Correctness and contracts

9.1 OVERVIEW

Eiffel software texts — classes and their routines — may be equipped with elements of formal specification, called **assertions**, expressing correctness conditions.

Assertions play several roles: they help in the production of correct and robust software, yield high-level documentation, provide debugging support, allow effective software testing, and serve as a basis for exception handling. With advances in formal methods technology, they open the way to proofs of software correctness.

Assertions are at the basis of the **Design by Contract** method of Eiffel software construction.

This chapter describes assertions and the resulting notion of correctness of a class. It also specifies how the supporting development environment should help check correctness conditions at run time.

9.2 WHY ASSERTIONS?

One could write entire systems without assertions. Some Eiffel developers are even rumored to have done so. In fact, assertions have no effect on the semantics of correct systems — in theory, the only one that matters.

"Deviant Eiffel Programmers", in Proceedings of RACOON 13 (Report of Annual Conference on Object-Oriented Neuropsychiatry), Tahiti, 2005, pages 3456-3542.

Do not look, however, for a SHORTCUT sign suggesting that you skip this chapter on first reading. Assertions are a key element of software development in Eiffel and omitting them would be renouncing a major benefit of the method.

Assertions serve to express the specification of software components: indications of *what* a component does rather than *how* it does it. This is essential information for building the components so that they will perform reliably, for using the components, and for validating them.

The classes of the EiffelBase Library provide many examples of the use of assertions to express abstract properties of classes and routines. Consider the many descendants of class *CHAIN*, describing sequential data structures ("chains") such as lists. They enable clients to manipulate a cursor, allowed to go one position off the right and left edges of a chain, but no further. An assertion occurring in the **class invariant** of the corresponding classes expresses this property:

```
0 <= index; index <= count + 1
```

where *index* identifies the current cursor position, and *count* is the number of elements in the structure. The invariant must be guaranteed by every creation procedure of the class, and maintained by every exported routine.

In the same classes, a client may use exported procedures such as *start*, *finish*, *forth* and *back* to move the cursor. The bodies of these procedures depend on the implementation chosen (linked, array etc.) but many of their important properties are expressed by implementation-independent assertions, so that a version of *forth* will have the form

```
forth
            -- Move forward one position.
        require
            not_after: not after
        do
            … Some appropriate implementation …
        ensure
            moved: position = old position + 1
        end
```

The **require** and **ensure** clauses introduce a **precondition** and **postcondition**:

- The precondition states the condition under which *forth* is applicable: the cursor must not be "after" the right edge, as defined by the boolean function *after*. Any client calling *forth* must guarantee this condition.
- The postcondition states the property which the procedure must guarantee at the completion of any correct call: the cursor index will have been increased by one (the expression **old** position denotes the value of *position* as captured on routine entry). Any client may assume this condition after a call to *forth*.

The optional labels not_after *and* moved, *specimens of* Tag, *serve as documentation and also for error messages. See below.*

As these examples indicate, assertions are not instructions; they do not necessarily have an effect at execution time. Instead, they express properties that should be satisfied by the implementation. In other words, an assertion is a *description*, not a *prescription*.

Preconditions, postconditions, class invariants and other uses of assertions described below (in particular loop invariants) define **contracts** for the corresponding software elements: features, classes, loops.

For precise terminology: a *contract* is the specification of a software element; it is made of *assertions*. For example the contract of a feature comprises its precondition and postcondition; the contract of a class includes the contracts (as just defined) of all its features, plus the class invariant.

Contracts have important applications throughout software development. Among others:

- By helping developers to state precisely the formal properties of software elements, they enhance the correctness and reliability of the resulting software. The underlying theory of **Design by Contract**, views the construction of a software system as the fulfillment of many small and large contracts between clients and suppliers.

See "Object-Oriented Software Construction".

- Assertions may also be monitored at run time, providing a powerful tool for **testing** and **debugging** software.
- Contracts serve as the basis for automatic documentation tools, which produce abstract interface documentation of a class by extracting implementation-independent information from the class text.

← "-DOCUMENTING THE CLIENT INTERFACE OF A CLASS", 7.9, page 212.

- Rules on the fate of assertions in inheritance (invariant accumulation, precondition weakening, postcondition strengthening) provide a clear methodological framework for the proper use of inheritance and associated mechanisms of polymorphism and dynamic binding.

The first and second of these applications explain why the semantics of contracts, as defined later in this chapter, involves two aspects:

- The basic semantic role of contracts is to define the **correctness** of classes and their features. a class is correct if its implementation satisfies the contracts. Determining correctness is a matter of *mathematical proofs*, performed by people or by proof tools. Most of today's development environments do not yet provide proof tools, but the notion of correctness is still essential to understand the role of contracts.

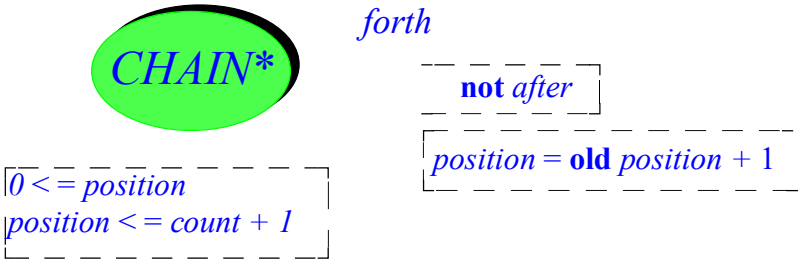
→ "THE CORRECTNESS OF A CLASS", 9.12, page 252.

- Even in the absence of proofs, Eiffel environments must provide mechanisms for **run-time monitoring** of assertions, for the debugging and testing benefits just mentioned. That part of the semantics specifies when and how to evaluate assertion clauses during execution,

→ "RULES OF RUN-TIME ASSERTION MONITORING", 9.13, page 253.

9.3 GRAPHICAL CONVENTION

The figure below illustrates the graphical representations that may be used to show routine preconditions, routine postconditions and class invariants.



For routines' assertions each "drawer" figuratively opens on only one side: the routine obtains an input condition from the left drawer and delivers an output condition through the right drawer. The class invariant combines both conventions.

9.4 USES OF ASSERTIONS

Assertions appear in the following constructs:

- The Precondition and Postcondition parts of an Attribute_or_routine.
- The Invariant clause of a class.
- The Check instruction.
- The Invariant of a Loop instruction.

Attribute_or_routine: page 143.
Class_declaration: page 119.
Check: page 249.
Loop: page 495.

In addition, loops may have variants, which are integer expressions rather than assertions but play a closely related role.

All the constructs involving assertions are optional.

9.5 FORM OF ASSERTIONS

An Assertion, introduced by such keywords as **require** (for preconditions), **ensure** (for postconditions), **invariant** (for class and loop invariants) and **check** (for Check instructions) is made of one or more Assertion_clause, each based on a boolean expression, as in the precondition

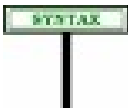
```
require
        point_exists: ce /= Void
        positive_radius: ra > 0.0
```

from a routine with formal arguments *ce* and *ra*. This expresses that in a correct call to be correct the first argument must be non-void and the second argument must be positive. In this example each Assertion_clause is labeled by a Tag (point_exists, positive_radius).

The general form of an Assertion, and the syntax of constructs where it may appear, are:

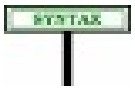
A related construct is the loop Variant, described later in this chapter (9.14).

← On Tag_mark, see page ====.



Assertions

Precondition	≜	**require** [**else**] Assertion
Postcondition	≜	**ensure** [**then**] Assertion [Only]
Invariant	≜	**invariant** Assertion
Assertion	≜	{Assertion_clause ";" …}*
Assertion_clause	≜	[Tag_mark] Unlabeled_assertion_clause
Unlabeled_assertion_clause	≜	Boolean_expression | Comment
Tag_mark	≜	Tag ":"
Tag	≜	Identifier



A Boolean_expression, as used in this syntax definition, is simply an Expression constrained to be of type *BOOLEAN*.

→ *Syntax on page 761 as part of the discussion of expressions,.*

The semicolon is optional as Assertion_clause separator. This requires a non-production rule to avoid any ambiguity:

Syntax (non-production): Assertion Syntax rule

An Assertion without a Tag_mark may not begin with any of the following:

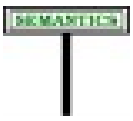
1 • An opening parenthesis "(".

2 • An opening bracket "[".

3 • A non-keyword Unary operator that is also Binary.

This rule participates in the achievement of the general Semicolon Optionality rule. Without it, after an Assertion_clause starting for example with the Identifier *a*, and continuing (case 2) with [*x*] it is not immediately obvious whether this is the continuation of the same clause, using *a* [*x*] as the application of a bracket feature to *a*, or a new clause that starts by mentioning the Manifest_tuple [*x*]. From the context, the validity rules will exclude one of these possibilities, but a language processing tool should be able to parse an Eiffel text without recourse to non-syntactic information. A similar issue arises with an opening parenthesis (case 1) and also (case 3) if what follows *a* is *–b*, which could express a subtraction from *a* in the same clause, or start a new clause about the negated value of *b*. The Assertion Syntax rule avoids this.

The rule does significantly restrict expressiveness, since violations are rare and will be flagged clearly in reference to the rule, and it is recommended practice anyway to use a Tag_mark, which removes any ambiguity.



Semicolons or not, the order of Assertion_clause components of an Assertion is significant. More precisely, the semantic specification below treats them as if they were separated by the **and then** binary boolean operator (replacing the semicolon if present). From that operator's own semantics this means that:

→ *"SEMISTRICT_BOOLEAN_OPERATORS", 28.6, page 774.*

- The value of an Assertion is true if and only if every Assertion_clause in the Assertion has value true.

- If an Assertion_clause has value false, so has the whole Assertion in which it appears, even if the value of a subsequent clause is not defined.

Because of the second of these properties, if an Assertion_clause *b* only makes sense when another, *a*, is true, you should write *a* before *b*. For example if you need a clause of the form *x.some_property* and *x* is of a detachable type, and hence could be void, you should write the assertion as

```
x /= Void
x.some_property
```

where the first property implies that the second one, whether or not it holds, is meaningful. As another example, assuming an integer array *a* and an integer *i*, a routine precondition could read

```
require
        i >= a.lower
        i <= a.upper
        a [i] > 0
```

The last clause relates to the *i*-th item of *a*, defined only if *i* is within the array's bounds as expressed by the first two clauses (which we could also express more concisely as *valid_key* (*i*)). It would be incorrect here to reverse the order of clauses.

A few straightforward definitions are useful:

Precondition, postcondition, invariant

The **precondition** and **postcondition** of a feature, or the **invariant** of a class, is the Assertion of, respectively, the corresponding Precondition, Postcondition or Invariant clause if present and non-empty, and otherwise the assertion **True**.

So in these three contexts we consider any absent or empty assertion clause as the assertion **True**, satisfied by every state of the computation. Then we can talk, under any circumstance, of "the precondition of a feature" and "the invariant of a class" even if the clauses do not appear explicitly.

These are "local" assertions because unlike later "unfolded forms" they do not take into account the ancestor versions that, combined with the local clauses, will yield the full precondition, postcondition or invariant applicable to a feature or class.

→ *"REDECLARATION AND ASSERTIONS", 10.17, page 283, and also UNFOLDING ASSERTIONS UNDER INHERITANCE below.*

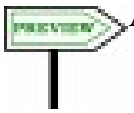
Coming back to the syntax: an Assertion_clause may be preceded by a Tag, such as point_exists and positive_radius in the earlier example. Such tags are identifiers, with no validity constraint. They are useful for documentation purposes; in addition, they help produce precise error messages in the case, studied below, of run-time assertion monitoring. Although the Tag is optional and does not affect the semantics of correct programs, the style rule recommends including it for clarity.

← *Page 232.*

Finally, you will have noted that the syntax for Unlabeled_assertion_clause allows a clause that consists of just a Comment. This is useful for documenting conditions that you cannot or do not wish to express formally; for example:

```
require
        acyclic: -- The structure is not cyclic.
```

9.6 UNFOLDING ASSERTIONS UNDER INHERITANCE



Assertions have a close relationship with inheritance. In particular:

- The actual invariant applicable to a class includes, in addition to any clauses appearing in the class itself, all those from its ancestors.
- An inherited feature may change the assertions of its parent version, weakening the precondition through a **require else** clause and strengthening the postcondition through an **ensure then** clause.

These notions will be captured, in the discussion of feature adaptation, by the notion of *unfolded form* of an assertion, the result of including both immediate and inherited elements. Most of the semantic properties of contracts rely on the unfolded form.

→ *"Unfolded form of an assertion", page 287.*

9.7 ASSERTIONS ON INDIVIDUAL FEATURES

We now look in more detail at the contracts governing a single feature. Class invariants, which apply to a whole class, will come next.

Preconditions and postconditions

The declaration of an Attribute_or_routine — deferred routine, effective routine with a **do** or **once** body, external routine, but also an attribute written with the explicit **attribute** keyword — may include a Precondition, a Postcondition or both.

Here is an example of a routine from class *CHAIN* in EiffelBase:

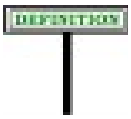
```
put_i_th (v: like first; i: INTEGER)
                -- Put item v at i-th position.
        require
                index_large_enough: i >= 1
                index_small_enough: i <= count
        deferred
        ensure
                not_empty: not is_empty
        end
```

The precondition expresses that no client should call the routine unless the actual argument for *i* is between 1 and *count*. The postcondition expresses that, after a successfully completed execution of the routine, *is_empty* (a boolean function of the same class) will yield false.

Each Assertion_clause of the Precondition and Postcondition has been labeled with a Tag such as index_large_enough.

The contract of a routine

A precondition and postcondition constitute a contract:



Contract, subcontract

Let *pre* and *post* be the precondition and postcondition of a feature *f*. The **contract** of *f* is the pair of assertions [*pre*, *post*].

A contract [*pre'*, *post'*] is said to be a **subcontract** of [*pre*, *post*] if and only if *pre* implies *pre'* and *post'* implies *post*.

Here "implies" is boolean implication. The notion of subcontract is important because it defines the clients' perspective on permissible changes in a routine's implementation. For a client of a class offering a routine

```
rout (…)
        require
                pre
        do
                …
        ensure
                post
        end
```

what counts is the contract defined by the assertions: a client which achieves *pre* at call time is entitled to obtain *post* on return. Another routine *other_rout* may be acceptable as a substitute for *rout* if it still satisfies that contract. This does not necessarily mean that the contract of *other_rout* must be the same as that of *rout*: it may also be a subcontract [*pre'*, *post'*], since in that case any client's call that satisfies the obligation defined by the original contract (*pre*) also satisfies the obligation of the new contract (*pre'*), and the benefits guaranteed by the new contract (*post'*) also entitle the clients to those guaranteed by the original (*post*).

→ "REDECLARATION AND ASSERTIONS", 10.17, page 283 and "Redeclaration rule", page 313.

The constraint on routine redeclaration will ensure that whenever a routine is redeclared (redefined, or effected if the original was deferred), the new specification is a subcontract of the original.

Constraints on routine assertions

For the contracts to be enforceable, preconditions and postconditions must satisfy constraints making them usable by clients.

The first constraint affects preconditions:

Precondition Export rule *VAPE*

A Precondition of a feature *r* of a class *S* is valid if and only if every feature *f* appearing in every Assertion_clause of its unfolded form *u* satisfies the following two conditions for every class *C* to which *r* is available:

1 • If *f* appears as feature of a call in *u* or any of its subexpressions, *f* is available to *C*.

2 • If *u* or any of its subexpressions uses *f* as creation procedure of a Creation_expression, *f* is available for creation to *C*.

If (condition 1) *r* were available to a class *B* but its precondition involved a feature *f* not available to *B*, *r* would be imposing to *B* a condition that *B* would not be able to check for itself; this would amount to a secret clause in the contract, preventing the designer of *B* from guaranteeing the correctness of calls.

More on avoiding secret clauses below.

The rule applies to the *unfolded form* of a precondition, which will be defined as the fully reconstructed assertion, including conditions defined by ancestor versions of a feature in addition to those explicitly mentioned in a redeclared version.

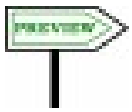
The unfolded form (by relying on the "Equivalent Dot Form" of the expressions involved) treats all operators as denoting features; for example an occurrence of *a* > *b* in an assertion yields *a*.*greater* (*b*) in the unfolded form, where *greater* is the name of a feature of alias ">". The Precondition Export rule then requires, if the occurrence is in a Precondition, that this feature be available to any classes to which the enclosing feature is available.

Condition 2 places the same obligation on any feature *f* used in a creation expression **create** *a*.*f* (…) appearing in the precondition (a rare but possible case). The requirement in this case is "available for creation".

The features mentioned in the constraint may include features of *C* and, for a complex precondition, features of other classes; for example, if the precondition of *r*, includes the expression

```
a.b (c) + d * (e.f (g).h)
```

where *b*, *f*, *h* are features of other classes, all these features must be available to *B* — as well as *a*, *c*, *d*, *plus* **alias** "+", *product* **alias** "*", *e* and *g* if all of these are features of *C*.



In addition to the Precondition Export rule:

- The Entity rule, studied as part of the discussion of entities, further restricts the kind of entities that may appear in an Assertion. For example **Result** may only appear in a postcondition.

→ "Entity rule", page 513.

- If *r* is a creation procedure, the Creation Clause rule puts extra requirements on its precondition, expressed by the Creation Precondition rule and its notion of "creation-valid" precondition clause.

→ "Creation Clause rule", page 548; "Creation Precondition rule", page 547.

For postconditions, there is no rule corresponding to the Precondition Export rule; the Entity rule is sufficient. An Assertion_clause of a Postcondition may — unlike in a Precondition — refer to features with a different availability status. For example, the postcondition for routine *put_right* in class *LINKED_LIST* of EiffelBase includes the clause:

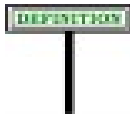
```
next.item = v
```

where *v* is a formal argument of the routine but the query *next* is only available to *LINKED_LIST* itself.

The reason for this difference of treatment between a Precondition and a Postcondition comes from the theory of Design by Contract. In the case of preconditions, as noted, using a secret query would make it impossible for clients to satisfy the contract. But including a secret query in a postcondition causes no harm to clients: they simply will not be able to rely on the corresponding properties, which indeed do not appear in the contract view.

← "-DOCUMENTING THE CLIENT INTERFACE OF A CLASS", 7.9, page 212; .

The difference in export status of the various entities involved in a precondition or postcondition explains the need for the following notion:



Availability of an assertion clause

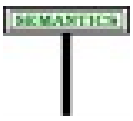
An Assertion_clause *a* of a routine Precondition or Postcondition is **available** to a class *B* if and only if all the features involved in the Equivalent Dot Form of *a* are available to *B*.

This notion is necessary to define interface forms of a class adapted to individual clients, such as the incremental contract view ("short form").

"Old" expression

A special form of expression, the Old expression, is available in routine postconditions only.

→ *Expressions are the topic of chapter 28.*



In a postcondition clause, the expression **old** *exp* has the same type as *exp*; its value at execution time, on routine exit, is the value of *exp* as evaluated on routine entry in the current call.

An example appeared in the postcondition for *forth*. Here is another, from routine *put* in class *FIXED_QUEUE* of EiffelBase; the routine inserts an element into a queue:

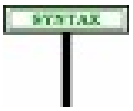
← *Page 230.*

```
put (v: T)
            -- Add item v to queue.
        require
            not_full: not full
        do
            …
        ensure
            count = old count + 1
            (old is_empty) implies (item = v)
            not empty
            array_item ((last – 1 + capacity) // capacity) = v
        end
```

// is the remainder operator on integers.

The highlighted postcondition clause indicates that if the queue was initially empty (**old** *is_empty*) the value at cursor position (given by *item*) will be the one just inserted. Operator **implies** is boolean implication.

The syntax of an Old expression is simply



"Old" postcondition expressions

Old ≜ **old** Expression

The validity constraint expresses that a Postcondition is the only permitted context for an Old expression:

Old Expression rule *VAOX*

An Old expression *oe* of the form **old** *e* is valid if and only if it satisfies the following conditions:

1 • It appears in a Postcondition part *post* of a feature.

2 • It does not involve **Result**.

3 • Replacing *oe* by *e* in *post* yields a valid Postcondition.

Result is otherwise permitted in postconditions, but condition 2 rules it out since its value is meaningless on entry to the routine. Condition 3 simply states that **old** *e* is valid in a postcondition if *e* itself is. The expression *e* may not, for example, involve any local variables (although it might include **Result** were it not for condition 2), but may refer to features of the class and formal arguments of the routine.

The semantic rule follows from the above informal explanations:

Old Expression Semantics, associated variable, associated exception marker

The effect of including an Old expression *oe* in a Postcondition of an effective feature *f* is equivalent to replacing the semantics of its Feature_body by the effect of a call to a fictitious routine possessing a local variable *av*, called the **associated variable** of *oe*, and semantics defined by the following succession of steps:

1 • Evaluate *oe*.

2 • If this evaluation triggers an exception, record this event in an **associated exception marker** for *oe*.

3 • Otherwise, assign the value of *oe* to *av*.

4 • Proceed with the original semantics.

The recourse to a fictitious variable and fictitious operations is in the style of "unfolded forms" used throughout the language description. The reason for these techniques is the somewhat peculiar nature of the Old expression, used at postcondition evaluation time, but pre-computed (if assertion monitoring is on for postconditions) on entry to the feature.

The matter of exceptions is particularly delicate and justifies the use of "associated exception markers". If an Old expression's evaluation triggers an exception, the time of that exception — feature entry — is not the right moment to start handling the exception, because the postcondition might not need the value. For example, a postcondition clause could read

```
((x /= 0) and (old x /= 0)) implies ((((1 / x) + (1 / (old x)))) = y)
```

If *x* is 0 on entry, **old** *x* /= 0 will be false on exit and hence the postcondition will hold. But there is no way to know this when evaluating the various Old expressions, such as 1 / **old** *x* on entry. We must evaluate this expression anyway, to be prepared for all possible cases. If *x* is zero, this may cause an arithmetic overflow and trigger an exception. This exception should not be processed immediately; instead it should be remembered — hence the associated exception marker — and triggered only if the evaluation of the *postcondition*, on routine exit, attempts to evaluate the associated variable; hence the following rule.

The "associated variable" is defined only for effective features, since a deferred feature has no Feature_body. If an Old expression appears in the postcondition of a deferred feature, the rule will apply to effectings in descendants through the "unfolded form" of the postconditions, which includes inherited clauses.

Like any variable, the associated variable *av* of an Old expression raises a potential initialization problem; but we need not require its type to be self-initializing since the above rule implies that *ov* appears in a Certified Attachment Pattern that assigns it a value (the value of *oe*) prior to use.

→ *"Self-initializing type"*, *page 515*.

There remains to define precisely the value of the "associated variable":

Associated Variable Semantics

As part of the evaluation of a postcondition clause, the evaluation of the associated variable of an Old expression:

1 • Triggers an exception of type *OLD_EXCEPTION* if an associated exception marker has been recorded.

2 • Otherwise, yields the value to which the variable has been set.

"Only" clause

With the Old expression we have a way to express the effect of a routine by specifying how some properties of the object after the routine's execution relate to values captured before the execution.

This technique can be used in particular to express that the values of some queries (attributes or functions) do *not* change through the application of the routine; it suffices to use a postcondition of the form

```
ensure
        q = old q
        r = old r
        … Other similar clauses …
        … Other postcondition clauses …
```

Although it does the job, this technique has some disadvantages:

- Often, a routine will change only a few queries. In the above style, its postcondition will have many "similar clauses", one for each query that it doesn't affect.
- You have to go through the class, for each routine, to make sure you don't forget any such queries.
- When you add a new query unrelated to existing routines, you have to add a "similar clause" to every one of these routines!

- Things become even more messy with inheritance: if you add a new query *s* in a descendant and do not redefine a feature *f*, most likely *f* will not modify *s*, but its original postcondition obviously cannot state that; to express this property, you would have to redefine *f* for the sole purpose of adding the clause *s* = **old** *s* to its postcondition!

→ *In the form of an ensure then clause; see "REDECLARATION AND ASSERTIONS", 10.17, page 283.*

These observations show that along with the mechanism for expressing how a routine affects the value of certain queries, we need a complementary notation to express what queries it does **not** affect. The **only** postcondition clause achieves this goal. In a postcondition you may include one (and only one) clause of the form

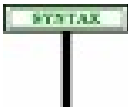
```
only q, r, s
```

to state that *all queries other than q, r, s* are left untouched by the enclosing routine. In other words, this is an abbreviation for postcondition clauses

```
q1 = old q1
…
qn = old qn
```

where q_1, …, q_n are all the queries other than *q, r, s*. The "unfolded form" defining the validity and semantics of an Only clause will express this.

The syntax is straightforward:



"Only" postcondition clauses

Only ≜ **only** [Feature_list]

The syntax of assertions indicates that an Only clause may only appear in a Postcondition of a feature, as its last clause.

← *Construct* Postcondition, *page 232.*

Those other postcondition clauses let you specify how a feature *may* change specific properties of the target object, as expressed by queries. You may also want — this is called the **frame problem** — to restrict the scope of features by specifying which properties it *may not* change. You can always do this through postcondition clauses *q* = **old** *q*, one for each applicable query *q*. This is inconvenient, not only because there may be many such *q* to list but also, worse, because it forces you to list them all even though evolution of the software may bring in some new queries, which will not be listed. Inheritance makes matters even more delicate since such "frame" requirements of parents should be passed on to heirs.

An Only clause addresses the issue by enabling you to list which queries a feature may affect, with the implication that:

- Any query *not* listed is left unchanged by the feature.
- The constraints apply not only to the given version of the feature but also, as enforced by the following rules, to any redeclarations in descendants (specifically, to their effect on the queries of the original class).

The syntax allows omitting the Feature_list; this is how you can specify that the routine must leave *all* queries unchanged (it is then known as a "*pure*" routine).

To express the validity and semantics of an Only clause, we must clarify how inheritance affects it. What does our example, **only** *q, r, s* appearing in the postcondition of a feature *f* of a class *C*, mean in a proper descendant *D*, which may add its own queries? Two cases are possible:

- *D* does not redeclare *f*. Then we must understand **only** *q, r, s* as we did in *C*: *f* may modify no query of *D*, whether from *C* or new in *D*.
- We may want to allow *f* to modify some features of *D* — say *t* and *u* — in addition to *q*, *r* and *s*. In that case we'll redeclare *f* with a new postcondition clause, introduced by **ensure then**, of the form **only** *t*, *u*. Such an Only clause appearing in a declaration means: "the feature may only affect, *among queries introduced since any preceding* Only *clause*, the queries *t* and *u*". The effect then is cumulative, as if there had been a single Only clause of the form **only** *q, r, s, t, u*.

Here are the rules stating these properties. First, the validity:

Only Clause rule *VAON*

An Only clause appearing in a Postcondition of a feature of a class *C* is valid if and only if every Feature_name *qn* appearing its Feature_list if any satisfies the following conditions:

1 • There is no other occurrence of *qn* in that Feature_list.

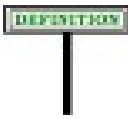
2 • *qn* is the final name of a query *q* of *C*, with no arguments.

3 • If *C* redeclares *f* from a parent *B*, *q* is not a feature of *B*.

Another condition, following from the syntax, is that an Only clause appears at the last element of a Postcondition; in particular, you may not include more than one Only clause in a postcondition.

← *Construct* Postcondition, *page* 232.

First, two useful notions of "unfolding". First we need to include inherited features:



Unfolded feature list of an Only clause

The **unfolded feature list** of an Only clause appearing in a Postcondition of a feature *f* in a class *C* is the Feature_list containing:

1 • All the feature names appearing in its Feature_list if any.

2 • If *f* is the redeclaration of one or more features, the final names in *C* of all the features whose names appear (recursively) in their unfolded Only clauses.

For an immediate feature (a feature introduced in *C*, not a redeclaration), the purpose of an Only clause of the form

only *q, r, s*

is to state that *f* may only change the values of queries *q*, *r*, *s*.

In the case of a redeclaration, previous versions may have had their own Only clauses. Then:

- If there was already an Only clause in an ancestor *A*, the features listed, here *q, r* and *s*, must be new features, not present in *A*. Otherwise specifying **only** *q*, *r*, *s* would either contradict the Only clause of *A* if it did not include these features (thus ruling out any modification to them in any descendant), or be redundant with it if it listed any one of them.
- The meaning of the Only clause is that *f* may only change *q*, *r* and *s in addition* to inherited queries that earlier Only clauses allowed it to change.

Note that this definition is mutually recursive with the next one.

The notion of unfolded feature list enables us to interpret an Only clause as a sequence of postcondition clauses asserting that the feature — double negation! — does *not* change any of the the *non*-listed features from the current class:

Unfolded Only clause

The **unfolded Only clause** of a feature *f* of a class *C* is a sequence of Assertion_clause components of the following form, one for every argument-less query *q* of *C* that does not appear in the unfolded feature list of the Only clause of its Postcondition if any:

q = (**old** *q*)

This will make it possible to express the semantics of an Only clause through a sequence of assertion clauses stating that the feature may change the value of no queries except those explicitly listed.

Note the use of the equal sign: for a query *q* returning a reference, the Only clause states (by *not* including *q*) that after the feature's execution the reference will be attached to the same object as before. That object might, internally, have changed. You can still rule out such changes by listing in the Only clause other queries reflecting properties of the object's *contents*.

9.8 CLASS INVARIANTS

The next category of assertion use is the class invariant, determined by the last clause of a class text, Invariant. Unlike a Precondition or Postcondition which characterizes a single feature, the class invariant applies to an entire class — more precisely, to all its exported features.

The invariant specifies properties which any instance of the class must satisfy at every instant at which the instance is observable by clients.

The class *CHAIN* quoted above has the following Invariant clause (with assertion tags removed from brevity):

```
deferred class CHAIN feature
…
invariant
        -- Definitions:
                empty = (count = 0)
                off = ((position = 0) or (position = count + 1))
                isfirst = (position = 1)
                islast = (not empty and (position = count))
        -- Axioms:
                count >= 0
                position >= 0; position <= count + 1
                empty => (position = 0)
                (not off) implies (item = i_th (position))
        -- Theorems:
                (is_first or is_last) implies (not empty)
end
```

As an example, the first Assertion_clause states that in all observable states the result of *empty* (a boolean function) called on a chain is true if and only if the value of *count* called on the same chain is zero. The second and third of those marked as Axioms state that the value of *position*, an integer attribute, always remains between 0 and the value of *count* plus one.

This Invariant has been divided into "Definitions", expressing that certain queries may be defined in terms of others, "Axioms", expressing constraints on the features, and "Theorems", expressing properties which may be deduced from other clauses. This classification helps for readability but has no semantic consequence.

As already noted, the semantics of a class's invariant — to define correctness and specify assertion monitoring — doesn't just rely on the invariant clauses listed in the class itself but includes ancestors' invariants, through the notion of "unfolded form". The unfolded form is the concatenation of the clauses of its parents's invariants, themselves unfolded in the same way, and its own clauses.

This ensures the consistency of the type view of inheritance. In particular, inheritance permits substitution of instances: if *C* is a descendant of *B*, instances of *C* will also be instances of *B*. But the soundness of this notion requires any semantic obligation on instances of *B*, as expressed by *B*'s invariant, to be also applicable to instances of *C*. Hence the definition of the invariant of a class as including all invariant clauses from ancestors.

← *On the two views of inheritance see "OVERVIEW", 6.1, page 169.*

9.9 THE CONSISTENCY OF A CLASS

Invariants and the notion of routine contract make it possible to define the consistency of a class, part of its *correctness*, one of its two characteristic semantic properties.

A class will be said to be consistent if its implementation satisfies the correctness requirements expressed by the preconditions on the routines of the class, the postconditions, and the class invariant.

The following notations serve to define this notion precisely. They are not part of Eiffel, but mathematical conventions used to talk **about** Eiffel classes and their semantic properties.

WARNING: These are mathematical notations, not Eiffel text.

- If *r* is a routine, do_r denotes its body, pre_r its precondition, and $post_r$ its postcondition.
- If *C* is a class, INV_C denotes its class invariant.

- If P and Q are assertions and A is an instruction or compound, the notation $\{P\}\ A\ \{Q\}$ expresses the property that whenever A is executed in a state in which P is true, the execution will terminate in a state in which Q is true. This is a standard concept from the theory of programming languages, taken here in a "total correctness" meaning:

Hoare triple notation (total correctness)

In definitions of correctness notions for Eiffel constructs, the notation $\{P\}\ A\ \{Q\}$ (a mathematical convention, not a part of Eiffel) expresses that any execution of the Instruction or Compound A started in a state of the computation satisfying the assertion P will terminate in a state satisfying the assertion Q.

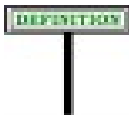
If P, A and Q are extracted from the text of a routine with arguments, the Hoare triple $\{P\}\ A\ \{Q\}$ will be considered to hold if and only if it holds for all possible values of the formal arguments.

Earlier conventions enable us to assume that every Attribute_or_routine has both a Precondition and a Postcondition, and that every Class has an Invariant, by considering any missing clause as an implicit form for **require True**, **ensure True** or **invariant True**. In addition, we will take inheritance into account by considering the *unfolded forms* of these assertions, integrating inherited properties.

← "Precondition, postcondition, invariant", page 234.

→ "Unfolded form of an assertion", page 287.

These conventions make it possible to define class consistency:



Class consistency

A class C is **consistent** if and only if it satisfies the following conditions:

1 • For every creation procedure p of C:

$$\{pre_p\}\ do_p\ \{INV_C \textbf{ and then } post_p\}$$

2 • For every feature f of C exported generally or selectively:

$$\{INV_C \textbf{ and then } pre_f\}\ do_f\ \{INV_C \textbf{ and then } post_f\}$$

where INV_C is the invariant of C and, for any feature f, pre_f is the unfolded form of the precondition of f, $post_f$ the unfolded form of its postcondition, and do_f its body.

The mathematical symbol ∧ represents boolean conjunction.

Class consistency is one of the most important aspects of the *correctness* of a class: adequation of routine implementations to the specification. The other aspects of correctness, studied below, involve Check instructions, Loop instructions and Rescue clauses.

→ "Correctness (class)", page 253.

9.10 CHECK INSTRUCTIONS

Another use of assertions is the Check instruction, of the form

```
check
        "Some_assertion"     -- May include zero or more clauses
note -- This part is optional
        Tag: "Explanation"
end
```

This lets you express that a certain property, captured by Some_assertion, will be satisfied whenever execution reaches the instruction. As the name indicates, the instruction is there to require that some mechanism "check" that the property indeed holds. The mechanism in question may be a human reader, a mechanical program prover (if possible), or the execution itself. (We have not seen yet what effect, if any, assertion constructs may have on execution, and must wait a few more sections for an answer.)

→ *The reader who does not want to wait may preview "RULES OF RUN-TIME ASSERTION MONITORING", 9.13, page 253.*

A common use of a Check instruction is just before a routine call, to express one or both of two properties without which the call's execution couldn't make sense:

- For a Qualified_call, the target is attached (not void).
- The routine's precondition is satisfied.

As an example of the first case, the body of procedure *remove_left* in the class *LINKED_LIST* of EiffelBase contains the following extract:

```
previous := item (position – n – 1)
        check
                previous /= Void
        note
                why: "Value at position – n – 1 cannot be void"
        end
previous.put_right (active)
```

In that class, *item* (*i*) yields the element at position *i*. The Check instruction expresses that the value at *position – n – 1* is not void, and so can be used as the target of the call to *put_right* that follows.

The **note part** is optional but recommended: as here, it lets you explain, through a why entry, the reasoning that leads you to expect that the assertion will hold.

← *"ANNOTATING A CLASS", 4.8, page 122.*

Another style recommendation illustrated here is to indent the Check, to separate it from instructions that perform actual steps of the algorithm.

As an example of the second case, the body of procedure *put* in the same class contains:

```
if i = 1 then
        lt.put_right (first_element)
else
        check
                i – 1 >= 1
                i – 1 <= count
        note
                why: "See condition of if and invariant"
        end
        left_neighbor := item (i – 1)
        …
end
```

This should be understood in light of the precondition for *item* (*j*):

```
require
        index_large_enough: j >= 1
        index_small_enough: j <= count
```

The Check instruction expresses that the call to *item* satisfies the precondition thanks to the context (the Conditional instruction) and the invariant.

More generally, a Check instruction is useful in the following situation. You know that the proper execution of a certain computation requires some consistency condition (such as a call's target being attached, or a precondition satisfied). To do your job properly, you so design the context of the computation as to guarantee the desired condition. If your reasoning and its consequence (that the consistency condition will be satisfied) are not obvious to a reader of the software text, a Check instruction will clarify that you did not overlook your obligations.

In the presence of assertion monitoring as discussed below, the Check will also help detect the mistake if you did make one after all.

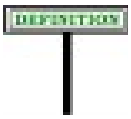
The general form of a Check instruction is:

Check instructions

Check ≙ **check** Assertion [Notes] **end**

The Notes part is intended for expressing a formal or informal justification of the assumption behind the property being asserted.

The following definition expresses the semantics:



Check-correct

An effective routine *r* is **check-correct** if, for every Check instruction *c* in *r*, any execution of *c* (as part of an execution of *r*) satisfies its Assertion.

9.11 LOOP INVARIANTS AND VARIANTS

Our next application of assertions uses them to guarantee the correctness of loops. It uses an assertion, the **loop invariant**, as well as an integer expression, the **loop variant**. The variant is not syntactically an assertion, but plays a closely related role.

The invariant determines the properties ensured by the loop on exit; the variant guarantees that the loop's execution terminates.

Here is an example of these constructs in a loop from routine *search_child* (which looks for a certain node among the children of the current node) in the EiffelBase class *LINKED_TREE*:

```
from
        go_before
invariant
        0 <= child_position; child_position <= arity + 1
until
        child_after or else (j = i)
loop
        child_forth
        if (sought = child) then j := j + 1 end
variant
        arity – child_position + 1
end
```

This discussion refers to instructions of a Loop: the **from** clause or Initialization, executed once at the start of the loop; the iteration or **loop** clause, executed zero or more times until the Exit_condition, introduced by **until**, holds.

→ "LOOP", 17.7, page 494.

The invariant expresses a property of *child_position*, the index of the child being looked at: *child_position* will remain between 0 and *arity* (the number of children) plus one. Stating that this property is a loop invariant means asserting that the Initialization ensures it, and that every iteration preserves it. This is indeed the case:

- The Initialization, *go_before*, moves the child cursor to the node's first child, or to position 0 if there is no child.

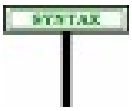
- In the iteration, *child_forth*, which moves the child cursor right by one position, is only executed when the property *child_after* is not satisfied (since this condition is part of the Exit_condition).

The variant is an integer expression, *arity – child_position + 1*, which is always non-negative and decreases on every iteration. This guarantees that the loop will terminate.

Here now are the precise rules governing these constructs, starting with the syntax. Invariant and Variant appear in the syntax of Loop, appearing in the chapter on control structures. A loop invariant is a specimen of Invariant, also applicable to class invariants and given earlier in this chapter. A loop Variant (which contains an integer expression, not an assertion) has the following syntax:

→ *Page 495.*

← *Page 232.*



Variants

Variant ≜ **variant** [Tag_mark] Expression

The optional Tag_mark labels the variant in the same way as for an Assertion_clause. The variant must satisfy a simple constraint:

Variant Expression rule *VAVE*

A Variant is valid if and only if its variant expression is of type *INTEGER* or one of its sized variants.

The sized variants are *INTEGER_X* and *NATURAL_X* where *X* is an explicit length, for example *INTEGER_16*.

→ *"INTEGERS", 30.6, page 820.*

Associated terminology:

Loop invariant and variant

The Assertion introduced by the Invariant clause of a loop is called its **loop invariant**. The Expression introduced by the Variant clause is called its **loop variant**.

Semantically, the invariant *INV* of a correct loop satisfies two properties:

- The loop's Initialization (**from** clause) ensures the truth of *INV*.
- Any iteration started in a state that does not satisfy the Exit_condition but satisfies *INV* terminates with *INV* true again.

As a result of these properties, the invariant will still be satisfied on loop exit, at which point the Exit_condition will hold. Their conjunction is the output condition of the loop. From a theoretical viewpoint, the goal of the loop is to achieve this condition, and the looping process reaches it by successive approximations.

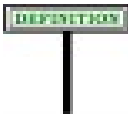
You may view the variant as an estimate of the remaining distance to the goal. The variant *VAR* of a correct loop satisfies two properties:

- The Initialization sets *Var* to a non-negative value.
- Any iteration started in a state that does not satisfy the Exit_condition decreases the value of *VAR* while keeping it non-negative.

As a result of these properties, since the variant is an integer expression, the iterations may not go on forever.

The following definition captures this correctness semantics:

← *For the Hoare triple notation {P} A {Q} see page 247 above.*



Loop-correct

A routine is **loop-correct** if every loop it contains, with loop invariant *INV*, loop variant *VAR*, Initialization *INIT*, Exit condition *EXIT* and body (Compound part of the Loop_body) *BODY*, satisfies the following conditions:

1 • {**true**} *INIT* {*INV*}

2 • {**true**} *INIT* {*VAR* ≥ 0}

3 • {*INV* **and then not** *EXIT*} *BODY* {*INV*}

4 • {*INV* **and then not** *EXIT* **and then** (*VAR* = *v*)} *BODY* {0 ≤ *VAR* < *v*}

Conditions 1 and 2 express that the initialization yields a state in which the invariant is satisfied and the variant is non-negative. Conditions 3 and 4 express that the body, when executed in a state where the invariant is satisfied but not the exit condition, will preserve the invariant and decrease the variant, while keeping it non-negative. (*v* is an auxiliary variable used to refer to the value of *VAR* before *BODY*'s execution.)

In keeping with the general use of the word "contract" — for features, for classes — the invariant and variant of a loop (only the former an assertion) are said to define the contract of the loop.

9.12 THE CORRECTNESS OF A CLASS

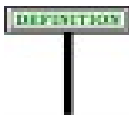
In connection with the various uses of assertions, we have seen how a class may be "correct" in several partial ways:

- Consistent: every feature satisfies its precondition and postcondition, every creation procedure ensures the invariant, every exported feature preserves the invariant.
- Check-correct: the assertion of every Check instruction holds.
- Loop-correct: loops preserve their invariants and decrease their variants.

Exceptions will lead to one more variant: a class is *exception-correct* if in the case of a non-retried exception, leading to a failure of the enclosing routine, the "rescue clause" will restore the class invariant.

→ "EXCEPTION CORRECTNESS", 26.9, page 702.

The combination of these properties yields the full notion of correctness:



Correctness (class)

A class is **correct** if and only if it is consistent and every routine of the class is check-correct, loop-correct and exception-correct.

Do not confuse correctness and validity. Correctness is a semantic notion, expressing that the implementation of a class matches its specification. Validity is simply the property that a construct is well-formed, such as a routine call with the proper number and type of actual arguments. The correctness of a software element is a meaningless notion unless the element is valid.

← "CORRECTNESS", 2.10, page 99.

Ideally, an Eiffel environment should come with tools which can prove or disprove the correctness of a class as defined here. This is still beyond the reach of most of today's environments, although the technology is progressing quickly. To understand classes and contracts, however, it is essential to refer to this notion of class correctness, even if you have to assess correctness by manual examination of the class rather than through proof tools.

As the next best thing to proofs, Eiffel environments must support run-time monitoring of contracts, as described next.

9.13 RULES OF RUN-TIME ASSERTION MONITORING

Complementing the notion of class correctness, run-time monitoring provides the second facet of assertion semantics.

Contracts, as expressed through assertions and loop variants, express correctness requirements on software texts. If we view them not just as specification (as in the definition of class correctness) but as language constructs, they raise the same semantic question as any other construct: what's the run-time effect? In the execution of a *correct* system — the only case that, in principle, matters — every assertion will always be satisfied at the times when it has to: a precondition on every call to its routine, a postcondition on return and so on. So in theory it should not be necessary to define the semantics of assertions evaluation: for a correct class, the effect of evaluating an assertion is irrelevant, since the value is always true! This may be called the paradox of assertion semantics.

The paradox is only theoretical. In practice, short of proofs as discussed above, one must often accept the prospect that a system may contain errors — may not be correct. Then assertions provide crucial help in detecting these errors and suggesting corrections. By directing the run-time system to evaluate assertions and variant clauses, and to trigger an exception if it detects a violation, you check the consistency between what the software does (the feature implementations) and what you think it does (the contracts): you let the tools call your bluff. This gives a remarkable tool for debugging, testing and maintaining software systems.

Eiffel implementations are indeed required to provide, usually through a compilation option, mechanisms to evaluate contract elements — assertions and loop variants — during the execution of a system. We now review the details of this requirement.

Associated boolean expression

We must first clarify what it means to evaluate an assertion. As a language construct, the notion of assertion derives from boolean expressions. Like a boolean expression, an assertion describes a property that, for the computation captured in a certain state, is true or false.

Two possibilities available for writing assertions — specifically, postconditions — do not exist in boolean expressions: Old expressions and Only clauses. Their semantics, however, does reduce in the end to that of boolean expressions thanks to the associated "unfolded forms":

- **old** *exp* stands for an extra variable (the "associated variable" of the Old expression) that would have been initialized to the value of *exp* on entry to the routine.

← ""Old" expression", page 239.

- **only** $q_1, \ldots q_n$ stands for a set of assertions of the form t = **old** t, one for every query t that is not one of the q_n.

← ""Only" clause", page 241.

We can capture these equivalences through a notion of unfolded form:

Local unfolded form of an assertion

The **local unfolded form** of an assertion *a* — a Boolean_expression — is the Equivalent Dot Form of the expression that would be obtained by applying the following transformations to *a* in order:

1 • Replace any Only clause by the corresponding unfolded Only clause.

2 • Replace any Old expression by its associated variable.

3 • Replace any clause of the Comment form by True.

The unfolded form enables you to understand an assertion, possibly with many clauses, as a single boolean expression. The use of **and then** to separate the clauses indicates that you may, in a later clause, use an expression that is defined only if an earlier clause holds (has value true).

This unfolded form is "local" because it does not take into account any inherited assertion clauses. This is the business of the full (non-local) notion of unfolded form of an assertion, introduced in the discussion of redeclaration.

→ *"Unfolded form of an assertion", page 287.*

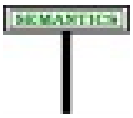
The Equivalent Dot Form of an expression removes all operators and replaces them by explicit call, turning for example *a* + *b* into *a*.*plus* (*b*). This puts the result in a simpler form used by later rules.

If an Only clause is present, we replace it by its own unfolded form, a sequence of Assertion_clause components of the form *q* = **old** *q*, so that we can treat it like other clauses for the assertion's local unfolded form. Note that this unfolding only takes into account queries explicitly listed in the Only clause, but not in any Only clause from an ancestor version; inheritance aspects are handled by the normal unfolding of postconditions, applicable after this one according (as noted above) to the general notion of unfolded form of an assertion

The syntax permits a Comment as Unlabeled_assertion_clause. Such clauses are useful for clarity and documentation but, as reflected by condition 3, cannot have any effect on run-time monitoring.

← *Page 232.*.

Assertion monitoring



An Eiffel environment must make it possible to evaluate assertions — that is to say, the boolean expressions resulting from their unfolding as just defined — during execution.

For a correct system, as noted, such evaluation will have no effect on execution semantics, except through possible side effects of the functions called by assertions.

For an incorrect system, if an assertion evaluates to true, it has no further effect on the outcome of the computation. If it evaluates to false, it will trigger an exception, disrupting the normal flow of computation.

→ *Chapter 26 discusses exceptions.*

The first notion is *evaluation* of an assertion or loop variant. For a variant — an integer expression — the meaning is clear; for assertions we must be more precise:

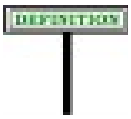
Evaluation of an assertion

To **evaluate** an assertion consists of computing the value of its unfolded form.

This defines the value of an assertion in terms of the value of a boolean expression, as given by the discussion of expressions.

→ "Expression Semantics (strict case)", page 773 and "Operator Expression Semantics (semistrict cases)", page 777

Should assertions be evaluated? Not necessarily. You may define various levels of *monitoring*, including no evaluation at all:



Assertion monitoring

The execution of an Eiffel system may evaluate, or **monitor**, specific kinds of assertion, and loop variants, at specific stages:

1 • Precondition of a routine *r*: on starting a call to *r*, after argument evaluation and prior to executing any of the instructions in *r*'s body.

2 • Postcondition of a routine *r*: on successful (not interrupted by an exception) completion of a call to *r*, after executing any applicable instructions of *r*.

3 • Invariant of a class *C*: on both start and termination of a *qualified* call to a routine of *C*.

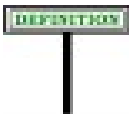
4 • Invariant of a loop: after execution of the Initialization, and after every execution (if any) of the Loop_body.

5 • Assertion in a Check instruction: on any execution of that instruction.

6 • Variant of a loop: as with the loop invariant.

Unlike other semantic definitions of this book the rule is in "may" rather than "must" mode, describing run-time checks that you will wish to enable or disable depending on the circumstances. The possibilities are detailed next.

The following definition describes how assertion monitoring can catch cases of software incorrectness:



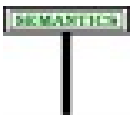
Assertion violation

An **assertion violation** is the occurrence at run time, as a result of assertion monitoring, of any of the following:

- An assertion (in the strict sense of the term) evaluating to false.
- A loop variant found to be negative.
- A loop variant found, after the execution of a Loop_body, to be no less than in its previous evaluation.

To simplify the discussion these cases are all called "*assertion* violations" even though a variant is not technically an assertion.

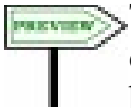
Assertions affect the semantics of system execution only in the case of assertion violations:



Assertion semantics

In the absence of assertion violations, assertions have no effect on system execution other than through their evaluation as a result of assertion monitoring.

An assertion violation causes an exception of type *ASSERTION_VIOLATION* or one of its descendants.



The result of the exception is detailed in the corresponding chapter. Any exception, when it occurs, interrupts the execution of some routine that has been started but not yet finished: the *recipient* of the exception. In accordance with the principles of Design by Contract:

→ *"SEMANTICS OF EXCEPTION HANDLING", 26.10, page 702*

- For a precondition violation, the recipient is the caller: it has not observed its obligations, and any attempt to execute the routine would be meaningless, and perhaps harmful. In this case the error — the *bug* — is on the client side.
- For a postcondition violation, the recipient is the routine itself: it is unable to fulfill its obligations. The bug is on the supplier side.

The definition of "exception cases", part of exception semantics, provides the full specification for these two kinds of violation and all others.

→ *Page 706.*

Levels of assertion monitoring

As noted for the definition of assertion monitoring, the execution "may" monitor exceptions but does not always have to do so. For a program that is known to be correct — through other means, for example a *proof* that it meets its contracts — you may prefer to avoid the overhead of monitoring. An Eiffel development environment, however, must offer you at least basic monitoring facilities.

A number of levels of monitoring are predefined:

Implementations may — and usually do — provide more combinations than the four required by the last part of the rule.

It is common to provide *precondition* checking only (variant 2) as the default. Checking preconditions on routine entry avoids disasters, since a Routine_body might attempt to perform erroneous or impossible actions when executed in a state which does not satisfy the routine's precondition, but normally causes only a modest performance penalty. More extensive monitoring is especially useful for quality assurance, maintenance, debugging, testing and regression analysis.

Assertion monitoring levels

An Eiffel implementation must provide facilities to enable or disable assertion monitoring according to some combinations of the following criteria:

- Statically (at compile time) or dynamically (at run time).
- Through control information specified within the Eiffel text or through outside elements such as a user interface or configuration files.
- For specific kinds as listed in the definition of assertion monitoring: routine preconditions, routine postconditions, class invariants, loop invariants, Check instructions, loop variants.
- For specific classes, specific clusters, or the entire system.

The following combinations must be supported:

1 • Statically disable all monitoring for the entire system.

2 • Statically enable precondition monitoring for an entire system.

3 • Statically enable precondition monitoring for specified classes.

4 • Statically enable all assertion monitoring for an entire system.

Invariant and qualified calls

As noted in the definition of assertion monitoring, a class invariant will only be checked for **qualified** calls. If *r* is a routine of a class *C*, a call in dot-notation is qualified if it is of the form

← *Case 3, page 256.*

```
a.r (…)
```

with an explicit target, here *a*.

An operator expression using a binary operator, such as *a* + *b*, is also a qualified call since it is an abbreviation for a call *a*.*plus* (*b*), assuming a feature *plus* **alias** "+". Since the difference is of syntax only, we can limit ourselves here to the case of calls with an explicit dot.

→ *"THE EQUIVALENT DOT FORM", 28.8, page 780. Chapter 23 discusses qualified and unqualified calls in details.*

Whether it appears in the text of a routine of *C* or in another class, such a qualified call will cause the invariant to be evaluated (both before and after the call).

A routine of *C* may also contain an **unqualified** call, written just

```
r (…)
```

where the target is not *a* any more but the current object. Such an unqualified call will not cause an invariant check.

→ *"Current object, current routine", page 649*.

The form *Current*.*r* (…), which has the same semantics in the absence of assertions, is qualified and so will trigger an invariant check.