Design by Contract Using Meta-Assertions

Isabel Nunes

DI-FCUL TR–02–7

July 2002

Departamento de Informática
Faculdade de Ciências da Universidade de Lisboa
Campo Grande, 1700 Lisboa
Portugal

Technical reports are available at http://www.di.fc.ul.pt/tech-reports. The files are stored in PDF, with the report number as filename. Alternatively, reports are available by post from the above address.

Design by Contract Using Meta-Assertions

I.Nunes
Lisbon University, Faculty of Sciences
Informatics Department
Bloco C5, Piso1, Campo Grande
1749-016, Lisboa, Portugal
in@di.fc.ul.pt

ABSTRACT

The important role that class contracts – pre and post-conditions of methods, and invariants – play in the specification, monitoring and reuse of classes is becoming increasingly accepted by the OO community. 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, when the aim is the writing of pre and post-conditions for methods in classes that are clients of those simple classes, the task reveals itself harder and brings undesirable effects, like the increasing in class coupling and encapsulation decreasing. In addition, the conviction that assertions should have no side effects in order to be possible to monitor them, weakens the expressive power of assertion languages and makes it more difficult to avoid the above mentioned undesirable effects.

In this paper we propose a pattern to the design of class contracts that is an adaptation of existing patterns of design into a declarative context – the world of assertions. The use of this pattern produces 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 formally define their syntax and semantics. In order to be possible to check contracts at runtime, we define rules for the expansion of meta-assertions that can be monitored by existing tools, and we show grammatical and semantic soundness of the expansion.

Keywords

Design by contract; low coupling; encapsulation; meta-assertions; runtime contract monitoring; patterns.

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 [14] 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 [8], COLD-1 [10], Jass [2], Eiffel [14], ContractJava [4], Larch family [6], JML [12] among them. Some of these – Jass, iContract, Eiffel – as well as, for example, jContractor [11], Handshake [3], 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 [7, 9, 15].

Testing contract assertions at run-time is important because it is a way to ensure that methods are executed only if they are given the proper conditions, and also 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 six 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. Section 4 gives the formal syntax of meta-assertions and the rules that define its operational semantics.

Section 5 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. It also proves the soundness of the expansion with respect to grammatical correctness and semantics. Section 6 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.

2.1 Points, Polygons and Drawings

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*).

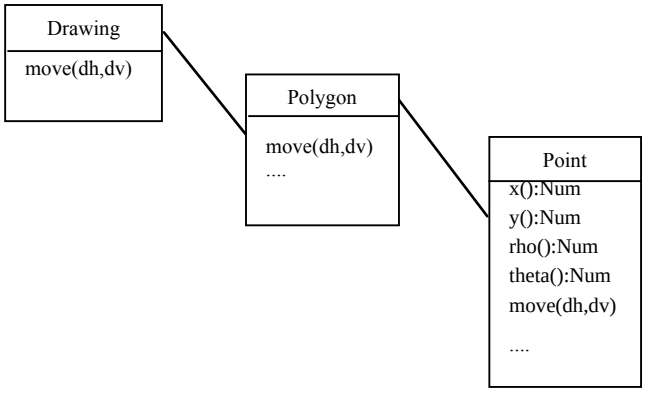

Figure 1

The semantics of these operations is given, in a rigorous way, through axioms in the abstract data types (ADTs) that define types `Point`, `Polygon` and `Drawing`. In order to build classes as correct implementations of these 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

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:num; v,v_1,v_2,v_3:Point; i∈[1..vertices(p)])

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),1) = v_1

vertex(new(v_1,v_2,v_3),2) = v_2

vertex(new(v_1,v_2,v_3),3) = v_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:num; p:Polygon; i∈[1..polies(d)])

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. These assertions will be useful both to the implementors of these types and for clients that will reuse them in the future.

We will use in this example a general assertion language that extends the syntax of Java expressions with several given constructs (`forall` – a quantifier, `old` – to refer to values before execution).

2.2 From ADT Specifications to Contracts

When we try to establish the correspondence between the ADT specifications and class assertions [14] 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, [14], 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`

because axiom `x(move(p,h,v))=x(p)+h` states that the `x()` coordinate of the object that results from moving `p` is equal to the `x()` coordinate of `p` plus `v`. In a) 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 same applies to `y()`.

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(this.vertex(i)).move(dh,dv))
```

In b) 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(this.vertex(i))`). The same reasoning could be applied to the `move` operation of type `Drawing`:

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

The assertions in b) and c) 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 advocated in [14] 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.

2.3 Assertions must be Side Effects Free

The assertions in b) and c) go 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(this.poly(i))` would modify it. If we had to refer to `old(this)` another time in this same post-condition we would not obtain the expected result).

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.

2.4 Contracts Bring Undesired Class Coupling

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

d)
```
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 in method `move` of type `Polygon` 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; clients of type `Polygon` shouldn't have to know about the exact changes in the vertices coordinates. The encapsulation that is shown in figure 1 should be maintained at the level of assertions too.

We are used to accept the solution in d) because `x()` and `y()` are of a primitive type – int – and thus the extra coupling between the `Polygon` type and that primitive type is irrelevant.

In type `Drawing` this problem is even worse, because of the higher level of abstraction it brings (a set of polygons abstracts away the (structured) sets of points that constitute the drawing).

e)
```
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)
```

The post-condition in this `move` method in type `Drawing` 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 `Polygon` does not supply any other way to show its changes after a movement.

This post-condition is very long and rather complicated, but the major drawback it brings is the increasing in the coupling between the classes of the system. As we know, strong coupling brings undesirable designs due to the decreasing in extension and reuse.

We are using here second level – `vertex(j)` – and third level – `x()` and `y()` – information. 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!

In what concerns coupling, the reference to `x()` and `y()` is, as above, not so bad because these are entities of a primitive type – there is no increase in coupling. But the coupling between the `Drawing` type and the type from which `vertex(j)` is an instance – type `Point` – is undesirable, because, as figure 1 shows, there is no need for this association. 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:

f) `forall int i in 1..vertices() | something_about_vertex(i)_only`
and

g) `@post forall int i in 1..polies() | something_about_poly(i)_only`

that would reveal the changes operated in the polygon and 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, together with proposed solutions, is described by design pattern "Don't talk to strangers" [13] which is related to "Chain of responsibility" [5] 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 (`Current`, `this`, `self`, etc); 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 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, the `Polygon 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.

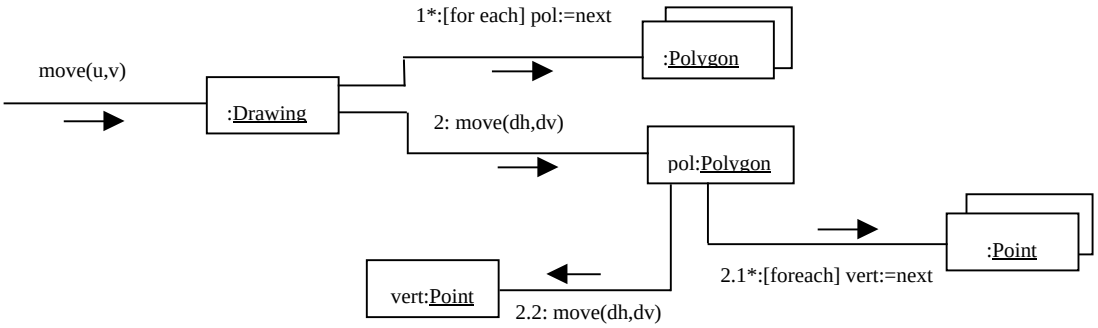

Figure 2

Our concern when designing by contract is the same – maintaining coupling low and all the benefits that it brings. But in design by contract we are dealing with specifications that is, with assertions, not with prescriptions. So, the advice to follow here is "don't talk *about* strangers"! Furthermore we are constrained by the fact that assertions must have no side effects, as motivated in section 2.3. 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.

One possible approach would be to 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. But, more manually written code means eventual additional errors. Furthermore, there are cases where we cannot add any more methods to a supplier class 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.

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 g), for example, would call the post-method applied to all the polygons of the drawing. 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; 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. 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:

– it allows the writing of very simple and easily understandable assertions;

– it helps keeping class coupling low;

– it promotes encapsulation;

– it eases the job of contract writers, of method implementors, and of client classes implementors.

Before presenting a formal syntax and semantics for meta-assertions, let us exemplify its use and discuss the benefits that they bring to the several entities involved in software specification and implementation.

3.1 Informal Syntax and Semantics

The new constructs are »`pre`, that is used to represent the pre-condition of the method to which it is applied and »`post`, that is used to represent the post-condition of the method to which it is applied.

Returning to the `move` operation for `Polygon` and `Drawing` types, we concluded above that we should be able to write that all vertices in a polygon would have been moved (f)) and all polygons in a drawing would have been moved (g)).

With the proposed approach we would write the post-condition in the `move` operation of type `Drawing` as:

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

which intended meaning is that 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.

Following the same reasoning the post-condition in the `move` operation of type `Polygon` would be:

i) `forall int i in 1..vertices() | vertex(i).move(dh,dv)»post`

with the same meaning as in h) above, but for polygon vertices instead of drawing polygons.

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 (a `Drawing` object must only talk about its constituting polygons, abstracting away the details that are encapsulated inside the concept of `Polygon`). We do this without creating unnecessary query methods for querying objects that are "strangers" to client classes.

3.2 A Look at Meta-Assertions

The way we look at a given method assertions depends on whether we are the implementors of that method or the implementors of client methods.

3.2.1 The Implementor's Point of View

Implementors wish for assertions that help them in code writing, that is, they wish for assertions that:

– leave no doubts about the resulting state of the method (post-conditions and invariants);

– assert what are the assumptions that can be made about the initial state (pre-conditions and invariants);

– give clues to the way things should be done (post-conditions).

– can be proved against implementations.

Meta assertions are useful to implementors:

– they promote more complete assertions because they allow the writing of conditions that would not otherwise be possible to write while maintaining low coupling;

– if a method `m()` in a type T has a pre-condition `ma().m'()»pre`, the implementor of `m()` knows that she can count on the truth of the pre-condition of method `m'()` in type T' (the type of object `ma()`);

– by allowing command methods to be specified using the specification of other commands, more clues can be given as to how to implement them without, however, being compromised with a specific implementation. For example, the post-condition `forall int i in 1..vertices() | vertex(i).move(dh,dv)»post` is much more helpful to an implementor than `forall int i in 1..vertices() | vertex(i).x() == old(this).vertex(i).x()+dh && vertex(i).y() == old(this).vertex(i).y()+dv`. If she implements the specification by calling the `move` method for all the polygon vertices, she is certain that she is implementing correctly.

– it is our intention to investigate formal proof mechanisms to allow for the proof that an implementation correctly implements a (meta) contract.

3.2.2 The Client's Point of View

Clients of a method wish for assertions that help them in their own code writing, that is, they wish assertions that:

– are easy to understand;

– clearly specify the obligations that they should accomplish in order to honour the method contract (pre-conditions and invariants);

– clearly specify the benefits that they will have if they sign the method contract (post-conditions and invariants);

Meta assertions are useful to clients:

– the specification of the benefits to clients can improve with meta assertions insofar as readability and completeness of post-conditions can be improved (as stated above).

– the clients of a method `m()` in type T should be able to ensure `m()`'s pre-condition before calling `m()`. The ideal way to do it would be for them to write the following code (for example, in Java) – `if (obj.m()»pre) obj.m() else ...` – where `obj` has type T. Later on we will propose a possible way into this.

3.2.3 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 move` method would expand to the (simple) post-condition:

j)
```
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 move` method would expand to the (simple) post-condition:

k)
```
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
```

This expansion implies the knowledge of other classes' assertions, and the application of those assertions to the objects to which the meta assertions are applied.

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.

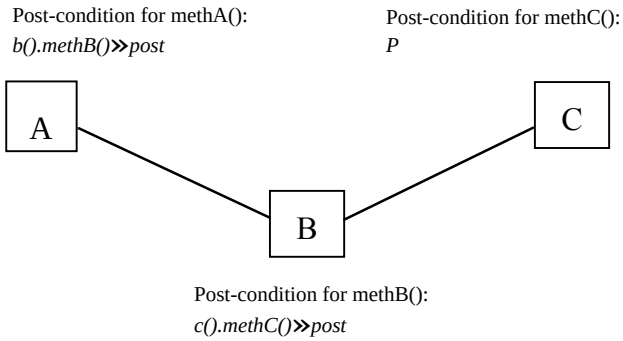

Figure 3

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().P`. 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 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. This can be so because we prove that meta-assertion expansion process preserves grammatical correctness and semantics.

4. META-ASSERTIONS – SYNTAX AND SEMANTICS

4.1 Syntax

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 a meta-variable 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 the assertion language. If, for example, the assertion language in question were iContract or the assertion part of 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, e.g. [1], 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 the method call 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)

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

ActualPar: MPath → Pow (Exp)

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* function gives the method identifier that corresponds to a given method call. The *Attr* function gives the attribute identifier that corresponds to a given 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 [8] and Eiffel [14], 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 [2] and ContractJava [4], 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]:

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

[Type2]:

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

[Type3]:

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

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.

4.2 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. The rules that define the semantics of other forms of meta-assertions are considered given.

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. The reason to this has to do with the fact that post-conditions of functions typically specify the result of the function (`return` in iContract, `Result` in Eiffel, Jass and JML, `$ret` in JMSAssert). A basic-meta represents an assertion – the pre or post-condition of its target method – not the execution of its target method. If it were possible to refer to post-conditions of functions in meta-assertions, we could obtain an assertion that would test the value of an entity – the result of the function – that wouldn't have been given any value. Moreover, this rule is 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.

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 (figure 4) that given a configuration that includes, among other elements, a meta-assertion and a state, gives a boolean value. The rules are expressed in a way that is similar to the one adopted in [16].

[basicM1]:

$$\frac{\begin{array}{lll} \text{ActualPar}(mc)=\{exp_0 \ldots exp_m\} & & \text{Params}(ty',mth')=\{par_0 \ldots par_m\} \\ o=\text{Object}(s(old),\texttt{cur()},p) & y=\text{Object}(s(young),\texttt{cur()},p) & ma=\text{Cond}(ty',mth',prest)\,[exp_j/par_j] \quad \text{for } j\in[0,m] \\ (ty',\,mth',\,prest)\notin Lm & \multicolumn{2}{l}{<ty',mth',ma,s[\ o\rightarrow s(old)[\texttt{cur()}],y\rightarrow s(young)[\texttt{cur()}]\ ],Lm\cup\{(ty',\,mth',\,prest)\}> \longrightarrow \text{bool}} \end{array}}{<ty,mth,\ \text{applyP}(p,mc)\text{»}prest,s,Lm> \longrightarrow \text{bool}}$$

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

[basicM2]:

$$\frac{(ty',\,mth',\,prest)\in Lm}{<ty,mth,\ \text{applyP}(p,mc)\text{»}prest,s,Lm> \longrightarrow \bot}$$

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

[applyMPath]:

$$\frac{\begin{array}{lll} \text{MembList}(ty')=\text{MList} & & \text{Members}(\text{MList},ma)=\{memb_0 \ldots memb_n\} \\ o=\text{Object}(s(old),\texttt{cur()},mp) & & y=\text{Object}(s(young),\texttt{cur()},mp) \\ \multicolumn{2}{l}{<ty,mth,ma,s[\ o\rightarrow s(old)[\texttt{cur()}],y\rightarrow s(young)[\texttt{cur()}]],Lm> \longrightarrow \text{bool}} & ma'=ma[mp/\texttt{cur()}][\text{applyP}(mp,memb_j)/memb_j] \text{ for } j\in[0,n] \end{array}}{<ty,mth,\ ma',s,Lm> \longrightarrow \text{bool}}$$

where *ty'* is the type of *mp*

Figure 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*. This obviously depends on the attribute and/or method calls in *p* that imply accessing attributes and attribute types, and/or executing methods that return objects with given dynamic types.

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 $<ty,mth,ma,s_0,\varnothing>$ 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 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,

$$<\text{ty,mth,}\texttt{x()}\text{,s',Lm}> \longrightarrow \text{bool implies} < \text{ty,mth,}\texttt{vertex(i).x()}\text{,s,Lm} > \longrightarrow \text{bool}$$

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,

$$\begin{aligned} s' = s\,[\ & \text{Object}(s(\text{old}),\texttt{cur()},\texttt{vertex(i)}) \rightarrow s(\text{old})[\texttt{cur()}], \\ & \text{Object}(s(\text{young}),\texttt{cur()},\texttt{vertex(i)}) \rightarrow s(\text{young})[\texttt{cur()}] \\ ]\ & \end{aligned}$$

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
```

```
 ...
 numberShips:INTEGER is do ... end
 ship(i:INTEGER):SHIP is do ... end
 ...
 addShip(s:SHIP) is
 do ...
 ensure
      (numberShips=old numberShips+1)
      equal(ship(numberShips),s)
 end
 ...
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 *ma* (notice the substitution of parameters)

```
(old full) implies numberBoxes=1 and
(not old full) implies (numberBoxes=old numberBoxes+1) and
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.

5. 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 all the functions and sets of identifiers defined in the previous section for the semantics of meta-assertions.

5.1 Expanding 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 in 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).

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.

The control of circularity is done, as before, by keeping information about the basic-metas that are to be expanded. This information is kept in a list of triples which composition is as in the semantic rules. 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 – remember the post-condition of the `move` method of class `Drawing`.

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` –

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

– 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*.

[Expand1]:

$$\frac{\begin{array}{lll} BasicM(ma)=\{bm_0 \dots bm_n\} & \text{where } bm_j = applyP(p_j,mc_j)\text{»}prest_j \quad \text{for } j\in[0,n] & \\ ty,mth \vdash p_j \longrightarrow_{type} ty_j & Meth(ty_j, mc_j)= mth_j & (ty_j, mth_j, prest_j)\notin Lm \quad \text{for all } j\in[0,n] \\ ty_j\ mth_j \vdash bm_j \longrightarrow_{lowM} ma_j & Lm\cup\{(ty_j, mth_j, prest_j)\},ty,mth \vdash ma_j \longrightarrow_E a_j & \end{array}}{Lm,ty,mth \vdash ma \longrightarrow_E ma[a_0/bm_0 \dots a_n/bm_n]}$$

[Expand2]:

$$\frac{\begin{array}{lll} BasicM(ma)=\{bm_0 \dots bm_n\} & & \text{where } bm_j = applyP(p_j,mc_j)\text{»}prest_j \quad \text{for } j\in[0,n] \\ ty,mth \vdash p_j \longrightarrow_{type} ty_j & Meth(ty_j, mc_j)= mth_j & (ty_j, mth_j, prest_j) \in Lm \text{ for some } j\in[0,n] \end{array}}{Lm,ty,mth \vdash ma \longrightarrow_E \bot}$$

[LowerMeta]:

$$\frac{\begin{array}{lll} ActualPar(mc)=\{exp_0 \dots exp_m\} & Params(ty',mth')= \{par_0 \dots par_m\} & Cond(ty',mth',prest) = ma \\ ma'=ma[exp_j/par_j] \text{ for } j\in[0,m] & MembList(ty')=MList & Members(MList,ma')= \{mc_0 \dots mc_n\ atc_0 \dots atc_l\} \end{array}}{\begin{array}{l} ty'\ mth' \vdash applyP(p,mc)\text{»}prest \longrightarrow_{lowM} ma'\ [p/\texttt{cur()}]\ [applyP(p,mc_j)/mc_j] \text{ for } j\in[0,n] \\ \qquad [applyP(p,atc_j)/atc_j] \text{ for } j\in[0,l] \end{array}}$$

[CurPath]:

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

Figure 5: Rules for the expansion of meta-assertions

We prove that the expansion preserves grammatical correctness. We do this under the following

ASSUMPTIONS:

a) The functions

Members: Pow (METHID $\cup$ ATTRID) $\times$ MAssn $\rightarrow$Pow (MC $\cup$ AttC)
ActualPar: MPath $\rightarrow$ Pow (Exp)
BasicM: MAssn $\rightarrow$ Pow (BMeta)
Params: TYPEID $\times$ METHID $\rightarrow$ Pow (PARAMID)
Cond: TYPEID $\times$ METHID $\times$ PRESTID $\rightarrow$ MAssn

preserve grammatical correctness.

b) The substitution of grammatically correct simple assertions for basic-metas in a meta-assertion preserves grammatical correctness.

c) The substitution of grammatically correct expressions for formal parameters in a meta-assertion preserves grammatical correctness.

d) Let *p* be a grammatically correct path and *memb* be a method or an attribute call. The substitution of *applyP(p,memb)* for *memb* in a meta assertion preserves grammatical correctness. The substitution of a grammatically correct path *p* for `cur()` preserves grammatical correctness.

PROPOSITION 1 (GRAMMATICAL CORRECTNESS).

Under the above assumptions, the expansion of a meta-assertion preserves grammatical correctness, that is, given $ma \in$ MAssn grammatically correct, the simple assertion $a \in$ Assn in

$$\varnothing,\text{ty},\text{mth} \vdash \text{ma} \longrightarrow_{\text{E}} \text{a}$$

where *ma* appears in some of the assertions of method *mth*'s contract in type *ty*, is grammatically correct.

PROOF.

The proof is by induction on the expansion rules. [Expand1] rule vacuously preserves grammatical correctness when BasicM(*ma*) is the empty set. In this case *a* is *ma*.

As induction hypothesis we have that

$$Lm \cup \{(ty_j, mth_j, prest_j)\},ty,mth \vdash ma_j \longrightarrow_E a_j \qquad for\ j \in [0,n]$$

preserves grammatical correctness where ma_j is such that $ty_j,mth_j \vdash bm_j \longrightarrow_{lowM} ma_j$ and BasicM(*ma*)={bm_0 ... bm_n}.

We have to prove that

$$Lm,ty,mth \vdash ma \longrightarrow_E ma[a_0/bm_0 \ldots a_n/bm_n]$$

preserves grammatical correctness.

There is a finite number of types, each with a finite number of methods. Therefore, there is a finite number of triples *(ty_j, mth_j, $prest_j$)* built from basic-metas `applyP`*(p,mc)*»`prest` in the way referred in [Expand1] rule. Each *(ty_j, mth_j, $prest_j$)* in a given list Lm_i refers to a basic-meta *bm* that will eventually generate other triples *(ty_k, mth_k, $prest_k$)* in Lm_{i+1} (if the expansion of the meta-assertion corresponding to *bm* contains basic-metas). Because the set of different triples built from basic-metas is finite, the sequence of Lm_is can never be infinite without repetition. Whenever a triple appears, that already exists in the list, the expansion process stops, by [Expand2] rule.

We know, from assumption b), that the substitution $a_0/bm_0 \ldots a_n/bm_n$ in *ma* preserves grammatical correctness provided that $a_0 \ldots a_n$ are grammatically correct.

The simple assertions $a_0 \ldots a_n$ are the result of expanding $m_0 \ldots m_n$ which are themselves the result of applying the [LowerMeta] rule to the basic-metas $applyP(p_j,mc_j)»prest_j$ in *ma* in this way:

$$ty_j,mth_j \vdash bm_j \longrightarrow_{lowM} ma_j$$

where ty_j and mth_j are the type identifier of p_j and the method identifier of mc_j, respectively.

We know, from assumption a), that the *BasicM* function preserves grammatical correctness, that is, all bm_j are grammatically correct (because *ma* also is). So, we have to prove that [LowerMeta] rule preserves grammatical correctness.

From assumptions c) and d) we know that the substitutions $applyP(p,mc_j)/mc_j$, $applyP(p,atc_j)/atc_j$, exp_j/par_j and $p/\texttt{cur()}$ preserve grammatical correctness provided that all exp_j and p are grammatically correct. From assumption a) we know that the *ActualPar* function preserves grammatical correctness, so all exp_j are grammatically correct (the generating basic-meta $applyP(p,mc)\text{»}prest$ is gramatically correct).

Because all bm_j, to which we apply [LowerMeta] rule, are grammatically correct from above, then the path p_j in each $bm_j = applyP(p_j,mc_j)\text{»}prest_j$ is grammatically correct and then p in rule *LowerMeta* is grammatically correct. To prove that

$$ma \qquad [exp_j/par_j]\ for\ j\in[0,m]\ [p/\texttt{cur()}]\ [applyP(p,mc_j)/mc_j]\ for\ j\in[0,n]$$
$$[applyP(p,atc_j)/atc_j]\ for\ j\in[0,l]$$

is grammatically correct we only have to prove that all mc_j and atc_j and par_j are grammatically correct. From assumption a) we know that i) function *Params* produces correct results and so par_j are grammatically correct and ii) function *Cond* produces correct meta-assertions. Again, by assumption a) we have that function *Members* preserves grammatical correctness and thus all mc_j and atc_j are grammatically correct. QED.

5.2 Soundness of the Expansion

We 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 2 (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*,

$1)\langle ty,mth,ma,s,Lm\rangle \longrightarrow bool\ implies\ \langle ty,mth,a,s,Lm\rangle \longrightarrow bool$

$2)\langle ty,mth,ma,s,Lm\rangle\ diverges\ implies\ \langle ty,mth,a,s,Lm\rangle\ diverges$

$3)\langle ty,mth,ma,s,Lm\rangle \longrightarrow \bot\ implies\ Lm,ty,mth \vdash ma \longrightarrow_E \bot$

where the simple assertion $a\in$Assn is such that

$$Lm,ty,mth \vdash ma \longrightarrow_E a$$

that is, *a* is the expansion of *ma*.

PROOF.

The proof is by induction on the structure of meta-assertions. If ***ma* is free from basic-metas** then, by the [Expand1] rule, its expansion is *ma*. Thus, 1), 2) and 3) are vacuously proved.

The induction hypothesis states that all meta-assertions $ma' <_E ma$ verify 1), 2) and 3) where the predecessor relation $<_E$ is such that $ma' <_E ma$ if ma' is a step ahead of ma in the evaluation process (that is, in order for ma to be evaluated, ma' must also be) and ma contains basic-metas.

Because ma must contain basic-metas in order to be related through $<_E$, we only have to prove that there aren't any infinite descending chains ... $ma_i <_E ... <_E ma_1 <_E ma_0$. We only have to worry here that the process of evaluating meta-assertions that are not free from basic-metas does not diverge.

There is a finite number of types, each with a finite number of methods. Therefore, there is a finite number of triples $(ty_j, mth_j, prest_j)$ built from basic-metas `applyP`(p,mc)»`prest` in the way referred in [basicM1] rule. Each $(ty_j, mth_j, prest_j)$ in a given list Lm_i refers to a basic-meta bm that will eventually generate other triples $(ty_k, mth_k, prest_k)$ in Lm_{i+1} (if the expansion of the meta-assertion corresponding to bm contains basic-metas). Because the set of different triples built from basic-metas is finite, the sequence of Lm_is can never be infinite without repetition. Whenever a triple appears, that already exists in the list, the evaluation process stops in error, by [basicM2] rule.

Thus, the relation $<_E$ is well-founded.

ma* is of the form *applyP(p,mc)»prest. In this case the condition $ma_0 = Cond(ty',mth',prest)\ [exp_j / par_j]$ *for* $j \in [0,m]$ is related with ma by the $<_E$ relation because it is one step ahead of ma in the evaluation process (see rule [basicMeta1]) and ma is not free from basic-metas. So, by the induction hypothesis we have that

$\langle ty',mth',ma_0, s[o \to s(old)[\texttt{cur()}],y \to s(young)[\texttt{cur()}]], Lm \cup \{(ty', mth', prest)\}\rangle \longrightarrow bool$
implies
$\langle ty',mth',a_0,s[o \to s(old)[\texttt{cur()}],y \to s(young)[\texttt{cur()}]], Lm \cup \{(ty', mth', prest)\}\rangle \longrightarrow bool$
where $Lm \cup \{(ty', mth', prest)\},ty,mth \vdash ma_0 \longrightarrow_E a_0$
and where
$ma_0 = Cond(ty',mth',prest)\ [exp_j / par_j]$ *for* $j \in [0,m]$.
Suppose we have
$\langle ty,mth,applyP(p,mc)\text{»}prest,s,Lm\rangle \longrightarrow bool$ (1)
We want to prove
$\langle ty,mth,a,s,Lm \rangle \longrightarrow bool$ (2) where $Lm,ty,mth \vdash applyP(p,mc)\text{»}prest \longrightarrow_E a$
If we have (1) then, by rule [basicM1], we have to have
$\langle ty',mth',ma_0, s[o \to s(old)[\texttt{cur()}],y \to s(young)[\texttt{cur()}]], Lm \cup \{(ty', mth', prest)\}\rangle \longrightarrow bool$ (3)
where
$ma_0 = Cond(ty',mth',prest)\ [exp_j / par_j]$ *for* $j \in [0,m]$.
From (3) and the induction hypothesis, we have
$\langle ty',mth',a_0,s[o \to s(old)[\texttt{cur()}],y \to s(young)[\texttt{cur()}]], Lm \cup \{(ty', mth', prest)\}\rangle \longrightarrow bool$ (4)
where a_0 is the expansion of ma_0 and o and y are as defined in [basicM1].
Meta-assertion ma_0 can be a simple-assertion or it may contain basic-metas.
If ma_0 is free from basic-metas, then its expansion, a_0, is equal to itself. Also, the expansion of *applyP(p,mc)»prest* is, as given by expansion rules [Expand1] and [LowerMeta], the simple assertion (we simplify method calls and attribute calls to members)
$Cond(ty',mth',prest)[exp_j / par_j]$ *for* $j \in [0,m]$
$[p / \texttt{cur()}][applyP(p,memb_j) / memb_j]$ *for* $j \in [0,n]$

which is

$a_0[p/\texttt{cur()}][applyP(p,mc_j)/mc_j]$ *for* $j\in[0,n]$

Call ma_0' to this simple assertion. So, we want to prove that

$<ty',mth',ma_0',s,Lm\cup\{(ty', mth', prest)\}> \longrightarrow bool$

which we obtain from (4) and the [applyMPath] rule.

Let us now go back and consider the case where ma_0 is not free from basic-metas. Then it can take the forms *applyP(p',mc')»prest'* or of some of the compound assertions (for example `and`, `or`, `not`, `forall` applied to one or two meta-assertions).

Let us take the form *applyP(p',mc')»prest'* for ma_0 (remember ma_0= *Cond(ty',mth',prest)* $[exp_j/par_j]$ *for* $j\in[0,m]$]). We have (4) where a_0 is the meta-assertion that is the expansion of ma_0. The expansion a of the original *applyP(p,mc)»prest*, as ruled by [Expand1] and [LowerMeta], is the expansion of the result of applying [LowerMeta] to its unique basic-meta which is itself. The [LowerMeta] rule applied to *applyP(p,mc)»prest* would give

Cond(ty',mth',prest)$[exp_j/par_j]$ *for* $j\in[0,m]$
$[p/cur()][applyP(p,memb_j)/memb_j]$ *for* $j\in[0,n]$

Call ma_0' to this simple assertion. So, a is the expansion of ma_0'. From (4) we have

$<ty',mth',a_0,s[o\rightarrow s(old)[\texttt{cur()}],y\rightarrow s(young)[\texttt{cur()}]],Lm\cup\{(ty', mth', prest)\}>\longrightarrow bool$

where a_0 is the expansion of

Cond(ty',mth',prest) $[exp_j/par_j]$ *for* $j\in[0,m]$

Remember we want to prove

$<ty,mth,a,_{s,Lm}> \longrightarrow bool$ *(2)*

where a is the expansion of

Cond(ty',mth',prest) $[exp_j/par_j]$ *for* $j\in[0,m]$
$[p/\texttt{cur()}]$
$[applyP(p,memb_j)/memb_j]$ *for* $j\in[0,n]$

If we can prove that, for all path p and method call mc, if we had:

$applyP(p,mc)\text{»}prest\longrightarrow_E a_2$ *(5)*

and also

$applyP(p,mc)[applyP(q,memb_j)/memb_j][q/\texttt{cur()}]\text{»}prest\longrightarrow_E a_1$ *(6)*

where $memb_j$ are the members that appear in the assertion, then it would be the case that

$a_1=a_2[applyP(q,memb_j)/memb_j][q/\texttt{cur()}]$ *(7)*

then, by rule [applyMPath] applied to (4) we would obtain (2).

Let us prove then the property needed in (5), (6) and (7). This should be proved by induction on the structure of meta-assertions. The interesting case is precisely the one where ma is the basic-meta *applyP(p,mc)»prest*. The expansion of a meta-assertion *applyP(p,mc)»prest* is, as rules [Expand1] and [LowerMeta] define, the expansion of *Cond(ty',mth',prest)*$[exp_k/par_k]$ *for* $k\in[0,m]$ with the additional substitutions $[p/\texttt{cur()}]$,$[applyP(p,memb_i)/memb_i]$ *for* $i\in[0,l]$.

In (6), the *Cond(ty',mth',prest)[exp$_k$/par$_k$] for k∈ [0,m]* suffers the substitutions *[p / cur()],[applyP(p,memb$_i$) / memb$_i$] for i∈ [0,l]* when in rule [LowerMeta]. But *p* had first suffered the substitution *[applyP(q,memb$_j$) / memb$_j$]* and *[q / cur()]* in its own member. So, when we perform the substitution *[applyP(p,memb$_i$) / memb$_i$] for i∈ [0,l] [p / cur()]* over the members *memb$_i$* of the expansion of

applyP(p,mc)»prest[applyP(q,memb$_j$) / memb$_j$][q / cur()],

we really get the substitution

[applyP(applyP(q,p),memb$_i$) / memb$_i$] for i∈ [0,l]

In (5), by the induction hypothesis that says that the expansion of the assertion obtained through the LowerMeta rule satisfies the property we are to prove (the predecessor relation here is the same as above and the property is vacuously verified by simple assertions), we would have that the expansion of

Cond(ty',mth',prest)[exp$_k$/par$_k$] for k∈ [0,m]

applied the substitutions *[p / cur()],[applyP(p,memb$_i$) / memb$_i$] for i∈ [0,l]*, would be equal to the expansion of

Cond(ty',mth',prest) *[exp$_k$/par$_k$] for k∈ [0,m]*
[p / cur()]
[applyP(p,memb$_i$) / memb$_i$] for i∈ [0,l]

So, the expansion of

applyP(p,mc)»prest[applyP(q,memb$_j$) / memb$_j$][q / cur()] for j∈ [0,n]

would be the expansion of

Cond(ty',mth',prest) *[exp$_k$/par$_k$] for k∈ [0,m]*
[p / cur()]
[applyP(p,memb$_i$) / memb$_i$] for i∈ [0,l]
[q / cur()]
[applyP(q,memb$_j$) / memb$_j$] for j∈ [0,n]

where memb$_j$ are the members of *Cond(ty',mth',prest) [exp$_k$/par$_k$] for k∈ [0,m]* and *memb$_i$* are the members of *Cond(ty',mth',prest) [exp$_k$/par$_k$] for k∈ [0,m]* including *p*! So, the substitution is the same in the previous case of (6):
*[applyP(applyP(q,p),memb$_i$) / memb$_i$] for i∈ [0,l] (*remember *p* would have been substituted for `cur()` as we wanted to prove.
In what respects point 3) of soundness, if the evaluation of *applyP(p,mc)»prest* stops with error, then by rule [basicM2] it is because (ty', mth', prest) ∈ Lm where the identifiers ty' and mth' are as defined in that rule. By [Expand2] rule we also have that the expansion process stops in error.
***ma* is of a compound form.** If *ma* is, for example, bm$_1$`and`bm$_2$ then, on the one hand, basic-metas bm$_1$ and bm$_2$ are predecessors of *ma* through $<_E$. On the other hand, the expansion of *ma*, by [Expand1] rule is *ma* with the expansion of its basic-metas substituted for its basic-metas. Thus, the expansion of bm$_1$`and`bm$_2$ is a$_1$`and`a$_2$ where a$_1$ and a$_2$ are the expansion of bm$_1$ and bm$_2$, respectively. Let us suppose that

<ty,mth,bm$_1$,s,Lm> ⟶ bool$_1$ and *<ty,mth,bm$_2$,s,Lm> ⟶ bool$_2$*

Then, by some rule for the conjunction,

<ty,mth,bm$_1$`and`bm$_2$,s,Lm> ⟶ bool

Also, by the induction hypothesis,

$\langle ty,mth,a_1,s,Lm\rangle \longrightarrow bool_1$ and $\langle ty,mth,a_2,s,Lm\rangle \longrightarrow bool_2$

Then, by the same rule for the conjunction,

$\langle ty,mth,a_1 and a_2,s,Lm\rangle \longrightarrow bool$

If one of the bm_i is such that $\langle ty,mth,bm_i,s,Lm\rangle \longrightarrow \perp$ then by rule [basicM2] it is because (ty', mth', prest) $\in$ Lm where the identifiers ty' and mth' are as defined in that rule. Evaluation of the conjunction stops with error. By the induction hypothesis, if $\langle ty,mth,bm_1,s,Lm\rangle \longrightarrow \perp$ then also $Lm,ty,mth \vdash, a_1 \longrightarrow_E \perp$. So, the expansion of the conjunction stops with error.

$\langle ty,mth,ma,s,Lm\rangle \longrightarrow_\perp$ *implies* $Lm,ty,mth \vdash ma \longrightarrow_E \perp$

If one of the bm_i is such that $\langle ty,mth,bm_i,s,Lm\rangle$ diverges then the conjunction also diverges. By the induction hypothesis, $\langle ty,mth,a_i,s,Lm\rangle$ also diverges. Then the evaluation of the conjunction of the expanded assertions also diverges. QED.

5.3 Testing Meta-Pre-Conditions

Let us talk now about the way the meta-assertions approach could help in testing pre-conditions without increasing class coupling.

Suppose we want to implement some method `meth` in some type `TA` and for that purpose we need to call a given method `methB` over an object `obj` of type `TB`. This makes `TA` a client of type `TB`. Suppose the iContract specification of `methB` in `TB` were,

```
@pre methB2()==u && methB3().otherTmeth(u)
public void methB(int u) {....}
```

In order to be well-behaved clients of `TB` when calling `methB(actual_u)` over `obj`, we should ensure methB's pre-condition before calling it:

```
if (obj.methB2()==actual_u && obj.methB3().otherTmeth(actual_u))
     obj.methB(actual_u)
else
     ....
```

But then we would be talking to strangers – `methB2()`, `methB3()` and `otherTmeth()`. Here the problem of increasing coupling is even worse because we are dealing with programming language code, not with assertions. The ideal way to do this without talking to strangers would be, for example:

```
if (obj.methB(actual_u)»pre)
     obj.methB(actual_u)
else
     ....
```

But this would imply the expansion of that meta-assertion, not to a simple-assertion as before, but to Java code. This is something our approach does not deal with – the generation of programming language code is left to the monitoring tools associated with the base assertion language.

We can envisage, however, a way to reach this goal. We would precede the process of expansion of meta-assertions with the verification of the existence of meta-assertions in the Java code and its replacement for something manageable by the expansion process.

Because we are dealing with iContract/Java the following statements would take the place of the statement above

```
if (methB_pre(obj,actual_u))
     obj.methB(actual_u);
catch (RunTimeException e) {...}
```

and a private method `methB_pre` with all the parameters of `methB` plus one – an object of type `TB` – would be created in `TA`, with the following specification and implementation:

```
@pre par.methB(u)»pre
private boolean methB_pre(TB par,int u) {return true}
```

This change would be followed by the expansion process which would then expand the pre-condition of this `methB_pre` method to `par.methB2()==u && par.methB3().otherTmeth(u)`. Later, at the time the monitoring code generation tool picked up this method, it would generate Java code in this `methB_pre` method to test for its own pre-condition (and generating an exception in case it is not true). This is exactly what we wanted: at execution time, with monitoring on, this method returns true only if the pre-condition of method `methB` is true and generates an exception otherwise. In case of `methB_pre` triggering an exception, the if statement "aborts" giving way to the catch statement. This statement would entail the same behaviour as the else branch above (the one taken when methB's pre-condition is false). Low coupling and encapsulation would be maintained because whenever the pre-condition of `methB` in `TB` changes, this code does not have to change. It has to be recompiled in order to re-generate the substitution and the expansion.

6. 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.

We also aim at the elaboration of formal proof mechanisms to be able to build proofs of correctness of (meta) contract implementations.

7. REFERENCES

[1] M. Carillo, J. Garcia, E.Pimentel, I. Repiso, Design by Contract in Smalltalk, Journal of Object-Oriented Programming, 9, 7, 1996, p.23-28.

[2] D.Bartezko, C.Fischer, M.Moller and H.Wehrheim, Jass-Java with assertions, Workshop on RunTime Verification, 2001, held in conjunction with the 13th Conference on Computer Aided Verification, CAV'01.

[3] A.Duncan and U. Holze, Adding Contracts to Java with Handshake, Technical Report 1998-32, Univ. California, Santa Barbara, Department of Computer Science (http://www.cs.ucsb.edu/labs/oocsb/papers.html)

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

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

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

[7] 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. Special issue containing selected submissions for the 4'th SPIN workshop, Paris, November, 1998.

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

[9] 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, 2001, p.284-299.

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

[11] M. Karaorman, U. Holze and J.Bruno, jContractor: A Reflective Java Library to Support Design By Contract, Technical Report, Univ. California, Santa Barbara, Department of Computer Science (http://www.cs.ucsb.edu/labs/oocsb/papers.html).

[12] 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. Also Department of Computer Science, Iowa State University, TR #00-15, August 2000 Conger., S., and Loch, K.D. (eds.)..

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

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

[15] 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, 2001 p.299--312.

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