JOURNAL OF OBJECT TECHNOLOGY

Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2002**

Vol. 1, No. 3
Special Edition: TOOLS USA 2002 proceedings

Design by Contract Using Meta-Assertions

Isabel Nunes, University of Lisbon, Portugal

Abstract

Contract writing for methods in classes that are clients of other classes can bring undesirable effects, like the increasing in class coupling and encapsulation decreasing. We propose a pattern to the design of class contracts that helps producing contracts that preserve low class coupling and data encapsulation. The expressive power of existing assertion languages is insufficient, however, to write these contracts. In order to fill this lack we propose meta-assertions, and we define rules for a grammatically and semantically sound expansion of meta-assertions in order to be able to monitor contracts at run-time using already existing tools.

1 INTRODUCTION

The use of contracts to establish the rights and obligations of clients and suppliers is becoming widely accepted in the construction of reliable object-oriented software systems.

In what concerns the construction of classes, the design by contract programming discipline [Meyer97] stresses the need to precisely define the behaviour of modules through claims and responsibilities – the contracts. The specification of contracts – pre and post-conditions – for each method of a type is possible in several existing assertion languages – iContract [iContract], COLD-1 [Jonkers91], Jass [Bartezko01], Eiffel [Meyer97], ContractJava [Findler01], Larch family [Guttag85], JML [Leavens00] among them. Some of these – Jass, iContract, Eiffel – as well as others, allow the monitoring of contracts at runtime.

Specifying contracts is very important to the correct reuse of software. Clients must know the rules of the business. Thus, methods must make their pre and post-conditions public knowledge. Moreover, contract specifications are important insofar as they can be used to verify program properties [Havelund00, Jacobs01, Van der Berg01].

Cite this article as follows: Isabel Nunes: Design by Contract Using Meta-Assertions, in Journal of Object Technology, Vol. 1, No. 3, Special Edition: TOOLS USA 2002 proceedings, pages 37-56.
http://www.jot.fm/issues/issue_2002_08/article3

Testing contract assertions at run-time is also important because it is a way to ensure that methods are executed only if they are given the proper conditions, and to ensure that only correct implementations of specifications are executed.

The several languages of assertions, and monitoring code generation tools that exist allow the specification and, eventually, the runtime checking of very powerful and elegant contracts. This is definitely so for classes as simple as Stack, Point or Account. However, the task of specifying contracts for methods in classes that are clients of these simple classes, is harder and can bring undesirable effects like the increasing in class coupling and encapsulation decreasing. The benefits we gain from writing monitorable assertions that do not suffer from these defects can turn to be considered as not enough rewarding when compared with the effort we must put in that task.

We propose a general responsibility assignment pattern for design by contract that is to be used in the writing of assertions while avoiding the above mentioned undesirable effects. This way of *doing* design by contract demands for additional expressive power from assertion languages. We also propose a general extension for assertion languages while maintaining the semantics of simple assertions and reusing monitoring code generation tools that eventually exist for those languages.

The paper consists of five sections. In the next section we show, through the use of an example, how contracts should and should not be written if one aims at low class coupling and encapsulation of object components. The approaches that existing assertion languages allow to follow in the specification of this kind of assertions are not satisfying in what respects several criteria. Section 3 presents our approach – meta-assertions – in an informal way, stressing its benefits from several points of view. It also gives the formal syntax of meta-assertions and the rules that define its operational semantics. Section 4 presents the rules for the expansion of meta-assertions into simple assertions abstracting away the details of the assertion language that serves as the basis for meta-assertions. Section 5 presents the conclusions and further work.

2 MOTIVATION

Let us take a first example to show the reasons why we are compelled to write assertions in a given way, and the reasons why it is not the best way to write them. This example deals with points, polygons (whose vertices are points) and drawings (which are composed of polygons). Each one of these types defines an operation of movement by given distances both horizontally (dh) and vertically (dv).

The semantics of these operations are given, in a rigorous way, through axioms in the abstract data types (ADTs) that define types `Point`, `Polygon` and `Drawing`.

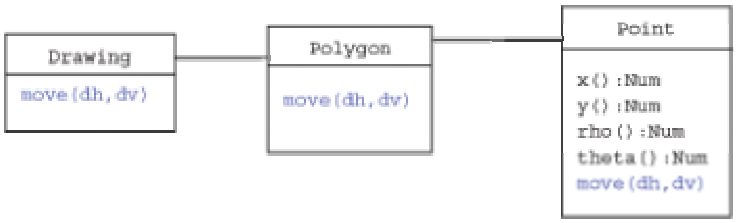


Fig. 1: Polygon is a client of Point and a supplier for Drawing

In order to build classes as correct implementations of the corresponding ADTs we have to implement each method of the classes in such a way that the axioms are true for each and every possible instance of the class. In what concerns the `move` operations, the relevant parts of these ADTs can be given by:

ADT Point
Operations
x: Point → num **y**: Point → num **move**: Point num num →Point
Axioms (forall p:Point, dh,dv:num)
x(move(p,dh,dv)) = x(p) + dh y(move(p,dh,dv)) = y(p) + dv

ADT Polygon
Operations
new: Point Point Point→Polygon **new**: Polygon Point→Polygon
move: Polygon num num→Polygon **vertex**: Polygon num→Point
vertices: Polygon→num
Axioms (forall p:Polygon; dh,dv,i:num; v, v_1, v_2, v_3:Point)
vertex(move(p,dh,dv),i) = move(vertex(p,i),dh,dv)
vertices(new(v_1,v_2,v_3))=3 vertices(new(p,v))=vertices(p)+1
vertex(new(v_1,v_2,v_3),i)=v_j i∈[1..3]
vertex(new(p,v),i)=if i=vertices(p)+1 then v else vertex(p,i)
Pre-conditions (forall p:Polygon; i:num)
vertex(p,i) requires i∈[1..vertices(p)]

ADT Drawing
Operations
new: → Drawing **new**: Drawing Polygon → Drawing
move: Drawing num num → Drawing
poly: Drawing num → Polygon **polies**: Drawing → num

```
Axioms (forall d:Drawing; dh,dv,i:num; p:Polygon)
  poly(move(d,dh,dv),i) = move(poly(d,i),dh,dv)
  polies(new()) = 0;          polies(new(d,p)) = polies(d) + 1
  poly(new(d,p),i) = if i=polies(d)+1 then p else poly(d,i)
Pre-conditions (forall d:Drawing; i:num)
  poly(d,i) requires i∈[1..polies(d)]
```

In order to create the types that implement these ADTs we shall define the assertions that specify their behaviour – the contracts for their methods – from the ADT's axioms and pre-conditions. We will use in this example a general assertion language that extends the syntax of Java expressions with several given constructs (`forall` – a quantifier, and `old` – to refer to values before execution, are the ones we use in this paper).

From ADT Specifications to Contracts

When we try to establish the correspondence between the ADT specifications and class assertions we should, among other things, create a post-condition for each axiom that involves a command auxiliary function (ones that return an object of the type being defined, as for example, `move`). When we think about implementing ADT command functions, [Meyer97], we usually abandon the ADT applicative kind of specification, in which all operations are modeled as mathematical functions (function `move`, for example, returns a new `Point` that results from moving the original one). Instead, we adopt the more imperative style that prevails in software construction (where structures are modified instead of producing new ones). By this reason, it is usual to implement command functions as procedures, that is, methods that do not return any value.

The axiom for the `move` operation in ADT `Point` suggests that the point coordinates change after a movement and it shows how they change. We would easily obtain the post-condition of the `move` method in type `Point`,

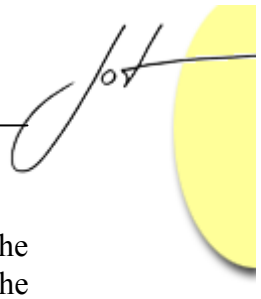
```
a)x()==old(x())+h && y()==old(y())+v
```

The object that results from moving the original point is the current object at the time the post-condition is evaluated (`x()` is implicitly applied to the current object); the `x()` coordinate of the original point object is given in a) by `old(x())`.

The meaning of the axioms for the `move` function in the ADT for `Polygon` can be expressed as the following post-condition for the `move` method in type `Polygon`,

```
b) forall int i in 1..vertices() |
                vertex(i).equals(old(vertex(i)).move(dh,dv))
```

The object that results from moving the original polygon is the current object at the time the post-condition is evaluated. So, `vertex(i)` is called over the moved polygon – the current object – and gives the moved vertex. As in the ADT's axiom, here we say that this moved vertex equals the vertex we would obtain if we moved the corresponding vertex of



the original polygon `(old(vertex(i))`. The same reasoning could be applied to the `move` operation of type `Drawing`. These post-conditions assume that the result of the `move` methods in types `Point` and `Polygon` are functions that return a point and a polygon respectively (they are compared with the existing vertices and polygons). This goes against the above suggested idea that all methods that change the state are procedures, and which is itself consistent with the need for the clear distinction between commands, which change objects but do not directly return results, and queries, which provide information about objects but do not change them.

Moreover, any expression *obj.meth(args)* that appears in an assertion, represents the value that is returned by the call of method *meth* over object *obj* with arguments *args*. If we recall that one of the important roles of assertions is to allow the monitoring of executions, we should look at these *obj.meth(args)* expressions as real method calls.

The assertion in b) goes against the reasonable rule that one should not use operations with side effects in the specification of contracts that are to be monitored (in a monitored call, the invocation of `move` over `old(poly(i))` would modify it. A classic example is the post-condition for the `push` operation on a `Stack`, `pop().equals(old(this))` that is obtained from the axiom pop(push(X,S))=S. The evaluation of this post-condition at the end of the method execution would change the current stack by popping it the element just pushed, leaving it as it was before the push operation was executed.

Assertions should be written using queries only, that is, its evaluation should be **without side effects**. Taking this as a rule from here onwards, let us see then how these post-conditions could be written.

Contracts Can Bring Undesired Class Coupling

In order to say – in `move` method of type `Polygon` – that all vertices of a polygon have suffered movement we have to use the queries that type `Point` offers:

```
c) forall int i in 1..vertices() |
          vertex(i).x() == old(this).vertex(i).x()+dh &&
          vertex(i).y() == old(this).vertex(i).y()+dv
```

This post-condition completely defines the changes that were operated in the state of the system. It is also the only way we can write it because type `Point` does not supply any other way to show its changes after a movement. Nevertheless this post-condition is more revealing than it should.

In order to say – in `move` method of type `Drawing` – that all polygons of a drawing have moved, we have to limit ourselves to the queries that type `Polygon` offers:

```
d) forall int i in 1..polies() |
     forall int j in 1..poly(i).vertices() |
          (poly(i).vertex(j).x() ==
```

```
                    old(this).poly(i).vertex(j).x()+dh)&&
(poly(i).vertex(j).y() ==
                    old(this).poly(i).vertex(j).y()+dv)
```

This post-condition is also more revealing than it should. As a consequence, the clients of this type must know about type `Point` and understand some of its methods in order to understand the result of applying the `move` method to a drawing composed of polygons! Clients of type `Drawing` shouldn't have to know about the exact changes in the polygon's coordinates (a set of polygons abstracts away the structured set of points that constitute the drawing). The encapsulation that is shown in figure 1 should be maintained at the level of assertions too.

This post-condition increases coupling between the classes of the system. As we know, strong coupling brings undesirable designs due to the decreasing in extension and reuse. We should be able to act over a drawing of polygons solely through the polygons themselves. The ideal way to do this would be something like:

e) `forall int i in 1..polies() | something_about_poly(i)_only`

that would reveal the changes operated in the drawing only through their most direct state revealing queries.

These examples show how a well-known problem that software designers deal with frequently, can emerge when we try to design by contract. This problem, and proposed solutions, is described by design pattern "Don't talk to strangers" [Larman98], which is related to "Chain of responsibility" [Gamma95], to be used during OO system design.

The pattern places constraints on what objects should be sent messages to within a method. It states that, within a method, messages should only be sent to the following objects: i) the current object; ii) a parameter of the method; iii) an attribute of the current object; iv) an element of a collection which is an attribute of the current object; v) an object created within the method.

The intent is to avoid coupling a client to knowledge of indirect objects and the internal representations of direct objects. Direct objects are a client's *familiars*, indirect objects are *strangers* and a client should only talk to familiars, not to strangers.

Applying these ideas to the design of the `Drawing`'s `move` method of our example would lead us to designing it as a call to the `move` method of each of its polygons (as was already suggested above in the ADT presentation). Likewise, `Polygon`'s `move` method would be designed as a call to the `move` method of each of its vertices. This is shown in UML collaboration diagram in figure 2. This would be consistent with the design class diagram of figure 1, which presents the desired low coupling.

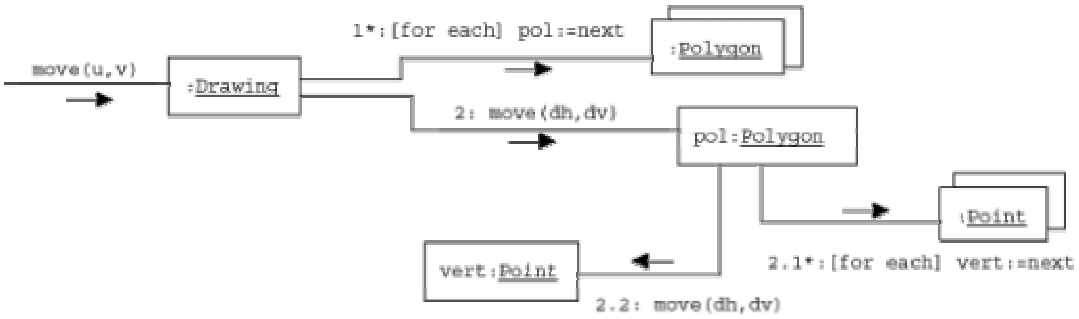


Fig. 2: UML colaboration diagram for the Drawing *move* operation

Our concern when designing by contract is the same, although in design by contract we deal with specifications that is, with assertions, not with prescriptions. So, the advice to follow here is "*don't talk **about** strangers*"! The familiars in this context are i) the current object, ii) the parameters of the method, and iii) all objects that are accessible through functions of the class. Furthermore we are constrained by the fact that assertions must have no side effects.

One possible approach would be to create several, and otherwise useless, methods that would reveal all about the class's internal objects and the internal objects of these ones or that would inform, in this case for example, whether a polygon had been moved for given distances. This has several drawbacks: i) the code in those methods – in the programming language in use – would be written manually, which brings an additional source of errors; ii) it may be the case that it is not possible to create any more methods in the supplier classes unless we extend it through inheritance. The additional effort that this approach requires can make design by contract unattractive.

Another approach could be to write contracts that refer to the contracts of other methods allowing to say, for example, that the *result* of moving a polygon is the same as the *result* of moving all its vertices. We see two possible ways to reach this goal.

One way to do this could be to create, for each method *m* in the supplier classes, two other methods that would evaluate the pre and the post-conditions of *m*. The post-condition in e), for example, would call the *post-move* method applied to all the polygons of the drawing. This has the same i) and ii) above described drawbacks plus: iii) in what concerns the methods written to evaluate post-conditions, the programmer would have to be able to compare the new state with the *old* state of an object, for a command that had not been executed (only its post-condition would be evaluated)... this is not a trivial thing.

The approach we advocate here allows to write assertions that talk about assertions of other methods without having to manually write any additional code. Furthermore, in order to be possible to monitor contracts at runtime, we propose a process of generation of (simple) assertions in some existing assertion language from these (meta) assertions, that can be automated. The assertions that will ultimately be monitored are assertions written in some existing assertion language. So, the automatic generation of code for monitoring, provided by existing tools, is fully reused. All the effort in designing by contract is put on the writing of assertions, not on coding.

3 META-ASSERTIONS – A PROPOSAL

The approach we advocate asks for an expressiveness that none of the assertion languages that we know possesses. Our proposal is a new construct, a kind of a meta-construct, for assertion languages that allows to refer to assertions of other methods.

This approach brings several enhancements to existing assertion languages and tools: i) it allows the writing of very simple and easily understandable assertions; ii) it helps keeping class coupling low; iii) it promotes encapsulation; iv) it eases the job of contract writers, of method implementors, and of client classes implementors. A discussion on the benefits that they bring to the several entities involved in software specification and implementation can be found in [Nunes02].

Informal Syntax and Semantics

The new constructs 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.

Let us return to the `move` operation for the `Drawing` type. With the proposed approach we would write this post-condition as:

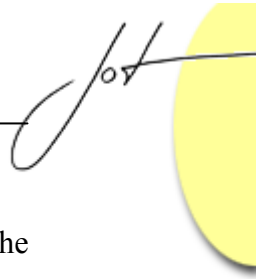
```
f) forall int i in 1..polies() | poly(i).move(dh,dv)»post
```

which intended meaning is: after the execution of the command `move` applied to an object of type `Drawing`, the state is the same that results from applying the `move` operation to all its polygons. In other words, 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 the post-conditions of those methods.

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

When Monitoring Enters the Scene

How can meta assertions be monitored, that is, how can code be generated from them that can be executed before (pre-conditions) and after (post-conditions) the method code itself? Meta assertions by themselves cannot be evaluated by existing tools. They denote other assertions that, in turn, may denote other assertions. In order to evaluate a given meta assertion by an existing tool, we have to expand it until it is composed of simple assertions only. Informally, simple assertions are assertions that do not contain any of the



»pre and »post meta constructs. For example, the (meta) post-condition of the Polygon's move method would expand to the (simple) post-condition:

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

Likewise, the (meta) post-condition of the Drawing's move method would expand to the (simple) post-condition:

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

Even if the assertions that are finally monitored are the ones that refer to second (and possibly lower) level information, the extra coupling that they bring do not imply the costs that are usually associated to high coupling. Let us see how.

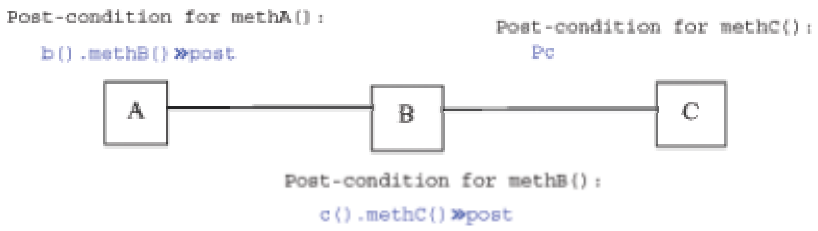


Fig. 3

When, in figure 3, the simple post-condition is generated from the meta post-condition of methA() in class *A*, an expression is obtained that refers to objects of types *B* and *C*: b().c().Pc. Usually, higher coupling brings more difficult extension, but this is not the case here: if the post-condition of methC() changes, the (meta) post-conditions of methB() and methA() do not need to change. Only the corresponding simple post-conditions must change; if the process of expanding meta assertions is automated, this is easily done automatically by recompiling types *B* and *A* in order for these simple post-conditions to be re-generated.

Thus, there is total encapsulation in what meta assertions are concerned, and almost total encapsulation in what the corresponding expanded assertions are concerned (depending on the re-compilation to generate the new simple assertions).

In order to check (meta) contracts at runtime, we depart from:

- i) a set of classes written in an existing OO programming language PL; ii) the (meta) contracts for those classes written in an assertion language MAL which is an existing assertion language AL (for PL) extended with meta-constructs;

and we want to:

- iii) generate the simple assertions in the assertion language AL as expansions of the original meta-assertions; iv) use the tools for monitoring code generation that already exist for the pair (PL,AL).

The idea is that all syntactic and semantic checking of simple assertions are done by the existing tool for AL. This can be so because we prove that meta-assertion expansion process preserves grammatical correctness and semantics.

Syntax of Meta-Assertions

As a first step we must specify the syntax of the meta-assertion language MAL as an extension of a base language AL. The syntactic notation we use is based on BNF. We list the various syntactic categories and give meta-variables that will be used to range over constructs of each category:

a will range over (simple) assertions, **Assn**

ma will range over meta-assertions, **MAssn**

bm will range over basic-metas, **BMeta**

mp will range over meta-paths, **MPath**

p will range over paths, **Path**

memb will range over members, **Memb**

mc will range over method calls, **MC**

atc will range over attribute calls, **AttC**

ref will range over references to objects, **RObj**

exp will range over expressions, **Exp**

The assertions of MAL have the same structure as those of AL with the difference that the ones in MAL are the ones in AL augmented with elements of BMeta, that is, basic-metas. We assume that the structure of assertions – method calls, attribute calls, references to objects and expressions – is given elsewhere by the syntax of the assertion language AL that serves as the basis for MAL. This assertion language eventually depends on the programming language for which it is designed. We define the other categories in a way that is independent of the details of the chosen assertion language. The structure of the other constructs is:

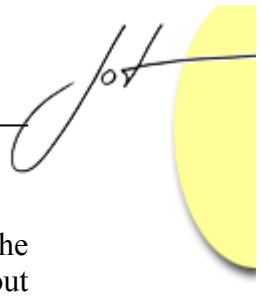
bm ::= *mp*»`pre` | *mp*»`post`

mp ::= `mc` | *applyP* (*p*,`mc`)

p ::= `ref` | *memb* | *applyP* (*p*,*memb*)

memb ::= `atc` | `mc`

The function *applyP: Path* × *Memb* → *Path* is used to define paths in a way that is independent of the details of AL. If, for example, the assertion language in question were iContract or Eiffel, the result of *applyP(p,memb)* would be *p.memb* because that is the way how application of methods and attributes is done in those languages. If it were an assertion language based on Smalltalk the result would be *p memb*. We also consider that the category RObj of the assertion language has a special element denoting the current object (`Current` in Eiffel, `this` in Java, `self` in Smalltalk) and which we represent by `cur()`. We call *target method* the method corresponding to `mc` in a basic-meta.



In order to evaluate or to expand a given meta-assertion *ma* that is part of the contract for a method *mth* in a type *ty*, we need to be able to access information about some *ty* supplier classes: to evaluate basic-meta *applyP(p,*`mc`*)* `»pre`, for example, we need to access information about the members of *p*'s type.

Consider the following sets of identifiers and corresponding meta-variables that will be used to range over their elements. These sets will constitute the context for the evaluation of meta-assertions and, later, for the process of expansion.

TYPEID – set of type identifiers ranged over by *ty*

ATTRID – TYPEID-indexed set of attribute identifiers ranged over by *attr*

METHID – TYPEID-indexed set of method identifiers ranged over by *mth*

PARAMID – METHID-indexed set of parameter identifiers ranged over by *par*

PRESTID – {pre, post} ranged over by *prest*

The following functions abstract away the details of the assertion and programming languages in which meta-assertions are based, allowing to represent information about types and assertions in terms of the sets of identifiers and of the syntactic constructs defined above. The way they are implemented obviously depends on the details of each assertion and programming languages.

MembList: TYPEID → Pow(METHID ∪ ATTRID)

Params: TYPEID×METHID → Pow(PARAMID)

TypeOfMeth: TYPEID × METHID → TYPEID

TypeOfAttr: TYPEID × ATTRID → TYPEID

TypeOfRef: TYPEID×METHID×RObj→ TYPEID

Meth: TYPEID × MC → METHID

Attr: TYPEID × AttC → ATTRID

Cond: TYPEID×METHID×PRESTID → MAssn

BasicM: MAssn → Pow (BMeta)

ActualPar: MPath → Pow (Exp)

Members: Pow (METHID ∪ ATTRID) × MAssn → Pow (MC ∪ AttC)

The *MembList* function gives the method and attribute identifiers that are defined in a given type – its members. The *Params* function gives the formal parameters of a given method in a type. The functions *TypeOf*... give the identifier of the type to which the given method, attribute or object reference belongs. In the *TypeOfRef* function the third parameter – reference to object – can be a logical variable of the assertion language (for example in a `forall` construct) or it can be a formal parameter; that is why an extra argument is needed – the identifier of the method containing the assertion – in order to know the type of the reference.

The *Meth* and *Attr* functions give the method and attribute identifier, respectively, that correspond to a given method or attribute call.

The *Cond* function gives the meta-assertion that is the pre or post-condition – depending on its third argument – of a given method on a given type. We assume that the pre- and post-conditions supplied by this function are the ones for the given type and method. That is, it is not supposed *oring* pre-conditions with ancestor pre-conditions to

obtain the pre-condition referred to by the meta-assertion, nor *anding* post-conditions with ancestor post-conditions to obtain the post-condition referred to by the meta-assertion. Its implementation obviously depends on the base assertion language. If this language supposes, like iContract and Eiffel, that, for example, the pre-condition written in a method is the result of *oring* it with its ancestors' pre-conditions, then function *Cond* has to be implemented in a way that returns this complete pre-condition. On the contrary, if the base assertion language supposes, like Jass and ContractJava, that the pre-condition written in a method is as it is (they generate code that check the hierarchical dependencies of assertions at runtime), then the *Cond* function has to result accordingly.

The *BasicM* function gives all the basic-metas that appear in a given meta-assertion. The *Members* function gives the method and attribute calls that appear in a meta-assertion that correspond to given method and attribute identifiers. Finally, the *ActualPar* function gives the expressions that constitute the actual parameters in the method call of a meta-path.

The following 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.

[Type1]:

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

[Type2]:

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

[Type3]:

$$\frac{}{ty,mth\ \hat{\imath}\ ref \longrightarrow_{type} TypeOfRef(ty,mth,ref)}$$

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.

Semantics of Meta-assertions

Next we present the semantics of MAL assuming a language AL as the base assertion language. We only present the rules for the operational semantics of basic-metas. We do not consider the evaluation of basic-metas which target method is not a command or procedure, that is, which target method is a function that returns a result, for obvious reasons. We consider that the basic-meta `mc»prest` is semantically equivalent to *applyP(`cur()`,mc)»`prest`*.

The semantics is given operationally through a set of rules (fig. 4) that, given a configuration that includes, among other elements, a meta-assertion and a state, gives a boolean value. The way rules are expressed is similar to the one adopted in [Winskel92].

[basicM1]:

$$\frac{\begin{array}{ll} \text{ActualPar}(mc)=\{exp_0 \ldots exp_m\} & \text{Params}(ty',mth')=\{par_0 \ldots par_m\} \\ o=\text{Object}(s(old),\texttt{cur()},p) & y=\text{Object}(s(young),\texttt{cur()},p) \qquad ma=\text{Cond}(ty',mth',prest)\ [exp_j/par_j]\ \text{ for } j\in[0,m] \\ (ty', mth', prest)\notin Lm & \langle ty',mth',ma,s[\ o\rightarrow s(old)[\texttt{cur()}],y\rightarrow s(young)[\texttt{cur()}]],Lm\cup\{(ty',mth',prest)\}\rangle \rightarrow \text{bool} \end{array}}{\langle ty,mth,\ \text{applyP}(p,mc)\gg prest,s,Lm\rangle \rightarrow \text{bool}}$$

where *ty′* is the static type of *p*, given by $ty,mth \hat{1}\ p \longrightarrow_{type} ty'$ and mth' is the method id given by *mth′ = Meth(ty′, mc)*

[basicM2]:

$$\frac{(ty', mth', prest)\in Lm}{\langle ty,mth,\ \text{applyP}(p,mc)\gg prest,s,Lm\rangle \rightarrow \ \perp}$$

where *ty′* is the static type of *p*, given by $ty,mth \hat{1}\ p \longrightarrow_{type} ty'$ and mth' is the method id given by *mth′ = Meth(ty′, mc)*

[applyMPath]:

$$\frac{\begin{array}{ll} \text{MembList}(ty')=MList & \text{Members}(MList,ma)=\{memb_0 \ldots memb_n\} \\ o=\text{Object}(s(old),\texttt{cur()},mp) & ma'=ma[mp/\texttt{cur()}][\text{applyP}(mp,memb_j)/memb_j]\ \text{for } j\in[0,n] \\ y=\text{Object}(s(young),\texttt{cur()},mp) & \langle ty,mth,ma,s[\ o\rightarrow s(old)[\texttt{cur()}],y\rightarrow s(young)[\texttt{cur()}]],Lm\rangle \rightarrow \text{bool} \end{array}}{\langle ty,mth,\ ma',s,Lm\rangle \rightarrow \text{bool}}$$

where *ty′* is the type of *mp*

Fig. 4: Operational semantics for meta-assertions in MAL

Remember that the only command that is executed and that can change the state is the original one – the one that contains the original meta-assertion in its contract. Throughout the evaluation of the meta-assertions that compose the original meta-assertion, no more commands are executed – only queries are executed and these do not change the state.

So, the objects that are of interest, in what a (non-constructor) method is concerned, are i) the current object before the method has been executed and the same current object after the method has been executed; ii) the objects referred to by the actual parameters before and after method execution, insofar as they may be modified.

The objects that the method may create are of two kinds: i) local entities that are not interesting in what concerns the evaluation of the method contract; ii) other objects that happen to be components of the current object or of the objects referred to by the parameters. These latter objects are already included in the objects that we referred above as being the interesting ones.

The need for the old versions of the potentially modifiable objects has to do with the `old` construct that typically all assertion languages define. This construct changes the object to which is applied its operand path – in `old` *p*, *p* is applied to the object as it was before the execution of the original command method.

A state *s* is a pair <*s(old),s(young)*> where *s(old)*, respectively *s(young)*, represents the part of state *s* that contains information about the old, respectively young, versions of objects. These two elements of a state are mappings from objects to type-values pairs <*dynamic_type,attribute_values*>. We denote by *s(v)[ref]* the type-values pair corresponding to object *ref* in its *v* version. For example, *s(old)*`[cur()]` is the type-values pair that gives the dynamic type and the values of attributes of the current object in its old version.

We denote by *Object(s,ref,p)* the type-values pair <*dynamic_type,attribute_values*> that represents the object that results from applying path *p* to object *ref* in state *s*. We denote by *s[val→ref]* the state that is equal to *s* except in the value of *ref* which is equal to *val*.

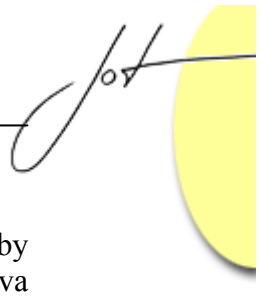
If the original assertion is a pre-condition, the old part of the state is irrelevant, because pre-conditions are evaluated before method execution. If the assertion is a post-condition then both these parts are important and are eventually not equal.

A configuration is a tuple <*ty,mth,ma,s,Lm*> where $ty \in$TYPEID, $mth \in$METHID, $ma \in$MAssn, *s* is a state and *Lm* is a set of triples (ty_n,mth_n,$prest_n$) where $ty_n \in$TYPEID, $mth_n \in$METHID and $prest_n \in$PRESTID. The initial configuration is <*ty,mth,ma,*s_0, ∅> where *ma* is a meta-assertion belonging to method *mth* in type *ty*, and that is to be evaluated in state s_0. State s_0 is such that s_0*(young)* contains information about the current object `cur()` and about all the objects referred to by the method parameters as they are at the time meta-assertion *ma* is evaluated. The old part of s_0, s_0*(old),* contains information about the old versions (that is, as they were before the method was executed) of the objects that are referred to in `old` constructs of assertion *ma*.

The rule [basicM1] says that the boolean value of a given basic-meta *applyP(p,mc)*`»prest` in a state *s* is the same as the result of evaluating the `prest` (pre or post) condition of method `mc` (with actual/formal parameters substitution) in another state. This other state is equal to *s* except in its old and young versions of the current object: these are the objects that result from evaluating path *p* over the old and young versions of the current object of *s*.

The evaluation of the basic-metas stops with an error (rule [basicM2]) if the original meta-assertion is circular that is, if in the process of evaluating basic-metas of assertions, a basic-meta is reached which target method has already appeared in the evaluation process. The control of circularity is done by keeping information about the basic-metas that are to be evaluated. This information is kept in a list of triples, *Lm*, composed of: i) the identifier of the target method of the basic-meta, ii) the type from which that method is a member and iii) the kind – pre or post – of the basic-meta. If there already exists a triple in the list for some of the basic-metas that must be evaluated, then the meta-assertion is circular and its evaluation is not possible by rule [basicM2].

Rule [applyMPath] says that an assertion *ma* in which we syntactically substitute *applyP(mp,memb)* for all its members *memb*, evaluates in a given state *s* to the same value as the given assertion *ma* when evaluated in a state where the current object is the



object given by *mp*. We denote by *ma[p'/p]* the assertion that is obtained from *ma* by syntactically substituting path *p'* for all occurrences of path *p*. As an example, with Java as the base programming language,

<ty,mth,`x()`,s',Lm> ⟶ bool implies < ty,mth,`vertex(i).x()`,s,Lm> ⟶ bool

where *s'* is identical to *s* except in the old and young versions of the current object. The old, respectively young, version is the pair composed of the dynamic type of the object returned by the `vertex(i)` applied to the old, respectively young, version of `cur()` in *s*, and the values of the attributes of that same object. That is,

s' = s [Object(s(old),`cur()`,`vertex(i)`) →s(old)[`cur()`],

Object(s(young),`cur()`,`vertex(i)`) →s(young)[`cur()`]]

The following example, where we take for the base assertion language AL the Eiffel language, will help understanding the rules:

```
class CARGOFLEET feature
   ship:SHIP is do ... end
   fleet:FLEET is do ... end
   putBox(b1:BOX) is
   do ...
   ensure
         (ship.putBox(b1)»post)
         ((ship.full)implies (fleet.addShip(ship)»post))
   end
end -- class CARGOFLEET

class FLEET feature
   ...
end -- class FLEET

class SHIP feature
   numberBoxes:INTEGER is do ... end
   box(i:INTEGER):BOX is do ... end
   full:BOOLEAN is do ... end
   putBox(boxToPut:BOX) is
   do ...
   ensure
         (old full) implies numberBoxes=1
         (not old full) implies
                        (numberBoxes=old numberBoxes+1)
        equal(box(numberBoxes),boxToPut)
   end
end -- class SHIP
```

Suppose we want to evaluate the first basic-meta of the post-condition of the `putBox` method of class `CARGOFLEET` which is `(ship.putBox(b1)»post)`.

<CARGOFLEET, putBox, applyP(ship,putBox(b1))»post, s, Lm> ⟶ ?

where *s* has information about the instance of class `CARGOFLEET` to which was applied the method `putBox(b1)` and also about object `b1` of type `BOX`. The result of this evaluation, by rule [basicM1], is the same as we get by evaluating the meta-assertion (notice the substitution of parameters):

```
(old full) implies numberBoxes=1
(not old full) implies (numberBoxes=old numberBoxes+1)
equal(box(numberBoxes),b1)
```

in a state where the current object is not a `CARGOFLEET` object as it was before but, instead, the `SHIP` object that results from applying the `ship` method to that former current object.

4 EXPANSION OF META-ASSERTIONS

If contracts are to be checked at runtime, meta-assertions must be expanded so that the monitoring code generation tool that is to be used can generate runtime checking code from simple assertions in the base assertion language. We propose a process of expansion that abstracts away from the details of the base assertion and programming languages. This is achieved by using the functions and sets of identifiers defined in the previous section for the semantics of meta-assertions.

The rules that define the expansion of a meta-assertion into a simple assertion are presented in figure 5. 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 (if for example *ma* is of the form *bm1*`and`*bm2* then the simple assertion that results from its expansion is also of the form *a1*`and`*a2* where *a1* and *a2* are the expansions of *bm1* and *bm2*, respectively).

[Expand1]:

$$\frac{\text{BasicM}(ma)=\{bm_0 \ldots bm_n\} \quad \text{where } bm_j = \text{applyP}(p_j,mc_j)\text{»}prest_j \quad \text{for } j\in[0,n] \qquad ty,mth\ \hat{\imath}\ p_j \rightarrow_{type} ty_j \quad \text{Meth}(ty_j, mc_j)= mth_j \quad (ty_j, mth_j, prest_j)\notin Lm \text{ for all } j\in[0,n] \qquad ty_j\ mth_j\ \hat{\imath}\ bm_j \rightarrow_{lowM} ma_j \quad Lm\cup\{(ty_j, mth_j, prest_j)\},ty,mth\ \hat{\imath}\ ma_j \rightarrow_E a_j}{Lm,ty,mth\ \hat{\imath}\ ma \rightarrow_E ma[a_0/bm_0 \ldots a_n/bm_n]}$$

[Expand2]:

$$\frac{\text{BasicM}(ma)=\{bm_0 \ldots bm_n\} \quad \text{where } bm_j = \text{applyP}(p_j,mc_j)\text{»}prest_j \quad \text{for } j\in[0,n] \qquad ty,mth\ \hat{\imath}\ p_j \rightarrow_{type} ty_j \quad \text{Meth}(ty_j, mc_j)= mth_j \quad (ty_j, mth_j, prest_j)\in Lm \text{ for some } j\in[0,n]}{Lm,ty,mth\ \hat{\imath}\ ma \rightarrow_E \bot}$$

[LowerMeta]:

$$\frac{\text{ActualPar}(mc)=\{exp_0 \ldots exp_m\} \quad \text{Params}(ty',mth')=\{par_0 \ldots par_m\} \quad \text{Cond}(ty',mth',prest)=ma \qquad ma'=ma[exp_j/par_j] \text{ for } j\in[0,m] \quad \text{MembList}(ty')=MList \quad \text{Members}(MList,ma')=\{mc_0 \ldots mc_n\ atc_0 \ldots atc_l\}}{ty'\ mth'\ \hat{\imath}\ \text{applyP}(p,mc)\text{»}prest \rightarrow_{lowM} ma'\ [p/\texttt{cur()}]\ [\text{applyP}(p,mc_j)/mc_j] \text{ for } j\in[0,n]\ [\text{applyP}(p,atc_j)/atc_j] \text{ for } j\in[0,l]}$$

[CurPath]:

$$\frac{\hat{\imath}\ \text{applyP}(\texttt{cur()},mc)\text{»}prest \rightarrow_E a}{\hat{\imath}\ mc\text{»}prest \rightarrow_E a}$$

Fig. 5: Rules for the expansion of meta-assertions

For a given meta-assertion *ma*, the [Expand1] rule gives a meta-assertion that is equal to *ma* with all its basic-metas expanded. The substitution $ma[a_0/bm_0 \ldots a_n/bm_n]$ takes into account the renaming of logical variables (for example when there are two quantifiers that use the same logical control variable). These *Expand* rules prevent circular meta-assertions, as was done in rules of fig. 4, by keeping information about the basic-metas that are to be expanded. If there already exists a triple in the list for some of the basic-metas that must be expanded, then the meta-assertion is circular and its expansion is not possible by the [Expand2] rule.

In the [Expand1] rule several applications of the [LowerMeta] rule are needed in order to obtain the meta-assertions that result from the basic-metas that have to be expanded. These meta-assertions have to be expanded because they may themselves contain basic-metas.

The meta-assertion that results from the application of the [LowerMeta] rule over a basic-meta *applyP*(*p*,`mc`)»`prest` (which we call *generating* basic-meta) is obtained by taking the post or pre-condition – depending on `prest` – of its target method, say *Cond*,

and transforming it. This transformation is done by substituting in *Cond* all actual parameters for formal parameters and then applying the path *p* to all the method and attribute calls – the members – that appear in this modified *Cond*. Moreover, all references to `cur()` change to *p*.

In the `CARGOFLEET` example above, when expanding the meta-assertion of the `putBox` method of class `CARGOFLEET`, we would obtain the basic-metas: `ship.putBox(b1)»post` and `fleet.addShip(ship)»post`.

When applying the [LowerMeta] rule to the first one, the post-condition of the `putBox` method of class `SHIP` would be taken and transformed as mentioned, that is, the following meta-assertion would be obtained:

```
(old ship.full) implies ship.numberBoxes=1 and
(not old ship.full) implies
                    (ship.numberBoxes=old ship.numberBoxes+1)
and equal(ship.box(ship.numberBoxes),b1)
```

There was a substitution of actual parameter `b1` for the formal parameter `boxToPut`. All calls to members of class `SHIP` – the type of the path of the generating basic-meta – were applied that path, that is, `ship`. In this case the member calls are only method calls: `numberBoxes`, `full` and `box`.

The expansion of a meta-assertion preserves grammatical correctness, that is, given $ma \in$MAssn grammatically correct, the simple assertion $a \in$Assn in $\varnothing$,ty,mth î ma $\rightarrow_E$ a where *ma* appears in some of the assertions of method *mth*'s contract in type *ty*, is grammatically correct. The proof of this result appears in [Nunes02].

We further prove the soundness of the expansion by proving that, for all meta-assertion *ma* that appears in some of the assertions of method *mth*'s contract in type *ty*, all state *s* and boolean value *bool*, if *ma* evaluates to *bool* in state *s* so does its expansion; and if the evaluation diverges, so does the evaluation of *ma*'s expansion. Furthermore, if the evaluation of *ma* stops in error, so does its expansion.

Proposition (SOUNDNESS).

The expansion of a meta-assertion is sound with respect to the semantics, that is, given $ma \in$MAssn that appears in some of the assertions of method *mth*'s contract in type *ty*,

1. $\langle ty,mth,ma,s,Lm\rangle \longrightarrow bool$ *implies* $\langle ty,mth,a,s,Lm\rangle \longrightarrow bool$
2. $\langle ty,mth,ma,s,Lm\rangle$ *diverges implies* $\langle ty,mth,a,s,Lm\rangle$ *diverges*
3. $\langle ty,mth,ma,s,Lm\rangle \longrightarrow \perp$ *implies* Lm,ty,mth î $ma \longrightarrow_E \perp$

where the simple assertion $a \in$Assn is such that Lm,ty,mth î $ma \longrightarrow_E a$ that is, *a* is the expansion of *ma*. The proof of this result appears in [Nunes02].

5 CONCLUSIONS AND FURTHER WORK

We presented an approach to design by contract that promotes low coupling and encapsulation in contract assertions: applying the general responsibility assignment pattern for design by contract "Don't talk *about* strangers" when building pre and post-conditions. This concern revealed a lack of expressive power in existing assertion languages, which we tried to fill by introducing the concept of meta-assertion. These meta-assertions extend some base assertion language, and allow the writing of very simple yet very expressive assertions. If monitoring is a goal, these can be expanded into simple-assertions of the base assertion language and can be monitored by existing tools, while maintaining total encapsulation in what meta assertions are concerned, and almost total encapsulation in what the corresponding expanded assertions are concerned (depending on the re-compilation to generate the new simple assertions).

There are some aspects of contracts that were not studied in what meta-assertions are concerned, as for example, assertions that include the *super* or *Precursor* reference, frame conditions (for example *MOD* in COLD-1, *assignable* in JML, *changeonly* in Jass), and others, which should be resolved if we want the expansion of meta-assertions to be possible when applied to a real assertion language.

Inheritance was a concern in the semantics and in the expansion of meta-assertions insofar as the pre and post-conditions that are picked up to be evaluated in figure 4 and to be expanded in figure 5 result from function *Cond* which, as explained, returns the complete assertion of the method (that is, if the method is redefining one of its ascendants, its assertions already reflect ascendant assertions). However, the semantic rules and the expansion rules ignore the polymorphism of object references in the definition of the supplier assertions that are picked up to be evaluated or to be expanded. These are the pre and post-conditions of target methods of basic-metas concerning the static type of the path to which they are applied (see rules [Expand1] and [BasicM1]). We are studying the pros and cons (against other solutions) of this approach in the monitoring of polymorphic entities.

REFERENCES

[Bartezko01] D.Bartezko, C.Fischer, M.Moller and H.Wehrheim: Jass-Java with assertions, Workshop on RunTime Verification, 2001.

[Findler01] R.B.Findler and M.Felleisen, Contract Soundness for Object-Oriented Languages, OOPSLA 2001.

[Gamma95] E.Gamma, R.Helm, R.Johnson and J.Vlissides, Design Patterns, Addison-Wesley 1995.

[Guttag85] J.V.Guttag, J.J.Horning and J.M.Wing, The Larch Family of Specification Languages, IEEE Software, 2(5), pp.24-36, Sept. 1985.

[Havelund00] K. Havelund and T. Pressburger, Model Checking Java Programs Using Java PathFinder, International Journal on Software Tools for Technology Transfer, STTT, 2(4) April 2000.

[iContract] iContract HomePage. http://www.reliable-systems.com/tools/iContract/iContract.htm.

[Jacobs01] B.Jacobs and E.Poll, A Logic for the Java Modeling Language JML, in: H. Hussmann (ed.), Fundamental Approaches to Software Engineering (FASE), Springer LNCS 2029, pp.284-299, 2001.

[Jonkers91] H. B. M. Jonkers, Upgrading the Pre- and Postcondition Technique, VDM Europe (1), pp.428-456, 1991.

[Leavens00] 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.

[Larman98] C. Larman, Applying UML and Patterns, Prentice-Hall PTR, ISBN 0-13-748880-7, 1998.

[Meyer97] B.Meyer, Object-Oriented Software Construction, 2nd edition, , Prentice-Hall PTR, ISBN 0-13-629155-4, 1997.

[Nunes02] I.Nunes, Design by Contract Using Meta-Assertions, Technical Report DI/FCUL, TR-02-7. Dept. of Computer Science, Lisbon Univ. July 2002.

[Van der Berg01] J.Van der Berg and B.Jacobs, The LOOP compiler for Java and JML, T. Margaria and W. Yi (eds.), Tools and Algorithms for the Construction and Analysis of Software (TACAS), Springer LNCS 2031, pp.299—312, 2001.

[Winskel92] G.Winskel, The Formal Semantics of Programming Languages, MIT Press 1992.

About the author

Isabel Nunes is an Assistant Professor at the University of Lisbon, Portugal. Her interests are in the area of program specification and verification and OO modeling and programming. She is also interested in methods for teaching object oriented concepts. She can be reached at in@di.fc.ul.pt.