, TR-02-7. Dept. of Computer Science, Lisbon Univ. July 2002.