Precondition Enforcement Analysis for Quality Assurance

Nadja Beeli

Submitted to the degree of
Master of Science ETH in Computer Science

Supervised by Prof. Dr. Bertrand Meyer and Dr. Karine Arnout
April - October 2004

Abstract

The crash of Ariane 5 dramatically showed the importance of correctness in software and that the goal to produce reliable software has not yet been achieved. Therefore, this master thesis targets the development of a static analysis, which ensures preconditions and thus enhances a sound reuse of software.

As a result, we could determine many preconditions to be fulfilled, especially preconditions of a certain class, which are used most often. This confirms that static analysis is justified in a development process of quality software.

Acknowledgements

I would like to deeply thank Dr. Karine Arnout for her support and explanations on the subject of the thesis, and her prompt answers. Furthermore I thank Prof. Dr. Bertrand Meyer, who gave me the opportunity to accomplish my master thesis in the field of Design by Contract, and for his introduction on the static analysis of preconditions.

A special thank goes to Éric Bezault, who introduced me to GOBO Eiffel, and swiftly answered my questions.

Table of Contents

Introduction

In Eiffel, Design by Contract [1] plays a central role in development of quality software. Contracts are the fundamental ingredient for the semantic description of software interfaces, explicitly regulating all the functionality, which a component offers to and consumes from the environment. Thereby, all dependencies between software components, be it procedures, functions, objects etc., must be solely defined by the interfaces. In this respect, interfaces both enforce obligations and offer guarantees to the participating components. Whereas Eiffel contracts hereby focus on the semantic specification, the syntactical contracts are manifested in the signatures of routines (ordered parameters with types and a return type for functions) as well as the features (attributes and routines) of classes.

The purpose of Design by Contract is the establishment of programming with clearly declared dependencies. It is more than pure program documentation, enabling language-institutionalized specification that can be verified by both man and machine. This inherently helps improving program quality and sets the basis for safe reuse of software components.

The concrete representation of Eiffel contracts is in terms of assertions, appearing in the role of a routine preconditions, postconditions or class invariants. Contracts are usually checked at runtime for testing and debugging reasons. When a contract rule is violated and not handled by the exception mechanism, the program immediately aborts with an exception. For release versions, the contract monitoring turned off, since the program is assumed to be correct. Even after exhaustive dynamic testing, we can never be convinced that every possible case has been tested and that the program is entirely correct.

A static analysis determines at compile time, whether the contracts hold in any case of an execution scenario. Furthermore, performance for dynamic testing can be enhanced by elimination of redundant runtime checks. However, a static analysis is conservative, i.e. the analysis sometimes cannot determine that a condition is fulfilled, although the condition holds de facto. Consequently, it occurs that contracts are undecidable to be fulfilled, although they are always kept. While time can be saved for dynamic testing, it has to be spent on the static analysis. However, the reward of a successful static analysis is unequally higher, because the contract is then guaranteed to permanently hold. Therefore, a hybrid approach is justified, which makes as great use of a static analysis as possible, and applies manual or dynamic testing for the remaining undecidable cases.

In this thesis, we focus on a static analysis for preconditions since in reuse of libraries, it is particularly important for a client to assure a routine's proper use. Consequently, we aim to determine for as many preconditions to be fulfilled as possible with a static analysis. This could be achieved considering the fact that the vast majority of precondition clauses consist of primitive expressions. This way a core problem of object-orientation could be addressed, being the conceptually dangling references, i.e. void pointers. A supplying routine often ensures in the precondition that an object reference must not be void. As a result, the developed static analyzer is particularly successful in detecting redundant nil checks.

Our analyzer also supports more general precondition expressions, but the results are not as rich for different reasons. On the one hand, there are not that many appropriately complex preconditions and on the other hand, there are such complex expressions, which would require an extensive analysis beyond the scope of this thesis.

As a further contribution, we provide a graphic tool showing in two separate enumerations, the calls that could not be determined to be fulfilled as well as the fulfilled calls. With these steps, one can ensure that software gets more and more reliable.

Chapter 1 - Concept of Contracts

1.1. The Crash of Ariane 5

„On 4 June 1996, the maiden flight of Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700m, the launcher veered off its flight path, broke up and exploded." (Report of the inquiry board [4])

The $500-million crash of the first Ariane 5 launcher, dramatically illustrated what unsafe code reuse can cause. The stirring error happened in the Inertial Reference System (SRI), which measures the attitude of the launcher and its movements in space. A conversion from a 64-bit-floating-point number (the missions "horizontal bias") to a 16-bit signed integer produced a runtime exception, because the number was not representable with 16 bits. The exception was not handled and the uncaught exception was processed in the On-Board computer like flight data, causing a failure and subsequently the crash of Ariane 5.

The SRI software actually worked well with Ariane 4, so the question was what caused the catastrophe with Ariane 5. The reason is the higher horizontal velocity of Ariane 5, resulting in an overflow. Furthermore, the alignment function of the SRI should already have been turned off, because "this software module computes meaningful results only before lift-off. As soon as the launcher lifts off, this function serves no purpose". This functionality was used in early versions of Ariane, to enable quick re-initialization in case of a hold in the count-down, but has fallen into oblivion.

1.2. Design by Contract in Context of Ariane 5

With an extensive use of Design by Contract, such an error would have been prevented. First, the conversion error would not have happened, if the precondition and the limit for the horizontal bias's bound would have been correctly set as outlined in Figure 1-1, where `Maximal_horizontal_bias` is set, and the precondition for the conversion operation is `horizontal_bias <= Maximal_horizontal_bias`. Second, the additional feature, which was responsible, that the alignment function has not been turned off, would have been properly documented as well and therefore not used for Ariane 5. This is also sketched in Figure 1-1, where the limits of the horizontal velocity for the alignment function are expressed with the precondition `0 <= actual_horizontal_velocity` and `actual_horizontal_velocity < Maximal_horizontal_velocity`.

The program has still to be checked with Ariane 5 requirements varying from Ariane 4, either by dynamic or manual testing. It would be a great advantage to verify such preconditions by a static analysis giving absolute guarantees.

```
…
feature -- Constants
   Maximal_horizontal_bias: INTEGER is 32767   -- = 2^15 – 1
         -- Limit of maximal horizontal bias

   Maximal_horizontal_velocity: INTEGER is 4122
         -- Limit of maximal horizontal velocity for
         -- alignment function

feature -- Conversion
   convert_horizontal_bias (horizontal_bias: DOUBLE): INTEGER is
         -- Convert `horizontal_bias' to `INTEGER'.
      require
         bias_limited: horizontal_bias <= Maximal_horizontal_bias
      do
         Result := horizontal_bias.truncated_to_integer
      ensure
         …
      end

feature -- Miscellaneous
   postpone_turn_off (actual_horizontal_velocity: INTEGER) is
         -- Postpone turning off the alignment function for
         -- reinitialisation in case of a hold in the count-down.
         -- `actual_horizontal_velocity' needs to be within the
         -- required limit. Turn off this feature, if the limit
         -- cannot be guaranteed.
      require
         positive_velocity: 0 <= actual_horizontal_velocity
         maximal_horizontal_velocity:
          actual_horizontal_velocity < Maximal_horizontal_velocity
      do …
      ensure
         …
      end
…
```

Figure 1-1: Conversion and alignment function, in Eiffel, using Design by Contract.

The programmer, who implemented the additional feature, was not recommended to use Design by Contract and therefore he could only document it separately from the code, where especially rarely used routines and their requirements can easily be forgotten.

Ken Garlington criticizes in [5] that contract writing needs too much time for no immediate reward, since a program can be correct without use of contracts. At a first glance, this might seem to be true but on the one hand, if a programmer is used to writing contracts and he is conscious about the requirements his software has to meet, the time spent is negligible. Even if a programmer is not trained in writing contracts, by declaratively expressing the requirements of his software, he is more conscious about the requirements and the benefits of his code and therefore can avoid many errors. On the other hand, if someone has to reuse the code he profits, if every effect of the program is properly described, so that he immediately understands the purpose of components he wants to reuse.

Of course, writing contracts is something everyone has to get used to. Especially at the beginning, it takes some discipline to write contracts properly but it is worth for testing and for everyone, who has to look at the code again.

More comments on the Ariane 5 crash and Design by Contract can be read in [5].

1.3. Concept of Semantic Contracts

One of the fundamental pillars of structured programming is component-oriented design paradigm. The complexity is hereby decomposed into small program units, called the components. As the most essential point, components are solely accessible via interfaces, which explicitly define all dependencies between the component and the environment. Each interface constitutes a syntactic and a semantic part that are equally important for totally expressing all the inter-component interrelations and dependencies. The purpose of component-oriented philosophy is apparent:

- According to the principle of *divide and conquer*, the complexity of a software system is rigorously reduced by decomposition. The dependencies are entirely concentrated within the interfaces acting as contracts that have to be fulfilled by both the providing and the consuming components. The components can fully encapsulate implementation details as black boxes.
- Due to the explicitly formulated interfaces, a component behind becomes exchangeable as long as it completely agrees with the interface. This is the sound foundation for safe software maintenance and flexible extensions.
- As components form abstractions with clearly specified functionality, safe software reusability is enabled.

In the object-oriented programming paradigm, we can identify two kinds of components: (1) a routine, covering functions and procedures, as well as (2) objects, described by classes.

Classically, the syntactic interface of a routine is specified by the signature, comprising the parameter types (also called the formal arguments) and the return type (in case of functions), their passing semantics and ordering. Analogously, an object's syntactic interface consists of the features an object offers, i.e. the applicable routines, provided attributes and generic type parameters. The rules of a programming language ensure that no program participant can

circumvent these syntactic contracts. (An appropriate compiler is typically sufficient to enforce these rules).

The other but not less essential part of an interface is the semantic contract. This is to concretise both the meaning of the component and its syntactic elements as well as all the terms for correctly using the components. Whereas a reasonable level of syntactic interface description has probably been reached in programming, the possibilities for declaring the semantics of an interface are clearly far from what is required for accurate contractual specifications. This is especially true for most of the popular main-stream programming languages, such as C++, Java, and C#, to only name a few. An extremely modest kind of interface semantics may be the comments and good naming of components, features, parameters, and attributes etc, provided they are adhered to the interface. What is particularly regrettable is that none of these semantic conditions are enforced for the interaction and use of the components. There is for instance, no help by the machinery that these perfectly weak semantics of comments and naming are effectually agreeing with use-semantics on the client side.

Although one may suggest that the precise semantic description is truly more difficult than for syntax, there are numerous promising approaches for semantic contract specification, among some are realized in Eiffel. However, a methodology for thorough semantic specification, which is also verifiable by a computer in reasonable time, has not yet been found. Nevertheless, a certain level of semantic accuracy in the contractual agreement between components can be achieved. First, we quickly review some archetypical constructs for semantic specifications that are both equally well understandable by man and computer.

Invariants

An invariant is a condition that permanently holds during the entire computation of a defined program scope. Thereby, the computation process is abstracted as a series of state transitions, called atomic actions. Both before and after the execution of an atomic action, the invariant is fulfilled.

An action may appear in different granularity. In a standard imperative programming language, it can be perceived as a statement, or a basis operation is the notion of operational semantics. There may nonetheless be more complex atomic action, such as a general statement block, an entire routine.

A further concept traditionally mentioned together with invariants is the variant.

Variants

A variant is only defined with respect to a repetitive execution process and specifies a maximum number of remaining iterations until the termination of the repetition.

A classical execution repetition is a loop in imperative programming. At the beginning of each iteration step, the variant defines a positive numeric value. The value of a variant has to monotonously decrease with each iteration step and the repetition must eventually terminate at latest when the value has reached zero.

The combination of invariants and variants already enable a surprisingly powerful set for verifying interesting programs. The invariant hereby primarily focuses on the state consistency for the intended purpose of the program, whereas the variant is for guaranteeing the termination of a program. The example in Figure 1-2 illustrates a binary search, where invariants and variants prove the correctness of the algorithm. From the invariant and loop

termination condition, it is immediately implied that the result is either in `a.item(l)` or it is not contained in the array at all.

```
binary_search (a: ARRAY [INTEGER]; k: INTEGER): BOOOLEAN is
      -- Does ‘k' exist in sorted ‘a' with ‘binary_search'?
      -- Suppose the following sentinels
      -- a.item (1) = -Infinity
      -- a.item (a.count) = Infinity
   local
      l, r, m: INTEGER
   do
      from
           l := 1; r := a.count
      invariant
           (a.item (l) <= k)) and (k < a.item (r))
      variant
           r – l
      until r /= l + 1 loop
           m := (l + r) // 2
           if a.item (m) <= k then
                l:= m
           else
                r := m
           end
      end
      -- (r = l + 1) and invariant implies (a.item (l) <= k < a.item (l+1))
      Result := a.item (l) = k
   end
```

Figure 1-2: Binary Search with a Loop

Further archetypes for semantic contracts are conditions that do not necessarily hold all the time but are exactly fulfilled at certain defined program places. This may be used for setting program actions in a semantic context, such as with the so called *Hoare triples*.

Hoare Logic

A Hoare correctness formula {P} A {R} defines a program action A and two conditions P and R, such that R is fulfilled after each execution of A, provided that P holds at the beginning of the execution. P is called the *precondition* and R the *postcondition* of A.

What is in primary interest of this analysis, is that the pre- and postconditions are exactly denoting a semantic contract part of an interface for a component, here being the program action. The precondition secures the terms of contract for the sound use of the functionality, whereas the postconditions concretises the effective result that is offered by the component. Of course, the conditions must not reveal any implementation details of the component.

The Hoare logic inherently manifests the notion of mathematical induction proof, where the precondition represents an induction hypothesis, under which fulfilment the postcondition can

be implied. Especially for recursive problems but not limited to those, the induction is the very natural and effective mechanism for correctness proofs (c.f. Figure 1-3).

```
recursive_binary_search (a: ARRAY [INTEGER]; k: INTEGER;
    l: INTEGER; r: INTEGER): BOOLEAN is
        -- Does ‘k' exist in sorted ‘a' with ‘binary_search'?
    require
        is_between_limits:
            (l < r) and (a.item (l) <= k) and (k < a.item (r))
    local
        m: INTEGER
    do
        if l = r + 1 then
          -- a.item (l) <= k < a.item (r)
          Result := a.item (l) = k
        else
            m := (l + r) // 2
            if a.item (m) <= k then
                l := m
            else
                r := m
            end
            Result := recursive_binary_search (a, k, l, r)
        end
    end
```

Figure 1-3: Binary Search on an array programmed as a recursive function

Invariants, variants as well as the pre- and postconditions can be verified and ensured with several techniques, which we discuss in the subsequent chapter.

A core question in the specification of semantic condition is in what terms they are expressed. There are roughly two categories of approaches.

- *Special descriptive languages:* Semantic conditions are described in a special descriptive language. This enables the high-level declaration of conditions, with mere focus on the semantics in total liberation from the imperative computation of this condition. It can be namely observed that the necessary semantic contracts are actually required due to the lacking expressiveness of standard programming language for expressing such constructs by means of the classical semantic elements. A good candidate for such descriptive languages would be for example, the support of quantifiers. On the other hand, there are not many declarative programming languages, which are also reasonably efficient in verification. (An example of a rather efficient descriptive language is SQL, whereas Prolog entails exponential time complexity). A further benefit of special languages is that more sophisticated semantic conditions may be supported, such as limits on execution time, memory or power consumption etc.
- *Conventional imperative language:* It is straightforward to formulate semantic conditions in uniformity of the standard imperative programming language. However,

more complex conditions are not definable as pure Boolean formulas, which do not involve execution of other program parts. Therefore, functions have to be enforcedly used in the semantic condition, to enable sufficient flexibility. In this case, it must be however excluded that the execution of such functions causes side effects on the system. We will now discuss this issue in detail.

To simplify the use of contracts, Eiffel features preconditions, postconditions and class invariants as institutionalized concepts of the programming language. Beside the interface-related contracts, loop invariants and variants are provided. For pragmatic reasons and convenient usability, Eiffel chooses the approach of self-expressed semantic Boolean formulas. In the following, we review the concrete kinds of Eiffel semantic conditions.

Preconditions

With a precondition, a routine can specify a semantic condition, which must be fulfilled for all invocations of the corresponding routine. A precondition hence obliges the clients to keep its part of the postulated contract, such that the routine can fully rely on this fact. The expression belongs to the interface of a routine and has an equal importance as the routine's signature.

```
bank_withdraw (account: ACCOUNT; amount: INTEGER) is
        -- Withdraw 'amount' from 'account'.
    require
        enough_mony: account.balance >= amount
        …
    end
```

Figure 1-4: Precondition in Eiffel

Postconditions

The postcondition describes a functionality, which must eventually hold at the execution end of a routine. A postcondition can be viewed as a contract element of the routine, guaranteeing certain semantic properties for its clients. In this way, it acts as the opposite concept of a precondition, where the contract imposes an obligation for calling the routine itself. In combination with preconditions, inductive reasoning about function compositions is enabled in a program. On the other hand, a postcondition can also be hidden as an internal condition of the routine and the enclosing object class respectively. For this special case only, the postcondition can also refer to encapsulated features.

```
change_tires (car: CAR) is
        -- Change the tires of all wheels of ‘car’.
    do
        …
    ensure
        tires_present:
           car.left_front_wheel.has_tire and
           car.right_front_wheel.has_tire and
           car.left_back_wheel.has_tire and
           car.right_back_wheel.has_tire
    end
```

Figure 1-5: Postcondition in Eiffel

Class Invariants

Class invariants describe the state for an object, which holds before and after each routine call that comes from outside the object. (As a natural exception, the class invariant has not to be true before invocation of the object creation routine). Therefore, the conceptual atomic action is here regarded to be the complete execution of an external called routine (including all transitively called routines that belong to the same object instance).

```
class TREE
    …
feature
    force (key: INTEGER; data: ANY) is … end
        -- Add `data’ with `key’ to tree
    remove (key: INTEGER; data: ANY) is … end
        -- Remove `data’ with `key’ to tree
    item (key: INTEGER): ANY is … end
        -- Get element with `key’ from tree

feature {NONE}
    is_balanced: BOOLEAN is … end
        -- Is tree balanced?
invariant
    is_balanced:
        is_balanced
end
```

Figure 1-6: Class invariant in Eiffel

Loop Invariants and Variants

Loop invariants and variants are not a true contractual element but are rather for structured programming with execution loops. Hereby, the execution of a single loop iteration step is considered as the atomic action. The invariant is an assertion (Boolean expression), variants are integer expressions, which hold as long as they are greater than zero, describing the loop's execution condition (see Figure 1-2).

As semantic conditions are also expressed in the language of Eiffel, particularly in terms of Boolean functions in contracts, special attention has to be paid to various issues. We therefore postulate some fundamental rules for avoiding flaws in the programming model.

Rule 1 (No side-effects in semantic conditions)

Under no circumstances, a condition of a semantic contract can invoke a function that causes side-effects. Conceptually, a side effect is any operation that can be detected from the rest of the program, outside the function.

This rule is crucial for guaranteeing that the use of semantic contracts does not even introduce new error sources in programs. A semantic condition is of thoroughly descriptive nature and must not influence the principal program execution. As a consequence, it should not make any difference whether a semantic condition is checked or not. One should imagine the negative consequences of a program that behaves completely differently when its conditions are checked with a different mechanism. (This is for instance in the case of static verification instead of dynamic checking, or in the case when checks are turned off for efficiency reasons[1]).

A side effect is any write access on externally visible data, input/output operation as well as detectable effects, such as the non-termination of a program or errors that occur during the dynamic evaluation of semantic conditions. However, it is often not really recognized that in a concurrent system, even read-accesses to externally visible data may cause side-effects. An activity may wrongly read a value which is in fact in intermediate computation of a concurrently running activity. (This is the reason why transactional correctness relies on semantic serializability).

With regard to concurrency, a further ultimately important rule has to be introduced.

Rule 2 (No concurrency)

Both the precondition and the postcondition must not infer with concurrent activities. Neither may the execution process of the enclosed program routine have non-deterministic dependencies involved with concurrency.

As preconditions have the meaning of wait conditions in the context of concurrency, static analysis cannot be applied. Otherwise, each condition that is non-deterministically influenced by concurrency is conceptually unsound. (It may be successfully checked at runtime but may not be statically verified in any case).

A further rather technical problem comes up, if the function of a semantic condition also comprises semantic conditions and so on, probably resulting in infinite recursion. The best way in the interest of clarity of the model is to exclude this with the following rule.

[1] As Hoare already observed, turning the conditions off in productive releases is really irresponsible. "What would we think of a sailing enthusiast who wears his life-jacket when training on dry land but takes it off as soon as he goes to sea?"

Rule 3 (No nested semantic conditions)

Each function, which is used within a semantic condition, must not feature semantic conditions for itself.

In ISE Eiffel, the chosen solution for dynamic contract checking is that semantic conditions are ignored if they belong to transitively invoked functions of other semantic contracts.

Chapter 2 - Contract Assurance

2.1. Checking Methods

The only existing checking mechanism in Eiffel at the moment is dynamic checking. That is contracts are evaluated at runtime for test and debugging purposes. We now give an overview of this and other verification methods.

- ***Dynamic checking:*** A program is executed and the assertions contained in the contracts are evaluated at runtime. If a contract is violated the execution process stops and the found error can be corrected afterwards by the programmer. Consequently, at most one error can be detected for every program execution. When the program runs until the end without runtime failure, its correctness only holds for the special scenario that has just been executed. In a future program run, where the context may be even changed, it may however unexpectedly abort, because the program has not been verified for all possible execution paths. Thus, for dynamic checking, a program has to be extensively tested, in order to get reasonable reliability.

  It is specified in the Eiffel *Ace* files how contracts are monitored at runtime, besides all compilation options and the assembly information of a program. The following options are available within an *Ace* file:

  (1) Treat the contracts as comments, such that they have no influence in program execution.

  (2) Dynamically evaluate all preconditions, with the appropriate computation costs.

  (3) Check both all preconditions and all postconditions.

  (4) Dynamically ensure all class invariants in combination with the checks of (3).

  (5) Additionally to (4), loop invariants and variants are also dynamically ensured.

  (6) Turn on all dynamic checks, i.e. all previously mentioned conditions and also all remaining assertions.

  Furthermore, these options can be defined as (a) default for the whole program, (b) for specific clusters (related classes in a directory) or (c) for single classes. Obviously, it results in a flexible dynamic testing environment, where the testing degree can be customized, depending on the computing overheads one is willing to invest and the degree of confidence we want to get in the program.

- ***Static checking:*** At compile time (or in a following development step), static analysis is performed to verify if the fulfillment of contracts. Its great advantage is that it verifies that a contract permanently holds for any execution path. Therefore, it makes further (dynamic) testing redundant. As static checking requires detailed program analysis, it is mostly a trade-off between accuracy and analysis time. The more complex the analysis is, the more time it takes but the richer the result is expected to be. This is in terms of how many more contracts are detected to be fulfilled. So if reliability and correctness of a program is important, it is recommended not to spare considerable expense on static checking. Since the static analysis is generally

conservative, for some contracts, it is undecidable[2] for the analysis whether they are always kept. For these remaining cases, dynamic checking is still necessary.

- *Conservative Positive:* This approach uses a conservative approach to detect calls that always fulfill the preconditions of their callee routines. If this cannot be fully guaranteed, a call is regarded as possibly incorrect and has to be tested by another method.
- *Conservative Negative:* This method conservatively discovers calls that can never fulfill the precondition of their callees. With this approach, not every wrong call can be identified. The remaining calls are treated as possibly correct and have to be checked by another mechanism.

- ***Manual Checking:*** The user decides by himself whether a call really satisfies the preconditions of its callees (noticing that with sub-typing polymorphism there can be multiple call targets). However, a user may not have enough time to check the conditions carefully or he meets wrong assumptions about a contract. (This is like knowing every detail when signing a legal contract in daily life). It is nonetheless an option to make sure whether a contract is fulfilled, supposed that the requirements are correctly and entirely taken into consideration. (As for the crash of Ariane 5, the launcher specification has been ignored for the SRI).

The table below (Figure 2-1) compares dynamic to static checking. It shows that dynamic checking can complement the static analysis. The cases, which the static analysis cannot decide, need to be tested with a different method. Consequently, a hybrid approach should be chosen as realistic solution.

	Dynamic Checking	Static Checking	
		Conservative Positive	Conservative Negative
Always fulfilled	-	X	-
Fulfilled specific	X	-	-
Undecidable	-	X	X
Not fulfilled specific	X	-	-
Never fulfilled	-	-	X

Figure 2-1: Comparison of different checking methods

Explanation: The method has determined that …

Always fulfilled: - the contract always holds.
Fulfilled specific: - the contract holds in a specific execution scenario.
Undecidable: - it can not decide, whether the contract holds or not.
Not fulfilled specific: - the contract is not fulfilled in a specific execution scenario – an error was found.
Never fulfilled: - the contract will never hold – an error was found.

X represents the scenario, where a method has come to pass.

[2] Contrary to the mathematical sense, undecidability means here that it depends on the strength of the static analysis. Therefore it is possible, that a call is determined as fulfilled by one analyzer and is undecidable for another.

This thesis has pursued the *conservative positive* static analysis. Since the contracts are designed to satisfy them, they are expected to hold in most cases. We hence expect to find many fulfilled contracts in the empirical evaluation. To implement the conservative negative approach, the expense would be at least as high as for the conservative positive analysis. This is because in our context, we have not necessarily an equivalence relation. More concretely, a conservative positive analysis finds out whether a call A implies the *fulfillment* of a contract C, whereas a conservative negative method detects whether A implies that C is surely *violated*.

The conservative negative approach is furthermore not that promising, as there are only few such thoroughly false calls expected, compared to a whole program. The effort of a static analysis is insofar not justified, as the analysis is still conservative and would not find every error. In contrast, a *conservative positive* analysis detects at least all not fulfilled conditions and is therefore chosen for implementation in this thesis, combined with dynamic testing of the undecidable calls, yielding hence a kind of a hybrid approach.

Throughout this thesis, we focus on the verification of **preconditions**. Since, it is the most important part of the contract for a client who wants to reuse a routine.

2.2. Development Process with Precondition Enforcement Analysis

We expect that static analysis once will be a regular part of the development process for quality software. Figure 2-2 illustrates the adapted process, where static verification is conceived as an independent element of the test phase, like the Precondition Enforcement Analysis is separate from other phases. For every phase, the development process is considered to be iterative. Errors, which can occur in every step, have to be iteratively corrected by returning to a previous phase. The arrows visualize the main stream of the whole development process.

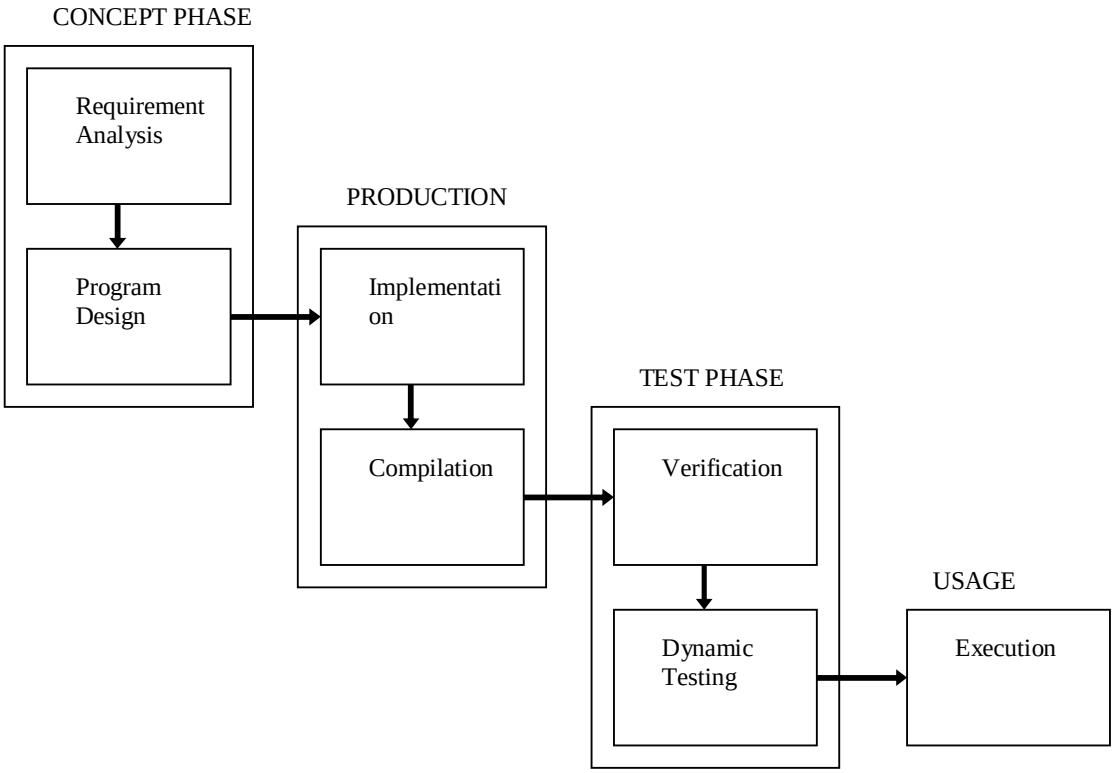


Figure 2-2: Development process of quality software

Chapter 3 - Static Contract Verification

This thesis pursues the positive conservative approach, with the ideal to find as many fulfilled preconditions as possible. Conservatively, every call is judged to be fulfilled, considering information about the caller and the callee routines. We describe in the next sections, how it is statically determined in our analyzer whether a precondition is fulfilled. Figure 3-1 shows an example, where the analysis is successful in detecting that the preconditions are always satisfied by the client.

```
    b_attribute: BOOLEAN

    caller_routine is
        local
            x, k, l: BOOLEAN
        do
            if x then
                foo1 (x)
            elseif not(k and l) then
                foo2 (l, k)
            end
        end

    foo1 (a: BOOLEAN) is
        require
            a or b_attribute
            …
        end

    foo2 (u, v: BOOLEAN) is
        require
            not (u) or not (v)
            …
        end
…
```

Figure 3-1: Example to illustrate, what expression the analysis can recognize to be fulfilled. Both preconditions of foo1 and foo2 are fulfilled.

3.1. Information Used for Verification

To perform the analysis within reasonable time, we limit the context of a considered call to the following information:

(1) The caller routine (and the enclosing object), wherein the call is contained.

(2) The callees' preconditions.

We are conscious, that the result is less rich than with an interprocedural or data flow analysis. In return, the complexity of the analysis is considerably lower.

3.2. General Approach

A client routine, which wants to make proper use of a called routine, has to make sure that he fulfills the corresponding preconditions. Therefore, it has to somehow check this condition before the target routine is eventually called, naturally with a control construct, if the condition holds not for some other reason. As there are different possibilities to check such a condition, the first task was to look for possible code patterns which can ensure a precondition.

(1) Conditional
Because the conditional is an often used conditional instruction, it has been studied very carefully for the possibility of assuring preconditions.

```
…
if condition then
      call_a
      …
else
      call_b
      …
end
…
```

Figure 3-2: Basic conditional construct, where `condition` is a boolean expression.

In the basic conditional construct as illustrated in Figure 3-2, there are already two possibilities, where a client can possibly assure a call's precondition.

a) `condition` implies `call_a`'s precondition

b) not `condition` implies `call_b`'s precondition

```
…
if c1 then
      call_a
      …
elseif c2 then
      call_b
      …
elseif c3 then
      call_c
      …
…
else
      call_d
      …
end
…
```

Figure 3-3: General conditional construct, where `c1, c2 and c3` are boolean expressions.

Figure 3-3 shows a general case of a conditional construct there are even more possibilities, that a call's precondition is fulfilled.

a) `c1` implies `call_a`'s precondition (like before)

b) (not `c1` and `c2`) implies `call_b`'s precondition

c) (not `c1` and not `c2` or `c3`) implies `call_c`'s precondition

d) (not `c1` and not `c2` or not `c3`) implies `call_d`'s precondition

So for a call_i, after the i-th `if` or `elseif` clause the implication is:

(not c_j AND) and c_i implies call_i, for j = 1..i -1, for all i in [1…n]

For the a call_i, after the `else` clause, it holds:

(not c_j AND) implies call_i, for j = 1..i, for all i in [1..n], being the `else` clause the i +1 th clause.

The conditional construct gives many options for a precondition to be fulfilled, increasing with the complexity of the construct.

To make sure that a call's precondition is not falsified again, before the call's execution, we only make use of conditional construct, if a call is the first instruction in the then part, or in the else part. Otherwise we would need a more extensive analysis, which includes control flow analysis, and analysis of reachability of references which is too expensive and beyond the scope of this thesis.

(2) Loop
The loop instruction offers not as many possibilities, but still a precondition may be assured this way.

```
…
from
      …
until
      exit_condition
loop
      call_a
      …
end
call_b
…
```

Figure 3-4: The loop instruction, where exit_condition is a Boolean expression.

As Figure 3-4 illustrates, there are two options, where a precondition can be fulfilled.

a) not `exit_condition` implies `call_a`'s precondition
b) `exit_condition` implies `call_b`'s precondition

The multi-branch instruction (**`inspect`**) has not been included, because it contains only a special condition (`e = vi`), where e is an expression of type *INTEGER* or *CHARACTER*, and v_i a constant of the corresponding type, and therefore appears rather seldom in code.

Neither the **`check`** instruction has been considered, since it is an assertion and can be turned on or off with assertion monitoring. So they do not appear always in the code.

Implication
With the non-strict Boolean operator `and then` and `implies` a call's precondition can hold as well (see Figure 3-5).

a) `c1` implies `call_a`'s precondition
b) `c2` implies `call_b`'s precondition

```
…
      c1 implies call_a
      …
      c2 and then call_b
…
```

Figure 3-5: Keywords for implication

3.2.1. Comparison of Precondition

After we have identified how a client can possibly check that he complies with the required precondition, we have to verify, whether the condition, really imposes the callee's precondition. Therefore a recursive approach has been chosen, which makes use of common logic rules and compares the precondition expression with the conditional expression in the caller (the calling routine).

The simplest case is illustrated in Figure 3-6. Here we only have to compare one argument with the condition expression. Since, we have this to test for more complex expressions as well, it is the most often used rule. All applied implication rules or equivalence are listened in Figure 3-7.

```
caller_routine is
   local
        x: BOOLEAN
   do
        if x then foo (x) end
   end

foo (a: BOOLEAN) is
   require
        a
        …
   end
```

Figure 3-6: The simplest case, of a condition and precondition expression.

Implication Rules

Client condition	implies	**Precondition**	Rule name
Special expressions, constants and identifier:			
Result		**Result**	
Current		**Current**	
constant		constant	
identifier		identifier	
Void		Void	
a		True	
Implication expressions:			
a and b		a b	
a b		a or b	
Equivalence expressions:			
a = b		f = g	
a /= b		f /= g	
a > b		a > b	
a < b		a < b	
a >= b		a >= b	
a and b		a and b	(for “and then” too)
a or b		a or b	(for “or else” too)
a op b		a op b	for op = {-, +, ·, /, //, \\, ^, xor, implies}
a = b		b = a	Commutativity for: =, /=, and, and then, or, or else, +, -, xor
a > b		b < a	Binary inversion for: >=, <=, <, >
+ a		+ a	Equality for unary operations: +, -, not
not (not a)		a	Unary inversion for: not, -
a		not (not a)	“

not (a = b)		a /= b	
a /= b		not (a = b)	
not (a /= b)		a = b	
a = b		not (a /= b)	
not (a and b)		not a or not b	De Morgan
not a or not b		not (a and b)	“
not (a or b)		not a and not b	“
a implies b		(not a) or b	“
(not a) or b		a implies b	“
(a)		a	Remove parantheses
a		(a)	“

Figure 3-7: Implication rules

The mathematic expressions have not been evaluated, because to reason about we would need information of the values, which needs a more complex analysis. But many logic rules have been implemented, with which we can reason about such preconditions, which are inherently Boolean expressions. The non-strict operators “and then” and “or else**”** operators are treated equally, because they hold in the same case as their strict operators hold if the other holds, it just saves an instruction in the non-strict case.

Of course, this implication test can be extended, but it needs further analysis about the particular variable values which is easy in case of constant value, or not to determine in case of attributes.

Since the rules which are recursively checked, get always smaller, the termination is determined. (Divide and conquer)

Arguments
If the precondition contains arguments, its expression has to be adapted, so that it can be compared to the caller's conditional expression (see Figure 3-8). The modified precondition in the example would be: t.*item* (i) /= Void; t.*count* > 7; *boolean_attribute*.

```
call_a (a: A, b: INTEGER)
      require
            a /= Void; b > 7; boolean_attribute
      …
      end
…
call_a (t.item (i), t.count)
…
```

Figure 3-8: Example for arguments

So a call's argument can turn out to be a call itself. This scenario is not analyzed any further, because the call could falsify the conditional expression, which has been checked before. It would need an inter-procedural analysis to reason about such a case.

The arguments are also resolved in the recursively. Arguments can of course, be only from the callee's class, or of its parent's class otherwise, the callee would not see it (visibility), the same stands for Boolean functions. Every element of the precondition has to be visible for the caller and the callee.

Precondition Splitting

To simplify the precondition, we choose to split them, if possible. Also there are several possibilities to express the same precondition in Eiffel like:

- A and B (A and B have to hold to fulfill the precondition.)
- A; B (The semicolon stands for an "and" operator)
- A
B (Because they are written on different lines, A and B are connected by an "and" operator.)

At the end of the analysis, we check for every call, whether all precondition parts are fulfilled, which implies that the whole precondition is satisfied. Of course, as Eiffel contains multiple inheritance, we have to consider the inherited preconditions as well. They are logically appended to other preconditions with "or else". So either one precondition or the inherited has to hold (analogous for many preconditions).

3.2.2. Is the Precondition satisfied?

After we have found a way to know how a client can check the precondition, and how the precondition can be compared with the client's condition expression, we have to make sure that a precondition is not falsified again, before the call. We have chosen the simple approach that the call has to be just after the client's check, like Figure 3-9 illustrates with `instruction_1`.

```
…
client_check
instruction_1
instruction_2
…
```

Figure 3-9: Instructions after a client's check.

A more complex analysis could check, for a call after instruction_1, whether the expressions used for the precondition are not changed, but this would need an inter procedural dataflow analysis.

3.3. Specific Approach

Although the analysis described before is quite successful, the reward was not as great as expected, since it turned out that it does not recognize the vast majority of the preconditions (see experimental results), which is that an object must not be void. This is important especially in object oriented programming, since it makes heavy use of references, and we often have to make sure that an object is instantiated. Often a supplier offers a routine, where the argument (or an attribute) must not be void, and he leaves it to the client, that he assures this condition. In Eiffel this is declared in the precondition as well.

To illustrate the scenarios, where such precondition can be determined to be fulfilled, we give a general sample to which we are going to confer in the special cases (Figure 1-1).

```
bar is
      require …
      do
            …
            b := f (x)
            foo (b)
            …
      ensure …
      end
```

```
foo (a: A) is
require
      a /= Void
      t /= Void
do
…
end
```

Figure 3-10: Sample, to show examples where a /= Void holds.

(1) In Eiffel, there are expanded types, which means, that instances of expanded types are automatically initialized and cannot be void. Base types e.g. *CHARACTER*, *BOOLEAN*, *INTEGER*, *REAL* and *DOUBLE* are also of expanded type. Now, if the formal argument `a` is of expanded type, the precondition `a /= Void` holds.

(2) Same thing for the attribute `t`, if it is of expanded type.

(3) If `b` is a manifest constant it naturally cannot be void.

(4) If `b` is a call to a manifest constant.

(5) Same thing when `b` is Current. The self reference cannot be void.

(6) The instruction just before the call of `foo`, `b` is the result of another feature call, e.g. `b := f (x)` and the feature f has a post condition `Result /= Void` and `b` is a local variable or the result of a function. (If `b` is an attribute, we cannot know anything because in a call of the form `g (foo, b)`, `foo` may be a routine that resets `b` to Void.)

(7) `b` has been created just before the call of `foo`, where `b` is either a local varable or the result of `bar` (see Figure 3-11).

```
bar
      local
            b: B
      do
            create b.make
            foo (b)
            Result.make
            foo (Result)
            …
      end
```

Figure 3-11: Argument has been created before the call of foo.

(8) The argument `b` of `foo` is another call with, either a creation call, a call, where the Return value is of expanded type or the call has a postcondition like Return /= Void (compare Figure 3-12).

```
…
      foo (create {SOME_TYPE}.make)
      foo (call_expanded)
      foo (call_pc)
…
call_expanded: BOOLEAN is
      do
            Result := True
      end

call_pc is
      do
            …
      ensure
            Result /= Void
      end
```

Figure 3-12: Illustrates that a call as an argument can assure, precondition /= Void.

(9) If the calling routine `a_routine` (see Figure 3-13) has a formal argument `b`, and a precondition `b /= Void`. This holds because, in Eiffel no assignment to s formal argument is allowed.

```
a_routine (b: A) is
      require
            b /= Void
      do
            …
            called_routine (b)
            …
      end

called_routine (a: A) is
      require
            a /= Void
      …
      end
```

Figure 3-13: Example for formal argument with precondition

In the case the feature `foo` may be executed successfully, the client has to be sure or to make sure that b is non void. In some cases we can check this condition statically.

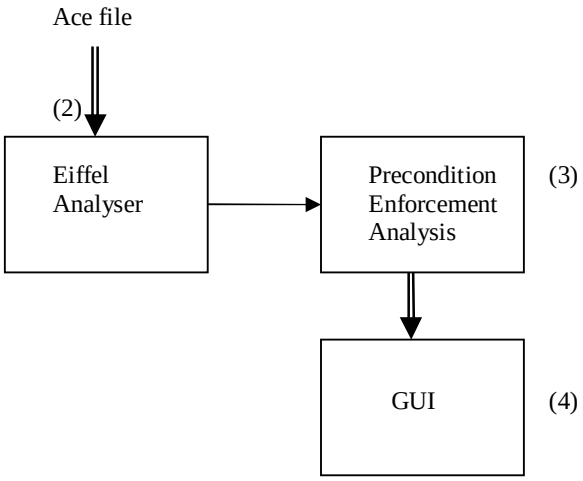
As a routine, which has no precondition, meaning no special requirements for a client, it similar to a precondition, which is always true. This is the case, also for native library calls (to c-libraries), which are not equipped with libraries. Consequently, they are verified to be fulfilled, which does not say anything about the analysis, and even about the correctness of the library call.

Of course, the analysis does not claim to be complete, and there are other aspects which can be considered, e. g. inter-procedurality. But the time needed for the analyis would increase too. And in spite of the limitation we can detect many preconditions, which hold.

Chapter 4 - Design and Implementation

This chapter describes the design and implementation of the Precondition Enforcement Analysis. The whole process roughly consists of the following steps: From a chosen program the source is parsed and needed information for the subsequent analysis is gathered. After the analysis the result is presented in the GUI.

The order of events is described as follows in Figure 4-1:



(1) From a chosen Ace file[1] the program is parsed, and the Universe is generated. Now, we know every class the program uses. The Eiffel Analyzer parses the source code and collects the information for every routine call.

(2) Every call is verified by the Precondition Enforcement Analysis.

(3) The result is presented in the GUI.

Figure 4-1: Order of events in the Precondition Enforcement Analysis

4.1. Eiffel Analyzer

A program, which is expected to be syntactically correct, is parsed in the Eiffel Analyzer. It bases on the tool GOBO Eiffel Lint [3] called *gelint*, which analyses Eiffel source code and reports validity errors and useful warnings. From the program's Ace file, the universe is computed, which are the class files the program needs. *ET_FEATURE_CHECKER* is part of *gelint* (see Figure 4-2) like other classes starting with "ET_".

The following graph gives an overview of the classes used for the whole Precondition Enforcement Tool.

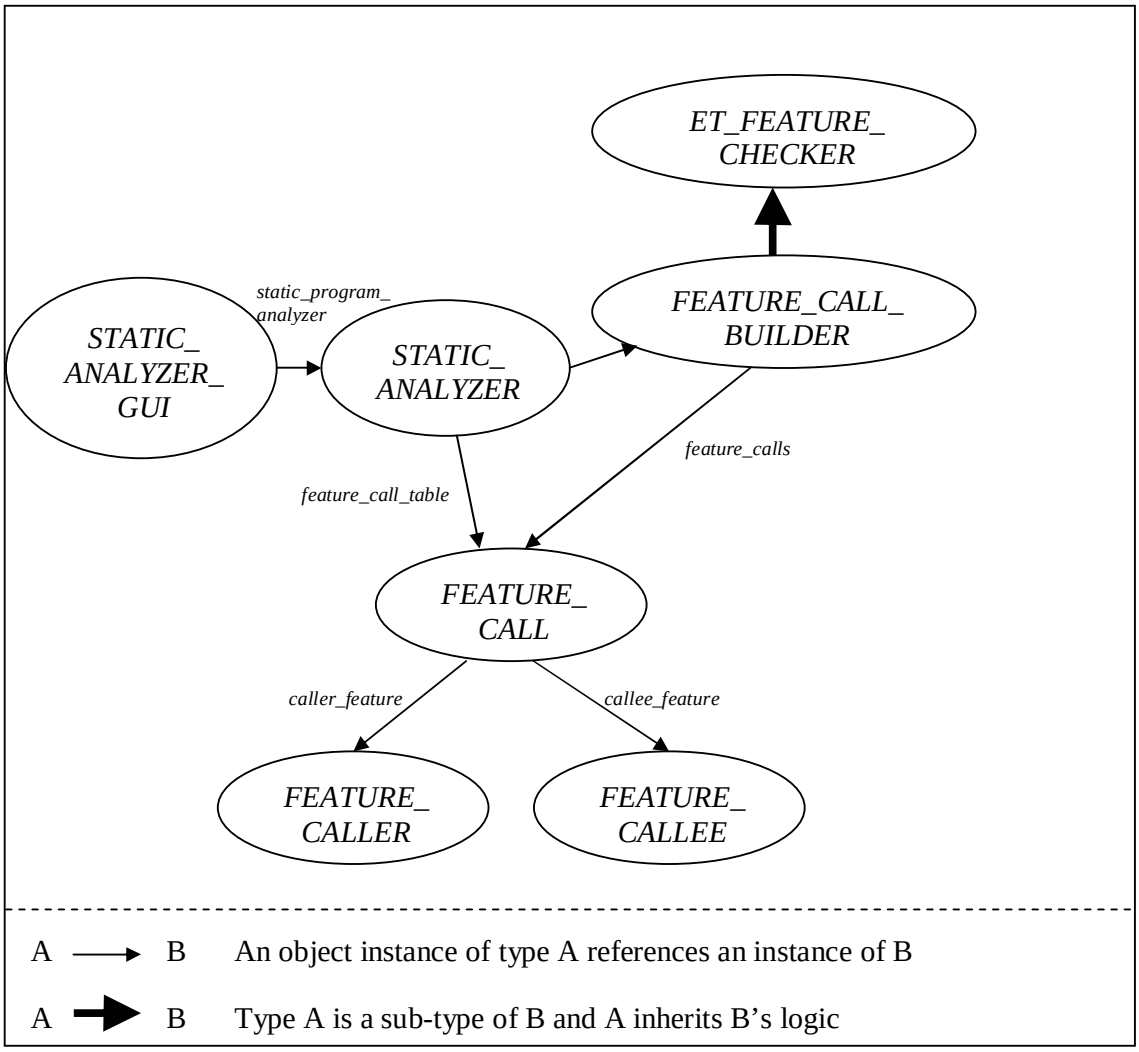


Figure 4-2: Diagram of the main classes of the Precondition Enforcement Analyis.

Collection of Calls

After the files have been parsed and the abstract syntax trees (ASTs) have been built, the features are checked, whether they are syntactically correct by the *ET_FEATURE_CHECKER*, which uses the visitor pattern [7] to traverse the ASTs. To get the needed information about every feature call, we have created the ***FEATURE_CALL_BUILDER***, which inherits from *ET_FEATURE_CHECKER*. For every processed call[3], a *report_call* routine is executed. The *report_call* routine is defined in *ET_FEATURE_CHECKER*. The actual implementation of *report_call* is in class *FEATURE_CALL_BUILDER*, where the routine had to be redefined.

Figure 4-3 illustrates the visitor pattern, which is used to traverse the ASTs. As an example, we take a creation expression, which is processed in *ET_FEATURE_CHECKER*.

[3] As attribute are at the moment not equipped with contracts, they are not considered as calls and therefore do appear not in the table *feature_calls*. If this will be the case, they can be easily included by commenting out the corresponding code in the class *FEATURE_CALL_BUILDER* in the routine *get_call_infos*.

1. Somewhere in class *ET_FEATURE_CHECKER*, actually in *check_expression_validity* an expression has to be processed. As we do not know its exact type (and we do not need it), we process it and pass the **Current** instance.
2. As the expression is of type *ET_CREATE_EXPRESSION*, which inherits from *ET_AST_NODE* and implements the feature *process*, like every node in the AST, which inherits from *ET_AST_NODE*, the corresponding feature in *ET_FEATURE_CHECKER* is executed and the current instance of the creation expression is passed with.
3. Again in *ET_FEATURE_CHECKER*, the feature *process_create_expression* is executed, which processes the creation expression, by calling the feature *check_create_expression_validity*, wherein the routine *report_creation_expression* is called, where finally in *get_call_infos* the call is stored in the table *feature_calls*.

```
-- 1. ET_FEATURE_CHECKER
   an_expression.process (Current)

-- 2. ET_CREATE_EXPRESSION
   process (a_processor: ET_AST_PROCESSOR) is
         -- Process current node.
      do
         a_processor.process_create_expression (Current)
      end

-- 3. ET_FEATURE_CHECKER
   process_create_expression (expression: ET_CREATE_EXPRESSION) is
         -- Process `expression'.
      do …
            check_create_expression_validity
                  (expression, current_context)
         …
      end

-- 4. FEATURE_CALL_BUILDER
   report_creation_expression (creation_type: ET_NAMED_TYPE;
      a_procedure: ET_FEATURE; actuals: ET_ACTUAL_ARGUMENTS;
      pos: ET_POSITION) is
         -- Report that a creation expression has been processed.
      do
         get_call_infos (a_procedure, creation_type, actuals, pos)
      end
```

Figure 4-3: Visitor pattern for traversing the ASTs.

Of course the most important information about a call is to know his caller and the callee feature. Then, the specific properties for a call have to be identified, which turned out to be the actual arguments, which are passed in a call, if there are some and the call's position in the code and to know, whether a call's precondition is fulfilled or not, which will be determined later in the Precondition Enforcement Analysis.

For this purpose we created the class ***FEATURE_CALL***, with the attributes:

- *caller_feature*: Is a reference of the *FEATURE_CALLER*, which stores mainly information about instructions which appear in the caller.
- *callee_feature*: It stores primarily information about the precondition expression.
- *arguments*: The actual argument, which are passed from the caller to the callee, if there are any.
- *call_position*: The call's position in the source code
- *preconditions_information*: As it is of type *ARRAY* [*BOOLEAN*], it can be specified, whether a precondition has determined to be fulfilled.
- *precursors_precondition_information*: Is of type *ARRAY2* [*BOOLEAN*]. As one precursor's precondition may contain any number of expressions, the maximal number of expressions was chosen to be the second dimension of the two dimensional array. To know the exact number of expression, we have to look it up in the *callee_feature* table.
- Like *preconditions_information* for every precursor, which has a precondition (the precondition is inherited by or else logic)
- *are_preconditions_fulfilled*: Is every precondition fulfilled?
- *considered_fulfilled*: Does the user think the preconditions are fulfilled?

As there are many calls, which have the same caller or the same callee, there are the tables *feature_callers* and *feature_callees*, which store them only once. The elements of these tables are of type *FEATURE_CALLER* and *FEATURE_CALLEE*.

FEATURE_CALLER: This class contains specific information about the caller feature, which is: its feature, *caller_feature*, with various information and its type, *caller_type*, and moreover different instructions, which can be used to assure a precondition.

FEATURE_CALLEE: Contains specific information about the callee, like its feature, *callee_feature* and its type *callee_type* and especially the preconditions and the precursors' preconditions in the attributes *preconditions* and *precursors_preconditions*, which are of type *DS_ARRAYED_LIST* [*PRECONDITION_INFORMATION*] and *DS_ARRAYED_LIST* [*DS_ARRAYED_LIST* [*PRECONDITION_INFORMATION*]] respectively. They keep the expression because the precondition may be changed by the *split_expression* routine (see below).

The following features in *FEATURE_CALL_BUILDER* add additional information to the tables *feature_callers* and *feature_callees*.

About the caller:

report_assignment

report_instruction

report_if_instruction

report_loop_instruction

report_infix_expression

report_creation_instruction: Reports the creation instruction as a call, and as a specific instruction for the caller.

About the callee:

report_precondition_expression: It splits the precondition, if necessary using the feature *split_expression* and adds precursors' preconditions.

split_expression: As preconditions form a part of the whole contract, a preconditions often consists of many conditions, which are connected by the "and" operator in case they all have to hold. We think it is interesting to know, if only a part of the precondition can be determined to hold. Since often, they are independent conditions.

Therefore a precondition is split into two (or more) expressions, so they can be verified separately. To keep these expressions, the class *PRECONDITION_INFORMATION* has been created, where this information is stored.

4.2. Precondition Enforcement Analysis

The Precondition Enforcement Analysis takes place in the class *STATIC_ANALYZER*. But at first the parsing process for the Ace file is started in this class, as well as the parsing of the source code, where the needed information is gathered.

The collected information is found in the attribute

- ***feature_call_table***

  Which is of type:

  DS_HASH_TABLE [*DS_ARRAYED_LIST* [*FEATURE_CALL*], *ET_FEATURE*]

  This means that if we search a certain call, we have to know its caller. Because with this information we receive a list of type *DS_ARRAYED_LIST* [*FEATURE_CALL*], which contains the call and other calls, which have the same caller.

- ***feature_caller_table***

  Which is of type:

  DS_HASH_TABLE [*FEATURE_CALLER*, *ET_FEATURE*]

  Specific information about a caller is found by searching with the caller feature.

- ***feature_callee_table***

  Which is of type:

  DS_HASH_TABLE [*FEATURE_CALLEE*, *ET_FEATURE*]

These tables are actually the same as the tables *feature_calls*, *feature_callers* and *feature_callees* in *FEATURE_CALL_BULDER*. Since the *STATIC_ANALYZER* passes references of these tables to the *FEATURE_CALL_BUILDER*.

Afterwards the analyzing process starts with the routine *check_contracts*, which gives possibilities for further extensions especially concerning other contract parts like postconditions or class invariants. At the moment *check_contracts* calls *check_preconditions*, from where the several rules are checked, which have described in chapter 3 in detail.

As every precondition is analyzed in every call, there are some attributes to implement that:

- *current_call*: Represents the call, which is currently analyzed.
- *current_precondition_information*: Is of type *PRECONDITION_INFORMATION* and keeps information which precondition is exactly analyzed besides the precondition expression.
- *call_before*: Is the call before the *current_call*, it is used in the analysis.

For the routine *implies_precondition*, where the condition from the client is checked, whether it implies the callee's precondition for every single call, it is worth looking at it more closely (see Figure 4-4). When the precondition p is compared to the client's conditional expression e, they can be equal or different, because arguments can be included in the p. As these expressions are recursively compared, the arguments are only checked at the leaves of the expressions, which are e.g. of type *ET_IDENTIFIER* of for constants. So, at the first call of *implies_precondition* the argument `are_parameters_resolved` is false, and it stays false until the expression is at the leave node, where the arguments are resolved. If the actual argument happens to be a call, it is no handled and the result of *implies_precondition* is false.

```
-- Signature of
implies_precondition
   (are_parameters_resolved: BOOLEAN; p: ET_EXPRESSION;
   e: ET_EXPRESSION): BOOLEAN is
      -- Does `e' imply (precondition) `p'?
      -- `p' is the precondition expression, `e' appears in the
      -- caller feature. `are_parameters_resolved' states, whether
      -- the formal arguments used in `p' have already been
      -- exchanged to the actual arguments.
   require
      p_not_void: p /= Void
      e_not_void: e /= Void
   …
   do …
      if are_parameters_resolved then …
            -- Compare `p' to `e' …
      else
         if arguments /= Void then
            from i := 1 until i > arguments.count or is_parameter loop
               formal ?= formal_argument (i).name
               if formal /= Void and then
                  formal.name = ident_p.name then
                     is_parameter := True
                     Result :=
                     implies_precondition (True, actual_args (i), e)
               end
               i := i + 1
            end
         end
         if not is_parameter then
            Result := implies_precondition (True, p, e)
               -- Variable not parameter
         end
      end …
   end
```

Figure 4-4: How the formal arguments are exchanged by the actual arguments. (Pseudocode)

In the routine *check_preconditions*, additional routines can be added, which perform other verification methods.

4.3. Graphical User Interface (GUI)

As within the project plan and the development of the project, the emphasis was rather set on the program analyzer than on the GUI. So its implementation is merely to describe the results of the analyzer and no incremental use (especially for compilation) has been implemented. As the GUI makes only use of the EiffelVision2 library, it should work for other platforms too, but it has only been tested on Windows XP.

After the user has launched the Precondition Enforcement Analysis, he has to choose a program by selecting File -> Open, which he wants to analyze.

This action is immediately followed by the static analyzing process, where it is statically determined whether a call's precondition is fulfilled, for the program's root classes.

After the analyzing process, the result is shown in the GUI (see Figure 4-5). There are two boxes, one for the tree with the fulfilled calls and another for the calls, which could not determined to be fulfilled. If a call's precondition is partially fulfilled it is considered not fulfilled by the static analysis. The calls are sorted according to the calling routine's class, the calling routine, the callee's class and the appearance order in the class. Each leaf of a tree represents a call, which is described by the callee's class, the callee's routine, the position in the code and of course the caller's class and routine in the parental graphical nodes.

Now, the user can analyze the not fulfilled calls. If he decides, that a call is still fulfilled, he can select it and move it to the other tree by pressing the <- button. Afterwards, the call is shown on the other tree marked with a "!" ahead, to show, that this is a user decision and the user has changed the call's status determined by the static analysis. If the user unchanged the call's status, there is no "!" appended to the call. Actually, it should be only possible for a user to move a call to the left tree, if he has decided that the call is fulfilled. But because the user might change his mind, it is possible to move calls in both directions.

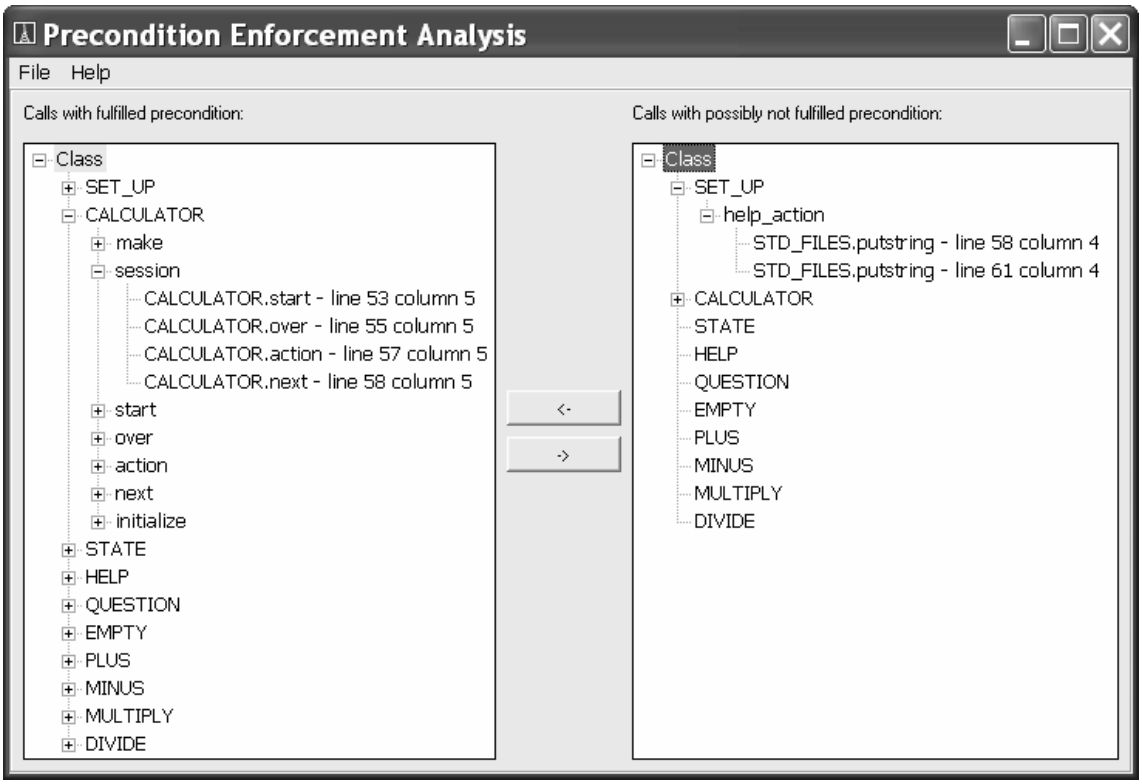


Figure 4-5: Main window of the GUI

The GUI part of this thesis consists mainly of these classes.

- *SPA_GUI*: From this class the whole application is launched, especially the GUI.
- *HOW_TO_DIALOG*: This class is for a pop-up window, describing succinctly the use of the Precondition Enforcement Analysis. It appears when pressing the Help Menu.
- *MAIN_WINDOW*: This class contains the main part of the GUI application, containing the composition of the main window, the menu bar etc. If a Ace file has been chosen, it creates an instance of *STATIC_ANALYZER*, which processed the precondition enforcement analysis. Furthermore it shows, the fulfilled calls and the others in a tree and is responsible for the movement of the single calls.

In Figure 4-6 the code extract from class *MAIN_WINDOW* shows, how a call can be moved to the fulfilled calls, which could not be determined to be fulfilled by the static analysis.

```
selected_call: FEATURE_CALL
make_fulfilled: BOOLEAN

-- In initialize the agent, move_call_left is attached to the
move_call_left_button.
      create move_call_left_button.make_with_text_and_action
            ("<-", agent move_call_left)

-- Agent selected_item has been attached to every call.
selected_item (a_call: FEATURE_CALL) is
            -- Save selected item in `selected_call'
      do
            selected_call := a_call
      end

fulfilled_calls_window (left: BOOLEAN) is
            -- To know, whether left or right tree is selected
      do
            make_fulfilled := not left
      end

move_call_left is
            -- Move `selected_call' from one list to the other and
            -- change the call's status
      do
            if selected_call /= Void and then make_fulfilled then
                        -- change flag in call
                  if not_fulfilled_calls_tree.selected_item /= Void then
                        -- User thinks call should be considered fulfilled
                        selected_call.set_considered_fulfilled
                        fill_call_lists
                        selected_call := Void
                  end
            end
      end
```

Figure 4-6: How a call is moved from one tree to the other.

The trickiest part turned out to be, that a previously selected call cannot be moved to the other tree, if a call from the other tree has been selected lately, and the wrong move button is

pressed. This has been solved by the Boolean attribute *make_fulfilled*, which is true, if the lately selected call is in the tree of the non-fulfilled calls. Furthermore every tree item, which is a leaf (every call) is equipped with an agent *selected_item*, which saves the lately selected call. The butten to move a call to the left is attached with the agent *move_call_left*, which changes the state of the call with *selected_call.set_considered_fulfilled* and then repaints the graphical trees.

4.4. Extensions

It has to be mentioned that the application can only be executed for one program. For another program another application has to be started. This is because the compiler cannot be reinitialized as is, but it keeps singleton to a hidden internal table, which can not be reinitialized a second time.

A further extension is for not fulfilled calls to show, if at least some part of the precondition could have been determined to be fulfilled. As this information is already contained in the table of calls, and the table of callees for the precondition expression, it could be done straightforwardly by extending the GUI.

Furthermore only the calls in the program's root cluster are analyzed, because the other calls (often within a library) are expected to be already assured. This can be easily changed, if one wants to verify every call of a program, by commenting out the corresponding code in *get_call_infos*, in *check_contracts* in the class *STATIC_ANALYZER*, and in *fill_calls* in *STATIC_ANALYZER_GUI*. In any case the whole program has to be processed, since the callees can be anywhere in the universe. A possible extension is, that a user can select the cluster or classes, he wants to analyze in the GUI.

It would enhance runtime checking, if we had the possibility to turn off the statically guaranteed preconditions for dynamic testing.

Chapter 5 - Project Plan

Eventually, we compare the goals determined in the project plan with the result of this project.

5.1. Achievements

5.1.1. Static Analysis

As we planned to start with a static analysis, which compares a precondition with a client's condition only by string comparison, the resulting analysis is quite striking, since it includes logic rules and special verification for a precondition, which assures that a reference is not void.

5.1.2. Graphical User Interface (GUI)

In the project plan, we intended to create one list of suspicious calls, for the calls, which could not be determined to hold in the static analysis and the list of false positive, for calls, which the user considers to be fulfilled, although the static analysis could not determine it. We think now, it is better to provide a list (or a tree) for the fulfilled calls and for the possibly fulfilled calls, because it gives the user more flexibility to analyse the result of the analysis. However, the GUI rather gives a view of the analysis and no other functionality, it can be changed easily.

5.2. Limitations

Although, we planned to persist the calls, which could not be verified to hold, but are considered to be fulfilled by a user, in order to enhance the incremental development and software quality analysis, this version of the Static Precondition Analyzer does not support that functionality.

The intended result to insert check instructions or bug fixes turned out to be useless, because check instructions are treated even weaker than preconditions. It now seems to us like double checking something, which is not recommended. On the other hand the expense of bug fixes seems not to be justified, as if an error can be found, it can be directly corrected in the source code.

Chapter 6 - Experimental Results

From the calculator example (included in Eiffel Studio), we obtained the following statistics:

Statically fulfilled calls	In the whole program	In the root cluster
if conditional	9	0
implication	18	0
loop	0	0
not void	344	10
“a /= void” expressions in preconditions	1455	14
# calls with preconditions	2318	14
# calls	11654	148
# classes	166	10

It shows clearly that the most successful pattern for finding a fulfilled precondition is to check whether the precondition is like a /= Void.

On the one hand it shows a big problem of object oriented languages, that references may or may not be void, because it appears so often. On the other hand it is quite difficult or some times impossible to verify a complicate precondition expression including attributes and calls under the scope of this thesis, which did not include more complex analysis like e. g. flow analysis.

Furthermore, if the whole program is analyzed, there are many calls without preconditions. One reason is that also native libraries are called, which are not equipped with preconditions.

Chapter 7 - Conclusions

The goal of this thesis has been to develop a static analysis tool, which verifies whether preconditions are guaranteed to hold. Especially for simple preconditions the analysis should be like a filter, which warns a project manager, when precondition compliance is not obvious. A GUI should comfortably present the result of the analysis.

Preconditions, which ensure that a reference is not void, are especially well addressed by the static analysis. As such preconditions are used most often, the analysis can detect many fulfilled preconditions. Furthermore, complex preconditions which are implied in terms of logic rules are supported as well. However, more complex preconditions, including attributes or even function calls are very hard to verify, as they require complex and extensive analysis beyond the scope of this thesis. Therefore, a hybrid approach is favoured, combining static and dynamic checking.

The GUI supports systematic checking for possibly unfulfilled calls, providing the user a clear overview. The calls can be comfortably managed. We have yet stood aside for incremental use of the analysis, because the GOBO Eiffel Lint tool does not support incremental compilation. Instead, we have focussed on further improvements of the static analysis.

Although there are many possible extensions for the Precondition Enforcement Analysis, it has turned out, that static verification is a useful and important tool in the development process of quality software.

Chapter 8 - User Manual

System Requirements

As the GUI makes only use of the EiffelVision 2 library, the Static Precondition Analyzer is supposed to run under different platforms, but it has been tested on Windows XP only.

Installation

To install the Precondition Enforcement Analyzer, the steps below have to be follows.

1. Download the file **PreconditionEnforcementAnalysis.zip**, which contains:

   - the source code in the “src” directory
   - user manual in the “doc” directory
   - GOBO Eiffel library in the “library” directory
   - **License.txt**, the licence for the software
   - **Readme.txt**, describes the installation for the Precondition Enforcement Analysis, and further information

2. Unzip the file to a directory and set the environment variable

   $PRECONDITION_ENFORCEMENT_ANALYSIS to the chosen directory.

Use of Precondition Enforcement Analysis

When you launch the Precondition Enforcement Analysis, the main window shows up (see Figure 8-1).

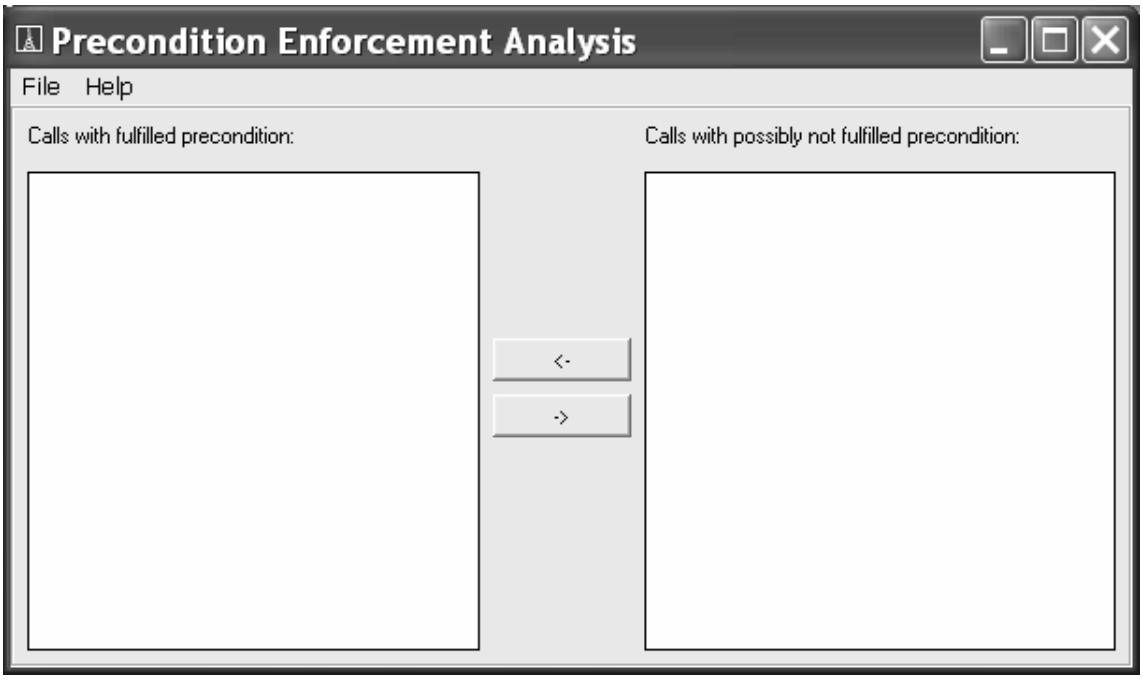


Figure 8-1: Application window, after the Precondition Enforcement Analysis has been launched.

Now you either to click on the Help menu, which pops up a succinct description of the Precondition Enforcement Analysis, you click the File -> Open menu and choose the corresponding Ace file for the program you intend to analyze or you click the File -> Exit menu to end the application. Additionally, the application can be terminated by the X- button.

If you have successfully chosen the Ace file, the Precondition Enforcement Analysis is immediately performed. Consequently two trees of calls are class-wise presented (see Figure 8-2). In the left tree the calls, which are determined to be fulfilled by the analysis are shown. On the right tree the calls, which could not be statically determined to be fulfilled.

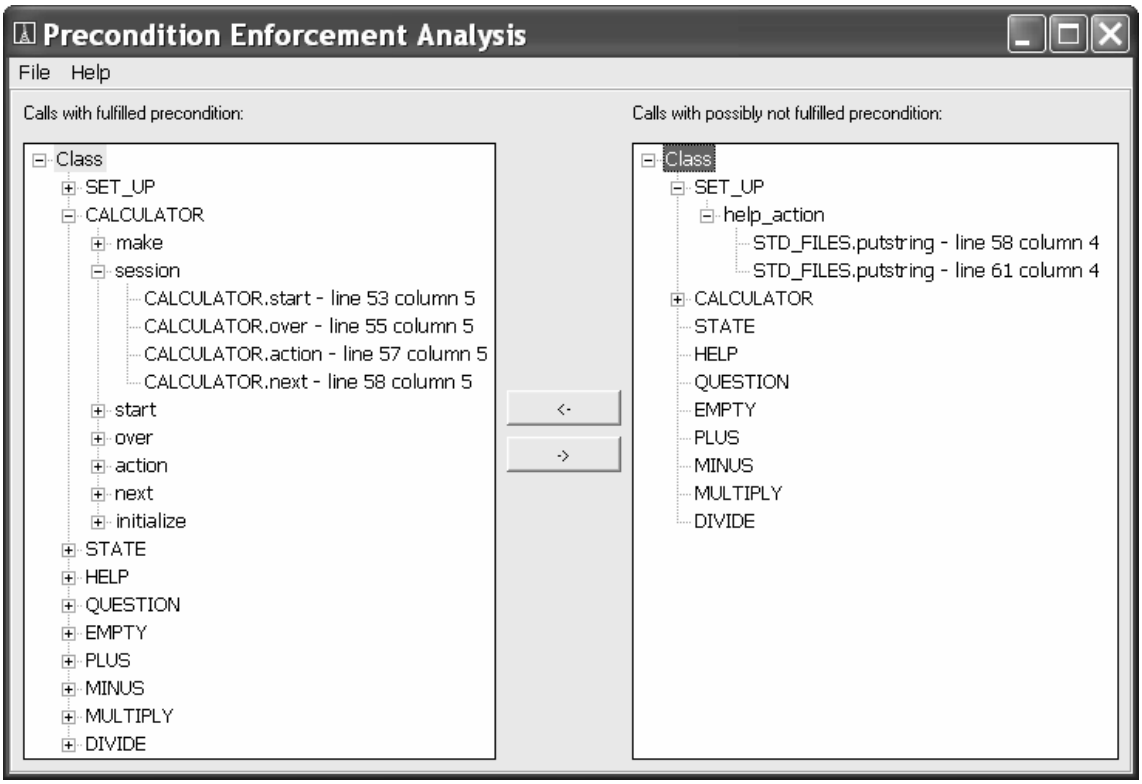


Figure 8-2: Result of the performed Precondition Enforcement Analysis.

Now, you go through the trees, where every leaf represents a call. If you think, that a call, which appears on the right hand side, does fulfill its precondition, you can move it to the other tree by selecting it and pressing the corresponding button. The first item in the tree "Class" gives the root, where its children are all classes, which have been analyzed e.g. the class *SET_UP*, *CALCULATOR*, etc. Below the class *CALCULATOR* are the analyzed features, e.g. *session*. After *session*, we have the item "*CALCULATOR.start* – line 53 column 5". This item represents the call to the routine start from the class *CALCULATOR*.

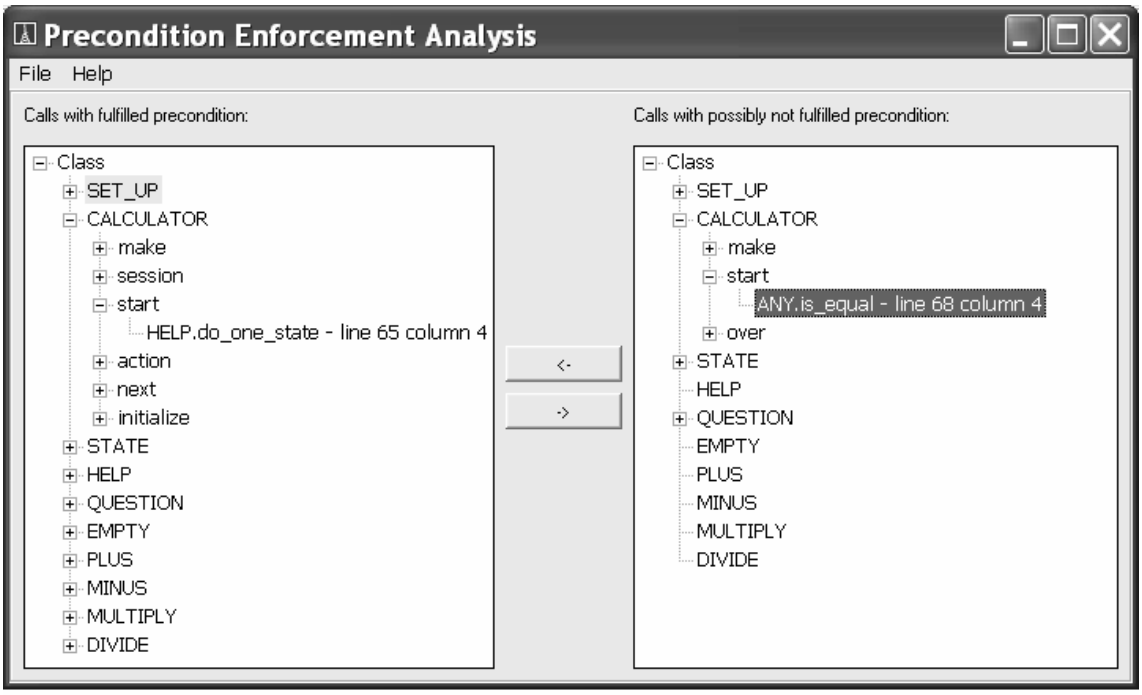


Figure 8-3: Select a call, to move it to the other tree.

Now, you may have decided that the call of *is_equal* in Figure 8-4 is fulfilled, as the Analyzer does not support Boolean functions. Therefore you want to put the call to the other tree. Thus, you select the call (see Figure 8-3) and press the "<-" button. The result is illustrated in Figure 8-5.

```
-- Extract of class CALCULATOR
      start is
                  -- Start session
            do
                  help_state.do_one_state
                  current_state := qst
            ensure
                  current_state.is_equal (qst)
            end
```

Figure 8-4: Extract of the calculator example

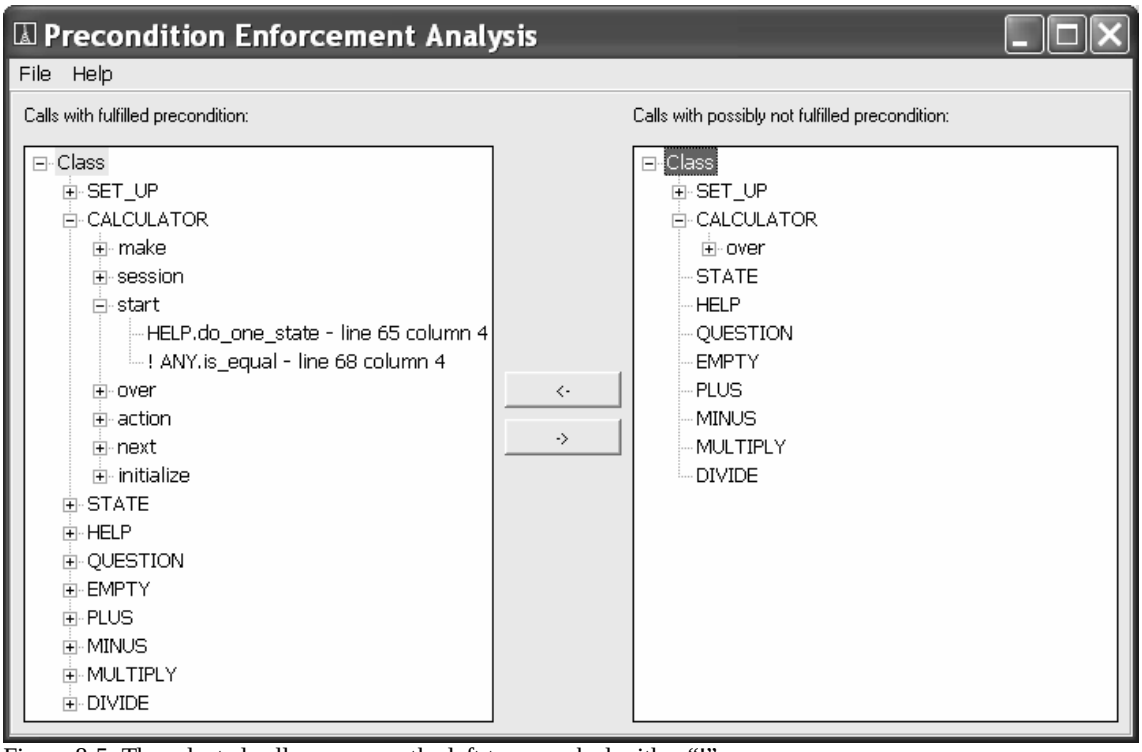


Figure 8-5: The selected call appears on the left tree, marked with a "!".

Figure 8-3 and Figure 8-5 illustrate the move of a call to the other (left) tree.

You should use the Precondition Enforcement Analysis, to carefully look at the calls which have not been determined to be fulfilled and to improve the quality of your software assuring the sound use of routines.

References

[1] Bertrand Meyer: *Object-Oriented Software Construction, 2nd edition*, Prentice Hall, 1997.

[2] Bertrand Meyer*: Eiffel, the Language*, Prentice Hall Object-Oriented Series, 1992

[3] Éric Bezault: *Gobo Eiffel Project*. Retrieved March 2004 from http://www.gobosoft.com/eiffel/gobo/index.html

[4] J. L. Lions (Chairman of the board): *Ariane 5, Flight 501 Failure* Paris, 19 July 1996 http://www.sunnyday.mit.edu/accidents/Ariane5accidentreport.html

[5] Jean-Marc Jézéquel and Bertrand Meyer: *Design by Contract: The Lessons of Ariane* http://archive.eiffel.com/docs/manuals/technology/contract/ariane/page.html

[6] Ken Garlington: *Critique of "Put it in the contract: The lessons of Ariane"*, 7 August 1997, Revised 16 March 1998, http://home.flash.net/~kennieg/ariane.html

[7] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, 1995, Addison Wesley Publishing Company