Assertions are objects too!

PHILIPPE COLLET & ROGER ROUSSEAU

UNSA (University of Nice – Sophia Antipolis)
I3S – CNRS (Scientific Research French Centre)
Sophia Antipolis – France

June 20–21, 1996

1 Introduction

Assertions have experienced varied successes since their introduction in programming [Hoa69]. After numerous teaching attempts in the seventies, they have been rarely used in industry until the arrival of languages providing capabilities to check them. They have almost fallen into disuse in universities. Thanks to a long experience with the Eiffel language, both in teaching [MRR88, BMP+90, BR95a] and in several industrial projects [BR95b, CFLR95, CFLR96, LJ95], we think, like Bertrand Meyer [Mey96b], that assertions are as important as other concepts in object technology, like inheritance, polymorphism or encapsulation.

Any improvement of the program semantics knowledge is obviously useful, but provided that verifications can be made. This is why assertions are an efficient and reasonable compromise between the knowledge they provide and the realistic handling they allow. Programmers use a language close to their programming habits and tools can check assertions *in situ*, revealing errors in acceptable time. This provides many uses such as reliability control [Gri81], contracts between programmers [Mey88, Mey92b, Mck95, MMHH95], automatic generation of internal documentation [Mey94a] and according to different points of view [BM93, BR95b], specification of dynamic behaviour during conception [WN95], evolution and adaptation management control [Mey88, Mey96a], comparison and semiautomatic updating [BR95b], consistency constraints [GD94]...

About Eiffel we believe this language "would not work" without assertions, because they play a fundamental role in checking the consistency of complicated, multiple and deep inheritance cases, which are typical in Eiffel libraries [Mey94b]. Moreover assertions are fundamental to control the work of programmers, who only have a *partial knowledge* of the complex and dense universe they reuse. This is why many works related to the integration of assertions into other languages have been conducted, for instance in Ada [Luc87], in C [Ros95], in Smalltalk [CL94a, CGP94] and in C++ [CL94b, PF95]. However the present assertions mechanism of Eiffel suffers from deficiencies of both language and its environments and from methodological problems linked to their use:

– Despite their good ability to reveal errors, assertions do not specify with enough precision the components' behaviour, because tools have some difficulty in extracting relevant information and because the assertion language lacks quantification [BMP+90, KM95].

– Although it is possible to express quantification with *ad hoc* routines, programmers tend to censor themselves, because of the additional work of writing those routines and because they fear a too heavy evaluation cost.

– There are several meanings for assertions (contracts, axioms, theorems, tests, constraints, etc.) and we need to clarify them in order to choose appropriate evaluation methods.

– Assertions are not immutable truths as in mathematics, but they are computer objects with their own life cycle. The confidence in the truth they express evolves with the classes and the population of instances they are in relation with.

Following the current trends in works dealing with Eiffel assertions [Fis92, BM93, MKM93, Mck95, KM95, MMHH95, WN95], this paper proposes a new point of view on assertions and a classification of their different uses in order to choose well-adapted evaluation methods. Section 2 clarifies the differences between mathematical concepts and computer objects and shows that assertions are definitely computer objects. Section 3 presents a classification of assertions. This allows to distinguish different evaluation methods in section 4. Section 5 compares our approach with related works and concludes with discussing further research.

2 Using assertions in programming

Eiffel assertions are logical formulas (currently not quantified) which are dealing with program variables and which are supposed to be true at given times during program execution: during an object inactivity time (class **invariant**), before (**require**) and after (**ensure**) a routine, before an iteration exit test (loop **invariant**) or at a relevant step of a computation (**check**).

To understand our point of view on assertions, it is useful to clarify the fundamental differences between mathematical concepts and computer objects. In our opinion the main difference between mathematics and computer science is about the **evolution** of handled objects. These two subjects are concerned about uncertainty, regularity, number, abstraction, etc., but with different approaches due to presence or absence of evolution.

2.1 Mathematical concepts

Mathematical concepts have no identity on their own, only a value. Without identity, it is impossible to detect any *space-time* evolution, because the memory which is necessary for recording different states cannot be designated. For example, it does not make any sense to say that the mathematical concept 3 was previously 4, because this concept can only be referred through its value. As well the value 3 is unique and cannot be located in space, contrary to the numerous integer memories that can have the **same** value 3.

As mathematical concepts have invariable and sound properties, their knowledge is eternal, without putting any calling into question. Thanks to this stability it is possible to make powerful abstractions on their number (infinite or even uncountable sets), on their individual (fuzzy logic) or collective (statistics) properties.

When uncertainty is introduced in mathematics, it is concerned with objects themselves, not with the reasoning used to describe them. The notion of knowledge life cycle is totally unknown to mathematics and it would never occur to any mathematician to work on "erroneous" concepts.

2.2 Computer objects

On the contrary computer objects are inherently evolutional. They need to be identified in order to distinguish the successive *space-time* states and to detect either temporal evolution (change of an objet state) or spatial evolution (object moving through a network). These states can be described with a finite quantity of information, which can be handled by a computer in almost always acceptable time on a human scale: all disks of a computer can be read in a few hours at most.

Computer objects are perpetually evolving and every object cannot be described by complete and sound theories. The basic and often used objects must be distinguished from more particular ones. As it is too expensive to build a theory of too specific or too short-lived ones, complete theories are built (using algebraic specification for example) in practice for basic library components which are very stable and come from a consensus.

A realistic computer scientist, even if he is rigorous, cannot make the assumption that errors cannot occur. He cannot claim to have a full and safe knowledge of the systems he deals with. The worst is that even his convictions are evolving and must be periodically verified, updated or even forgotten...

2.3 Assertions and other logical formulas

Contrary to mathematicians, computer scientists use a wide variety of formalisms and languages which can be mainly distinguished from the stability and the reusability of the properties they express.

Starting from the most stable and general, logic, which is universal to both exact sciences and social sciences, comes first. Next the mathematical language or its by-products (Z, algebraic specification, lambda-calculus, etc.) can be used to specify the basic elements of computer science (the pre-defined types of programming languages or base libraries). Even if they are very stable, standardization committees are needed to achieve a half consensus (an almost unknown notion to mathematics).

Then come formalisms used to describe specialized components libraries according to several points of view: external for client programmers or internal for programmers who implement or adapt those components using inheritance. There is a fuzzy borderline between stability and complexity levels of those components and some can be the subject for detailed theoretic studies. The main thing is the possibility of incremental definitions, with adjustable levels of precision and abstraction. This is why

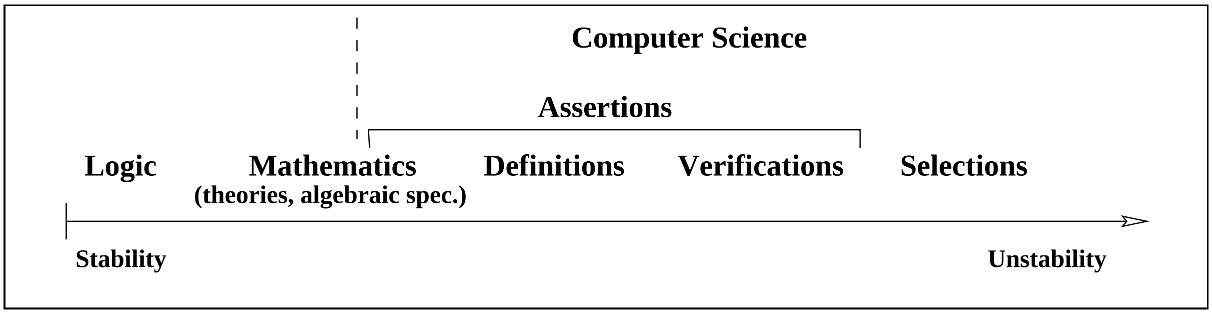

Figure 1. *Assertions and other logical formulas.*

"precondition-postcondition" assertions match well to build those specifications. They can just as well provide a very close definition to algebraic formalism as only some relevant formulas which annotate a routine definition.

In a similar way, class invariants make abstractions on successive states of their instances. They are easier to write and to understand than finite-state automata or transition graphs of Petri nets. Besides, one of the strong points of the BON method [WN95] is to use assertions to describe main behaviours whereas other OOD methods are most often either too brief or too verbose about this subject.

Assertions can also be used to annotate the main steps of an algorithm (in Eiffel **check, invariant and variant** keywords). It was the first use of assertions in program proving. In comparison with the code they annotate, they bring out a better abstraction, and better possibilities of stability and reusability. Tools can exploit them and provide equivalent of Pseudo Description Languages (PDL) of the seventies which are renamed these days "Design Patterns" [GHJV95] but with less precise information.

Finally, at the most changeable level, inside the routines code itself, logical formulas express selections on either data like databases criteria or code like conditions in control structures.

Logical formulas have very different uses in computer science (figure 1) and they express three different kinds of knowledge:

1. Definitions
2. Verifications
3. Selections

Definitions are the most stable and the most general ones. They can use very formal and abstract languages. **Selections** describe temporary properties. **Verifications** are between these two poles. Assertions are both definitions and verifications, but with a wide variety of use cases which are described in section 3. They can express tests, more or less complete definitions, contracts or even consistency constraints.

2.4 Assertions are computer objects

Assertions are not perfectly stable and they are evolving through time. At a given time, they can be inconsistent according to their authors assumptions, to program entities they are connected to (routines, classes, applications, libraries, databases, etc.) and to the state of entities they control, which we call its "applicants" (population of objects, variables). Thus, they go themselves through different **consistency states**. The truth values they express must be qualified with different **confidence degrees**. But the interactions between an assertion and its related entities are implicitly expressed in a class. An implicit definition would facilitate their handling.

Assertions are not only writing, but also *thinking in motion*: they are **computer objects** with their own life cycle. Like other computer objects their usefulness is bound to the quality of the systems that handle them and their use must come with a methodology which control their "software process" evolution. Thus, general handling routines would facilitate this work.

As assertions need a framework which can both express their state and supply handling routines, they can be described by classes using **reification**. Reification consists in making implicit features explicit. An "assertion" object has a state which expresses its consistency level, the relations with its applicants, the confidence degrees of its author and its surrounding system... This object has also routines, which extract a relevant documentation according to the asserted entities and a given point of view, which choose and use an evaluation method (see section 4). The expressive power of an assertion depends not only on the logical formalism used (for example with or without quantification) but also on effective accesses to its *applicants*. The reification facilitates these accesses.

When assertions are reified, the objects which represent them must be regarded as higher-order objects, *i.e. meta-objects*. As assertions are integral parts of the class they annotate, their reified forms are integral parts of the reified entities they are associated with. It is sufficient to observe assertion state and routines to be convinced. The confidence degrees in an assertion, evaluation methods and extracting documentation are not about programming but about meta-programming.

Nevertheless, assertions reification may seem threatening to efficiency. The IREC system [BR95b] developed in our team for the ESPRIT "Business Class" project, aims at the semiautomatic updating of Eiffel classes. It uses a full reification in Eiffel of every entity of the Eiffel language itself, including assertions. Thus, this reification is *reflective*. This work required the writing of about 250 Eiffel classes and the program size in its full reified form is about twenty times bigger than the source text in concrete syntax. Thus this solution is only viable for handling classes during the development or maintenance stages but not during final execution. In our opinion, it is possible to achieve a good compromise *easy handling/efficiency* using a "by necessity" reification technique. The advantage is that additional cost is only paid for when needed contrary to a meta-object protocol.

3 Classification of assertions in Eiffel

In order to propose a classification of Eiffel assertions, the needed object model must be described and classification criteria must be clarified. Then the different kinds of assertions can be studied.

3.1 A model of assertions-related concepts

This model only describes concepts directly linked to assertion issues through an extension of the Eiffel language [Mey92a]. Due to a lack of space, it is briefly presented but most of its principles can be found in [RBC94]. Figure 2 shows the main aspects of the model.

An object is described by its identity, its memory state and its type. Only its memory state may evolve. The identity is represented by a type which is isomorphic to an integer range and which allows to distinguish an object from some others. Thus it permits to detect evolution and to establish explicit references to objects. The object memory is made of its *slots*. Each slot contains either a reference to another object (its identity) or the expansion of an object.

The type of an object defines the subset of its properties which are stable and interesting enough to be controlled. The stated properties of a type are therefore true at any time during program execution and synthetic enough to be applied to several objects. A type has an identity, static properties which are of mathematical kind and independent of any evolution [1], and dynamic properties which express object behaviour like routines definition.

Classes factorize, using inheritance and genericity, properties which are common to several types and also to several objects (instances of these types). Classes are property description units and texts modified by programmers. They are therefore evolving during programming or maintenance stages.

Classes may be generic. Non-generic class defines one and only one type. Generic class describes several types, for example `ARRAY[T]` describes `ARRAY[INTEGER]`, `ARRAY[STRING]`, etc. In Eiffel both classes and types are *frozen* during program execution. Classes can be grouped together in applications or libraries.

Classes may be reified as aggregations of metafeatures which are described by other class instances, not necessarily as metaclasses like in CLOS MOP [2]. Thus classes structure can be described either in Eiffel concrete syntax, or in an abstract syntax which is eventually given in a reified form. This reified form can be used to evaluate assertions which would need access to classes metafeatures.

A collection is a container whose elements can be sequentially handled. Every quantification can be seen as a collection operator. This point of view allows to keep the model opened to unification of databases and software engineering handling [Rou95]. A selection criterion is indeed a filter on one collection.

An extent is a collection of instances which have the same type. It can be *potential* and thus defines the set of all "instantiable" objects, which is often infinite and not traversable. The *real* extent is the set of

1. "$3 + 5 = 8$" is a static property of the `INTEGER` type, true at any time.
2. This classes reification is partly made by the Eiffel library (`INTERNAL`) and fully implemented in the IREC system [BR95b] using Eiffel and also rewritten in Smalltalk at the Laforia laboratory, University of Paris VI.

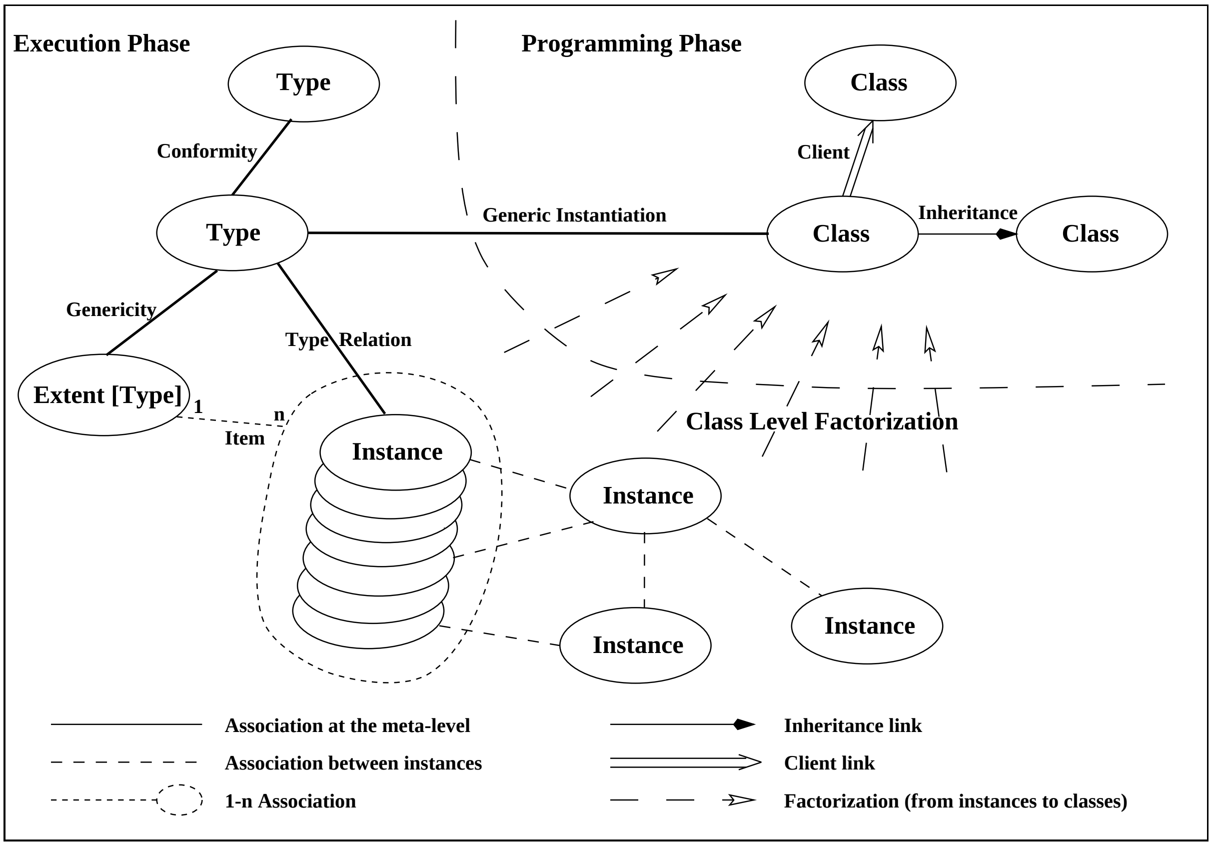

Figure 2. *An overview of the object model.*

all instances which have been created and can be accessed at a given time. Real extents are traversable collections, even if they are implicitly built by the runtime support, and eventually when needed for an assertion evaluation. Our team have already experimented such extensions to the Eiffel language, using both volatile and persistent extents [Col94, CFLR96].

A real extent can be subdivided according to two criteria: *conformance* which allows to deal with only $\mathcal{T}$ type instances (own extent) or with all instances conforming to $\mathcal{T}$ (conforming extent) and *persistency* which allows to deal with persistent instances whose lifetime goes beyond program duration (persistent extent) or not (volatile extent).

3.2 Classification criteria

The retained criteria must provide a methodological help (life cycle, choice of an evaluation technique, exploitation by tools, etc.) that cannot be automatically deduced by tools. Two criteria are taken into account: **entities** on which assertions act and the programmer's **intention**.

Entities on which assertions act

The first classification criterion deals with the relationships between entities which allow to deduce the most appropriate **location** to place the assertion and the most relevant **time** to evaluate it.

Only external relationships are considered so iteration invariants and routine body checks are not taken into account. Three cases are distinguished according as assertions are essential to only one class feature, to several class features or to several classes.

In Eiffel three privileged locations for assertion writing can be pointed out and they are particularly bound to the evaluation time in the object life cycle: just before (**require**) or just after (**ensure**) a computation, or during the time the object state is observable (class **invariant**). Of course these temporal considerations are connected to the concerned entities.

require and **ensure** clauses are normally essential to only one feature, but the class invariant mixes assertions which are about several features of one or several classes and attributes properties. The current Eiffel syntax does not allow to put these properties close to the concerned attributes.

The lack of precision of this layout can be criticized on several points.

– A feature can just as well describe a part of an object state as a behaviour which is going to modify this state. Whereas it is possible to define the result of a function with a postcondition, it is not possible for an attribute. This lack of uniformity, already noticed in [MKM93], will be corrected below.

– an object state is described by features which we call *accessors*. They can be distinguished according to their memorized (attribute) or computed (function) aspects. They must be distinguished from each other and from behavioural features described below.

– Behaviour is described by procedures which modify the object state and which we called *modifiers*. These procedures are defined in a synthetic way by a pre-post mechanism, but used assertions should be distinguished from the ones which are used for computed accessors (functions).

Class invariant mixes properties which are located in this place for many reasons:

– They cannot be located elsewhere (attribute properties).

– They can only be verified during the inactivity time of the object so that they are independent of routines activity.

– They describe essential properties which are connected to several features of the same class.

– They express relevant properties of several objects which are logically linked to each other (consistency constraints for example).

We propose to better specialize class invariants so that they only express properties that concern several attributes of the same class. So attributes definitions (currently placed in the class invariant) are moved to attributes themselves. As the Eiffel language does not provide any particular construction to deal with *associations* (contrary to ODMG-93 model [Cat94]), we propose [CFLR96] to reify these associations using specialized classes (possibly linked by inheritance). Thus assertions about associated objects could be regarded as assertions which are about features of the association class.

Thus we retain four privileged places to put assertions according to concerned entities:

– the control of a state access (attribute or function),

– the beginning of a computation (function or procedure),

– the end of a procedure,

– the class invariant which would only express properties about attributes of the same class.

Programmer's intention

The second classification criterion is the intention which justifies the assertion writing. Indeed using assertions allows to specify in an incomplete way. It lets the programmer write what sounds relevant to him.

Contrary to complete and formal specification techniques which only aim for a "necessary and sufficient" definition (what is not always the most readable and the most useful), assertions can be used in a partial and redundant way according to the different intentions of the programmer. As seen in section 2, assertions are located between definitions of mathematical concepts and selections during program execution and have two uses:

– The **definition** describes what is necessary to state in the assertion. Even though this definition is not complete, the main thing is always described. The remainder can be written in comments.

– The **verification** describes all *redundant* information that helps towards documentation and test of the specified entity and even of the specification itself. We propose to distinguish them from others by using the reserved words **check** which is already used inside routines body.

The two intentions can be clearly differentiated by noticing that everything that can be deduced from definition is part of the additional verification. Thus: **Definition** $\Rightarrow$ **Verification**.

3.3 Routines prerequisite

The prerequisite of a routine is linked to the beginning of a computation, *i.e.* function or procedure in Eiffel. It expresses every condition that must be true when calling the routine: this is the *client contract*. So it must only use features which are visible to client programmers, *i.e.* which are about the object abstract state or the routine parameters.

Example:

```
class STACK [ T ]
...
top: T
  require not_empty: not empty
```

The notion of requisite may be applied to attributes [MKM93] but they would be redundant to access functions so we did not choose this solution as discussed below.

One can note that contrary to other assertions, requisites must be "necessary and sufficient": "sufficient" in order to make the agreement between class programmers and clients assumptions and "necessary" in order not to reduce offered services. Thus no **check** clause is added to the **require** one in routine preconditions.

Example:

```
class STRING
...
prepend ( s: STRING)
  require argument_not_void: s /= Void

put (c: CHARACTER; i: INTEGER)
  require valid_index: i > 0 and i <= count
```

3.4 Access specification

A state consultation can be expressed by an accessor which can be implemented by a function or an attribute. Primary and secondary accessors are distinguished.

Primary accessors describe the object abstract state, independently of implementation choices that can use attributes or functions. So they cannot be directly defined but some of their properties can be tested thanks to the **check** clause or deduced from their use in modifiers specification.

Example:

```
class STRING
...
item(i: INTEGER): T
  require valid_index: i > 0 and i <= count

count: INTEGER
  check count >= 0 ; count <= capacity
```

Secondary accessors are calculated using primary accessors by functions which we propose to define with a **define** clause instead of the **ensure** one. The **check** clause is also added.

Example:

```
duplicate: STRING
  define
    Result.is_equal(Current)
    Result.capacity = count
  check
    Result.count = count
```

As stated in the previous section, an accessor which is implemented by an attribute has no requisite. One can think that it distorts the language uniformity and restricts possibilities of redefinition between attributes and functions. In fact, an accessor implemented by a function with a requisite has no reason to be redefined as it is easy to return any attribute value in an optimized way. An attribute must not be redefined as a function with requisite for two semantic reasons: The function would not be able to change the attribute and its requisite would violate the substitution rule. We deduce that the presence or absence of a precondition is a very stable property that cannot be changed with inheritance. Only the weakening (or the suppression) of an existing precondition is allowed.

3.5 Modification specification

In Eiffel, modifiers can only be implemented by procedures. Their specification is only expressed in the **ensure** clause. The previous state is referenced through the **old** operator[3]. In order to represent verification a **check** clause is added to the **ensure** one.

3. The too high priority of old does not make its use easy.

Example:

```
append(s: STRING)
  require
    s /= Void
  ensure
    Current.is_equal(old Current + s)
  check
    count = old count + s.count
```

Using two different keywords, **define** and **ensure**, facilitates the specification of side-effet functions as that clearly distinguishes the definition of the result from the side-effet itself.

3.6 Invariant

Invariants can be distinguished according to whether they are about the current object state or not. If an invariant is about this state, it is true for this instance at any time.

On the contrary if an invariant does not make reference to the object state, it expresses a higher-order property which is much more fundamental and stable. In our model, this is a *property of the object type*, which is therefore true at any time for any instance of this type extent. Type properties often establish inheritance contracts which are automatically propagated in heirs through an implicit boolean **and**. This prevents subsequent adaptations from contradicting the initial intentions of the ancestor classes authors: type properties play a "genetic code" role.

We propose to distinguish instance invariant from type invariant using a syntactic construction:

```
invariant{instance}
  ...
invariant{type}
  ...
```

No distinction is made between abtract and concrete (implementation) state of an object as this can be automatically processed by view extraction tools like `short`.

Instance invariant

At the class level, the instance invariant summarizes behaviours and states and it forms the *semantic network* of the class. As with consultations and modifications, definitions are distinguished from verifications using the **define** and **check** keywords.

Example:

```
class LIST [ T ]
  empty : BOOLEAN
    -- Is structure empty? (FROM FINITE)
    define
      Result = (count = 0)
  off : BOOLEAN
    -- Is there no current item?
    define
      Result = before or after
  ...
invariant{instance}
  define
    empty implies off
  check
    empty implies (after or before)
    before implies off
    after implies off
```

It is easy to prove that everything in the **check** clause can be deduced from the **define** ones. On the other hand, the assertion given in the **define** clause of the invariant cannot be deduced from any other and it connects the two features `empty` and `off`. So they must be written at the class level, *i.e.* in the class invariant.

Type Invariant: axioms and theorems

The type invariant follows the same syntax as the instance invariant. The definitions that express properties are axioms as the ones that are deduced from axioms are theorems.

Example: **deferred class** `GROUP`

```
   ...
invariant{type}
  define -- Axiom
    ∀ x, y, z : like Current • x + (y + z) = (x + y) + z
```

This example is borrowed from [BMP+90] and defines the associative axiom of the `+` operator in a group. Many other examples of type invariant in Eiffel can be found in [San93].

4 Assertions evaluation

In the same way as for the classification, the evaluation criteria that must be taken into account are going to be studied first.

4.1 Evaluation criteria

Of course the location and intention criteria which have been useful for classification are going to be reused. Two other criteria are added as they act on the evaluation method and they are related to both its life cycle and its evaluation cost.

The assertion life cyle integrates the assertion period of use and relationships with entities which are internal or external to a system.

The assertion period of use is a function of its nature. It may be only used to test an entity which has evolved and then it plays the role of non-regression test.

During system execution some external entities interact with assertions:

- programmers because contracts are devoted to them,
- persistent collections as assertions act on them as consistency constraints,
- classes whose evolution is controlled by assertions [BR95b],
- applications whose assertions are some tests,
- libraries whose assertions are definitions.

Internal entities are the *applicants* of assertions (see section 2.4). They can check the assertion itself, or the assertion can test them. The evaluation may also simultaneously control the assertion and its applicants.

The evaluation cost must also be taken into account when dealing with quantifications. The compromise which optimizes the ratio *confidence degree / evaluation cost* must be found. Evaluation time may be reduced in two ways:

- *time*: the time during which object properties are checked must be finely controlled.
- *population of concerned objects*: objects related to assertions are important through both their nature (execution time objects, persistent collections, test-case generated objects, etc.) and the technique that can be used to evaluate their assertions (exhaustiveness or sampling).

4.2 Applications

Different points which could be improved in Eiffel programming systems are now described. They focus on assertions as they aim at putting them to their best use.

Semiautomatic triggering

Currently Eiffel systems organization is set up in the ACE[4] file which describes two different aspects of a system. The first one is very static as it describes all elements which make an Eiffel system up: root, cluster, external parts. The second is a more evolutional aspect as it deals with assertions and debug clauses triggering. Triggering is defined through different levels which apply on the whole system, on one or several clusters or on a set of given classes.

Although it is possible to handle an *ACE file* using a simple text editor for stable properties, unstable ones which concern in particular assertions triggering should only be expressed through an interactive

4. Assembly of Classes in Eiffel [Mey92a, Appendix D].

tool like a "dashboard". It would permit very quick modifications of some properties without caring for syntax and with the possibility of controls driven by methodological rules.

Just like typing, assertions evaluation could be reduced to the only clauses that are affected by classes evolution. When a class is modified, a sophisticated evaluation system could establish which classes are still compatible, which ones should be reverified, which clauses must be evaluated and, when and how according to their nature. Of course the programmer should always manually intervene in problematic situations. A similar technique was used with success in the IREC project [BR95b] in order to update in a semiautomatic way Eiffel classes using the knowledge base created by reification.

This technique could be applied to assertions triggering and would allow associating more or less strong verifications according to the nature of observed change. It would also allow to activate stronger controls on "hot" parts of a system. Fine-grained assertion levels would be needed and they would take into account quantification and other techniques sketched out below. An evolution management system like IREC would be necessary to raise or to reduce assertion levels according to changes and dependencies. Everything would be controlled through a dashboard that could set up both default levels on whole system and *critical* classes which must be strongly controlled. The classes which are dependent of these ones (clients, heirs) would automatically be checked. Assertions would thus be triggered according to both critical classes and changes.

Efficient evaluation

Like in any optimization activity, the number of elements to be verified must be limited and the cost of every verification must be reduced.

The number of verifications can be reduced in universal quantification by using a random sampling technique which allows to get a good confidence degree for a reasonable cost [DW80]. Only requisites need an exhaustive verification, other kinds of assertion may be sampled, including type invariants.

Currently Eiffel environments only use application objects to test classes. Evaluation costs can be reduced by deferring to a period of low activity (for instance during the night) controls which are not urgent. Requisites are not concerned as they must always be immediately evaluated during testing stages but all other assertions are concerned. In a non-persitent application, assertions evaluation could be coupled to garbage collecting. A persistent application allows many possibilities of adapting the exhaustiveness degree to the computer load.

Moreover **check** clauses evaluation can be coupled to definition clauses (**ensure** and **define**) because definitions imply verification clauses. So it is not necessary to achieve a high level of exhaustiveness for those verifications. Only a sample of instances which are used to test the definition (which may be a sample too) are needed to control the consistency of the **check** clauses.

Finally, instances samples could be built by testing programs (by hand or using generators). They could be made persistent and could be used as a "standard" in order to verify the non-regression of an application. In this case assertions would be evaluated on the "standard" instances instead of the application instances. They would play the role of consistency constraints for the "standard".

5 Discussion

5.1 Related works

Holland approach [Hol92] may be regarded as a reification of the contract notion but only applied to algorithms description. Thus contract defines the constraints between the different complex structures which are used in an algorithm.

Quantified assertions in Eiffel were proposed by McKim and Mondou [MKM93]. Kent and Maung [KM95] are giving a formal semantics for type quantification. The BON method [WN95] proposes quantifications on both types and collections. Quantifications on collections are then translated into boolean funtions which explicitly traverse these collections. But all those works do not regard assertions as objects and quantifiers are not regarded as collection operators in the same way as selection operators. To our knowledge, our prototype which was implemented in Eiffel 2.3 is the only one to provide traverse of own and conforming type extents.

Works on the classification of assertions are rare. McKim [Mck95] presents building methodological rules but he uses a succinct division of assertions. Rosenblum [Ros95] makes a full classification of assertions which are used in his system APP (a C preprocessor). This classification is too specific to C and is only used to distinguish different sorts of errors according to assertions nature.

Geppert and Dittrich [GD94] apply "programming by contract" to object-oriented databases so they can express contracts using quantifications on classes, class extents and applications. It would be interesting to use the introspection facilities of the IREC system [BR95b] in order to apply the same principles to Eiffel coupled to an OODBMS [CFLR95, CFLR96].

5.2 Conclusion

This paper have shown that assertions should be no more regarded as simple program annotations or as immutable mathematical truths but as full-fledged objects with their own life cycle. Like other objects they can be reified by necessity which can facilitate their interactions with other entities of a system, in particular collections of volatile or persitent instances.

A classification of assertions have been presented. It is built on a general object model which is compatible with both software engineering and databases. This classification distinguishes definitions from verifications, state modifications from function results and instance invariants from type invariants. It allows to differentiate evaluation methods which could be easily applied to current Eiffel environments.

The different projects which have been conducted in our team show the feasibility of the proposed techniques to traverse any type extent (volatile or persistent) and to reify concepts which are useful to advanced programming systems.

Research prospects are numerous. On the one hand we are working on the complete integration of assertions in the sofware process, from conception and specification to implementation and testing of classes. On the other hand we continue our research in object models common to software engineering and databases. In particular we aim at the reconciliation of assertions and selections criteria of databases queries.

References

[BM93] Bielak R. & McKim J. C., " The many faces of a class: Views and contracts ". Meyer B., Singh M. & Ege R., Eds., *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 11, USA'93)*. p. 153–161. Prentice Hall Inc. (Englewood Cliffs, NJ), 1993.

[BMP+90] Boussard J.-C., Michel C., Partouche J.-P., Rousseau R. & Rueher M., " An Assessment of the Eiffel language *(in french)* ". *Technique et Science Informatique*, vol. 9, n° 4, p. 331–373, Sept. 1990.

[BR95a] Boussard J.-C. & Rousseau R., " Why and How to teach Eiffel in Software University Education. Our Experience at UNSA for Seven Years ". In Hoffmann H.-J., Ed., *Eiffel in Schule und Hochschule; Ausbildung in objektorientiertem Programmieren*, p. 1–24. Technische Hochschule Darmstadt, `http://www.pu.informatik.th-darmstadt.de/eis/` , 26–27 May 1995.

[BR95b] Brissi P. & Rousseau R., IREC: An Object-Oriented Abstract Representation to Handle Software Components in a Persistent Framework. In Alagar V. S. & Missaoui R., Eds., *Object-Oriented Technology for Database and Software Systems*, p. 6–21. World Scientific Pub. Co., 1995.

[CGP94] Carrillo-Castellón M., Garcìa-Molina J. & Pimentel E., " Eiffel-like assertions and private methods in Smalltalk ". In Magnusson B., Meyer B., Nerson J.-M. & Perrot J.-F., Eds., *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 13, Europe'94)*, p. 479–488. Prentice Hall Inc. (Englewood Cliffs, NJ), 1994.

[Cat94] Cattell R., Atwood T., Dubl J., Ferran G., Loomis M. & Wade D., *The Object Database Standard: ODMG - 93*. Morgan Kaufmann Publishers Inc., 1994.

[CL94a] Cheon Y. & Leavens G. T., " The Larch/Smalltalk interface Specification Language ". *ACM Transactions on Software Engineering and Methodology*, vol. 3, n° 3, p. 221–253, July 1994.

[CL94b] Cheon Y. & Leavens G. T., " A Quick Overview of Larch/C++ ". *Journal of Object Oriented Programming*, vol. 7, n° 8, p. 39–49, Oct. 1994.

[CFLR95] Chignoli R., Farré J., Lahire P. & Rousseau R., FLOO: A Strong Coupling Between Eiffel Language and O2 DBMS. In Alagar V. S. & Missaoui R., Eds., *Object-Oriented Technology for Database and Software Systems*, p. 206–223. World Scientific Pub. Co., 1995.

[CFLR96] Chignoli R., Farré J., Lahire P. & Rousseau R., " FLOO: a persistent programming system for Eiffel *(in french)* ". *Technique et Science Informatique*, vol. 15, n° 3, p. 1–29, June 1996. To appear, June 1996.

[Col94] Collet P., " *An efficient Runtime Support for Eiffel (in french)* ". Techn. Report, I3S – URA CNRS 1376 – University of Nice – Sophia Antipolis, Sept. 1994.

[DW80] Duran J. W. & Wiorkowski J. J., " Quantifying Software Validity by Sampling ". *IEEE Transactions of Reliability*, vol. 29, n° 2, p. 141–144, June 1980.

[Fis92] Fischbach R., " Programming by contract – Does Eiffel satisfy the model? – *(in german)* ". In Hoffmann H.-J., Ed., *Eiffel, Proceedings of Workshops of German Chapter of the ACM*, N° 35. B.G. Teubner, Stuttgart, 1992. p. 55-68.

[GHJV95] GAMMA E., HELM R., JOHNSON R. & VLISSIDES J., *Design Pattern – Elements of Reusable Object-Oriented Software (Foreword by Grady Booch)*. Addison-Wesley Publishing Co. (Reading, MA), 1995.

[GD94] GEPPERT A. & DITTRICH K. R., " Specification and Implementation of Consistency Constraints in Object-Oriented Database Systems: Applying Programming-by-Contract ". In *GI-Conference DatenBanksysteme in Büro, Technik and Wissenschaft (BTW)*, March 1994.

[Gri81] GRIES D., *The Science of Programming*. Springer-Verlag (Berlin), 1981.

[Hoa69] HOARE C., " An axiomatic Basis for Computer Programming ". *Communications of the ACM*, vol. 12, n° 10, p. 576–580, Oct. 1969.

[Hol92] HOLLAND I. M., " Specifying reusable Components using Contracts ". In LEHRMANN MADSEN O., Ed., *European Conference on Object-Oriented Programming (ECOOP'92)*, vol. 615 of *Lecture Notes in Computer Science*, p. 287–308. Springer-Verlag (Berlin), Jul. 1992.

[KM95] KENT S. & MAUNG I., " Quantified Assertions in Eiffel ". In *International Conference on Technology of Object-Oriented Languages and Systems (Tools 18, PACIFIC'95)*, Prentice Hall Inc. (Englewood Cliffs, NJ), 1995.

[LJ95] LAHIRE P. & JUGANT J.-M., " Lessons Learned with Eiffel 3: the K2 Project ". In EGE R., SINGH M. & MEYER B., Eds., *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 17, USA'95)*, p. 207–215. Prentice-Hall Inc., (Englewood Cliffs, NJ), 1995.

[Luc87] LUCKHAM D. & OTHERS, Eds., *Anna: A Language for Annotating Ada Programs*. N° 260 in Lecture Notes in Computer Science. Springer-Verlag, New York, 1987.

[MKM93] MCKIM J. C. & MONDOU D. A., " Class Interface Design: Designing for Correctness ". *The Journal of Systems and Software*, vol. 23, n° 2, p. 85–94, Nov. 1993.

[Mck95] MCKIM J., " Class Interface Design and Programming by Contract ". In EGE R., SINGH M. & MEYER B., Eds., *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 17, USA'95)*, p. 395–419. Prentice-Hall Inc., (Englewood Cliffs, NJ), 1995.

[Mey88] MEYER B., *Object-Oriented Software Construction*. Series in Computer sciences (C.A.R. Hoare Ed.). Prentice Hall Inc. (Englewood Cliffs, NJ), 1988.

[Mey92a] MEYER B., *Eiffel: The Language*. Object-Oriented Series, Prentice Hall Inc. (Englewood Cliffs, NJ), 1992.

[Mey92b] MEYER B., " Applying "Design by contract" ". *IEEE Computer*, vol. 25, n° 10, p. 40–51, Oct. 1992.

[Mey94a] MEYER B., *An Object-Oriented Environment: Principles and Application*. The O-O series. Prentice Hall Inc. (Englewood Cliffs, NJ), 1994.

[Mey94b] MEYER B., *Reusable Software: The Base Object-Oriented Component Libraries*. Prentice Hall Inc. (Englewood Cliffs, NJ), Apr. 1994.

[Mey96a] MEYER B., *Object-Oriented Software construction*. The O-O series. Prentice Hall Inc. (Englewood Cliffs, NJ), 2nd edition, To appear, 1996.

[Mey96b] MEYER B., " Building Bug-free O-O Software: An Introduction to Design by Contract ". *Object Currents*, `http://www.sigs.com/objectcurrents/`, vol. 1, n° 3, March 1996.

[MRR88] MICHEL C., ROUSSEAU R. & RUEHER M., " First Experiences with Eiffel: an Assessment in Teaching and Research *(in french)* ". In *First Eiffel users meeting, Paris*, p. 27 Jan. 1988. p. 1–13. Tech. report LISAN 88-02, F - Sophia Antipolis In *4ème Colloque AFCET de Génie Logiciel*, p. 249–258. AFCET, Oct. 1988.

[MMHH95] MITCHELL R., MAUNG I., HOWSE J. & HEATHCOTE T., " Checking Software Contracts ". In EGE R., SINGH M. & MEYER B., Eds., *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 17, USA'95)*, p. 97–106. Prentice-Hall Inc., (Englewood Cliffs, NJ), 1995.

[PF95] PORAT S. & FERTIG P., " Class Assertions in C++ ". *Journal of Object Oriented Programming*, vol. 8, n° 2, p. 30–37, May 1995.

[Ros95] ROSENBLUM D. S., " A practical Approach to Programming with assertions ". *IEEE Transactions on Software Engineering*, vol. 21, n° 1, p. 19–31, Jan. 1995.

[RBC94] ROUSSEAU R., BRISSI P. & COLLET P., " From Eiffel to a General Object Model: Our Experience and Suggestions ". Position paper. In CUMMINS F. A., Ed., *9th Annual ACM Conf. on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'94), Workshop on "The Object Engine: Foundation for Next Generation Architectures"*, p. 29–37, 23–27 Oct. 1994.

[Rou95] ROUSSEAU R., " Towards a Common Object Model for Software Engineering, Data Bases and CAD *(in french)* ". In *Journées du GDR Programmation du CNRS*, p. 99–109. ENSIMAG, Univ. of Grenoble, 22–24 Nov. 1995.

[San93] SANDERS P., " A Case study in Object-Oriented Programming. Algebraic Structures in Eiffel ". MEYER B., SINGH M. & EGE R., Eds., *International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 11, USA'93)*. p. 379–388. Prentice Hall Inc. (Englewood Cliffs, NJ), 1993.

[WN95] WALDÉN K. & NERSON J.-M., Eds., *Seamless Object-Oriented Software Architecture – Analysis and design of reliable systems*. The O-O series. Prentice Hall Inc. (Englewood Cliffs, NJ), 1995.