Master's Thesis

Design and Implementation of the Testing Framework "Tennessee"

Christelle Braun

23rd November 2006

Technische Universität Dresden
International Master in Computational Logic

Supervisor: Prof. Dr. rer. nat. Horst Reichel
Instructors: Dipl.-Inf. Michael Hohmuth
Dipl.-Inf. Hendrik Tews

Declaration

I declare to have written this work independently and without using unmentioned sources.

Acknowledgments

I wish to thank Prof. Dr. Horst Reichel, my supervisor, for allowing me to write my thesis on such an interesting and challenging topic.
I would also like to thank Prof. Dr. Hermann Haertig for the opportunity to pursue my work in the Operating Systems Group. Furthermore, my thanks go to my instructors Dipl.-Inf. Michael Hohmuth and Dipl.-Inf. Hendrik Tews for their greatly appreciated help and guidance.
I thank all other people in the Operating Systems Group and International Center for Computational Logic for interesting and helpful discussions. In particular, I would like to thank Dr.-Ing. Monika Sturm for all her support and help.
Finally, I am very grateful to my family, in particular my parents, Laurent and Bua, for their love and patience, and my friends for encouraging me all the way.

Contents

List of Figures

List of Tables

1. Introduction

Program testing can greatly improve the quality of a software by revealing errors contained in the code before its production use. The general idea is to add to the initial source code elements that yield the error-prone situations while providing feedback on their origin and on their severity, so that the code can be corrected and the future occurences of the same errors prevented.

With the increasing size and complexity of today's applications, these testing capabilities have become a significant sales argument, justified by the high cost that the need to recover from a late-discovered bug can imply. Dramatic examples of such situations in the history of software engineering could have certainly been avoided by the integration of a testing stage in the development process.

Despite this statement, testing remains a task that programmers are reluctant to accomplish and which is therefore still widely neglected by today's development teams. The main reason is the cost in time, money and computer ressource involved in the configuration and maintenance of an efficient testing strategy. Software testing, while not directly contributing to give more functionality to the application itself, was shown to typically consume at least 50% of the total costs of developing software [1]. An additional argument against program testing is that it can never, even by validating the best tests, constitute a proof of correctness, only achievable by formal proving. Finally, the new concepts (inheritance, encapsulation...) introduced by the object-orientation paradigm, whose popularity has constantly increased in the last years, complicate program testing by restricting access to objects which should belong to the tested data.

On the other hand, program testing seems, in the current community of program developers, to take a preferred place for strengthening the reliability in softwares than formal methods, harder to put into practice. On the contrary to static strategies, dynamic testing takes additionally into account the compiler, operating system and program environment under concern, thus simulating with more credibility real-life situations.

In this thesis, we describe the design and implementation of *Tennessee*, a testing framework for C^{++} modules. The properties of the program under test, specified in form of abstract annotations in the source files, are captured and translated by specific components integrated in *Tennessee* which can then generate, in a fully automatic manner, executable test cases that check during its execution the compliance of the program with its expected behaviour.

In Chapter 2, we discuss the concepts and approaches related to program testing that are of interest for this work, before the existing technologies and related work in this field are pointed out in Chapter 3. The design of the framework *Tennessee* is described in Chapter 4, before specific details related to its implementation are given in Chapter 5. In a last part, the evaluation of the performance of *Tennessee* is outlined, before we discuss future developments and conclude our work in Chapter 8 with a summary of the achieved results.

2. Background

2.1. *Design By Contract*

Testing in an independent manner a function or class in a program is called *unit testing*. It is used to check whether the expected behaviour of this single element is indeed satisfied at run-time. This approach follows the so-called *Design By Contract* philosophy, a strategy first sketched by Hoare in 1969 [2] for developing more robust programs . The main idea behind this approach is that a class or method belonging to the program has a *contract* with the program user. This contract specifies the properties that should hold before the call to the method and after its execution. Whenever the client ensures that the method is called in a state fullfilling the specified conditions, its execution should be guaranteed to lead to a state in which some properties, also specified by the contract, are satisfied.

According to these reciprocal expectations and depending on the point of execution in which they are supposed to hold, two categories of conditions are distinguished. The *preconditions* are conditions to be fulfilled by the client in the prestate of the method. The *prestate* of a function corresponds to the last state preceding the function call, when the formal parameters have already been instanciated but the body of the function has not been entered yet. It is therefore not the fault of the function when a precondition is not satisfied.

On the other hand, the *postconditions* are supposed to hold in the *poststate* of the function, reached immediatly after the end of the execution of the function body but before the result value (if any) is returned. A postcondition is therefore a direct requirement to the function itself.

With such conditions, detailed information about the right and responsibilities held both by a function and by its caller are available. The distinction between pre- and postcondition failures makes it possible to clearly assign the responsibility of any detected error, which corresponds to a behavioural failure on the side of the caller or of the callee of a function.

This approach requires however the existence of a description of the expected behavior of the classes or functions under concern. The traditional way to achieve this is to write these properties within annotations embedded in the original source-code. They take the form of *abstract specifications* that describe what should happen whenever this function is called but without relying on its actual implementation. Nowadays however, few programs provide for an abstract specification, mainly because of the additional work it represents for the programmer to find the relevant conditions of a method and to learn the appropriate syntax for expressing them. On the other hand, giving such high-level descriptions is the price to pay for remaining independent from the implementation, which also means that the specifications hold in the long-run, without regard on updates or slight modifications performed on the source code.

Several formalisms have been defined for writing the specification of a program. In the following, we give an overview of the most widely used of them, along with their main advantages and drawbacks.

2.2. Formal specification languages

2.2.1. The early abstract languages

The development of the first *specification languages* used for system and requirement analysis in computer science started for about half a century ago. The notation Z [3] and VDM-SL [4] are famous examples of such formalisms and were mainly used for the creation of proofs of program correctness.

Because it describes the system at a higher level than a traditional programming language, a specification language is in general not directly executable. Practical use may however imply the need of translating it into an executable form, a process called *refinement*.

The development of *program testing* for finding bugs in the early stages of the coding process triggered increasing interest for formal specifications, which provided a detailed description of the functionalities of a program. For applications different from proof for correctness and program verification however, the need for a more practical formalism became soon a priority and lead to the creation of new specification languages.

2.2.2. The new approaches

Up to the eighties, two new formalisms were developed. The first family remained close to the *Design By Contract* philosophy. Meyer [5] gave for instance the behavioral specification of a program written in the language Eiffel in the form of Hoare-style pre- and postconditions. Similar appraoches and new notations followed for many other programming languages (C [6], C^{++} [7] [8] [9], Java [10] [11] [12], Python [13] [14]). They had in common the integration of expressions taken directly from the source-code programming language, thus directly executable and less intimidating for the programmers. The price to pay for this practicability was a restricted expressivity which limited the degree of abstraction and the completeness of the specifications.

The second family, the Behavioral Interface Specification Languages (BISLs), still favored the definition of abstract, precise and complete specifications but strived to keep them as instinctive as possible. Even if the family of the Larch languages [15] -the most famous examples of Behavioral Interface Specification Languages- became quite popular in the late eighties, they still suffered from a lack of use in practice, directly implied by their lack of executability.

2.2.3. JML

A better trade-off between executability and expressivity was achieved by Gary Leavens at Iowa State University, who presented for a few years a new Behavioral Interface Specification Language called JML [16] [17]. Executable statements written in Java are combined with a rich set of *specification expressions* (quantifiers, logical connectives...) and *abstract assertions* that preserve a very high level of expressivity for the specification. Advanced concepts such as *inheritance*, *subtyping* and *visibility restrictions* are also supported by JML.

JML won increasing interest in the last years and many tools based on it were developed, in particularly in the field of *program testing*. Popular examples are ESC/Java [18] and the JML release[1] from Iowa State. It includes an assertion checker for JML specifications, that we will describe later with more precision.

[1] http://www.cs.iastate.edu/ leavens/JML/index.shtml

2.3. Assertion checking

2.3.1. Motivation

Program testing aims at checking whether the specifications of a program are indeed satisfied when it is executed. Errors detected at this stage can then be corrected early in the development process, which helps debugging the code [19], facilitates the checks for non-regression and improves the final quality and reliability of the released program [20]. The main difficulty with this run-time evaluation is however the translation step that needs to transform the formal specifications into an executable form. We will see in this thesis ways to solve this issue and that allowed the development of efficient tools for program testing such as automatic run-time assertion checkers.

2.3.2. Run-time assertion checkers

Current testing tools that rely on the analysis of formal specifications take advantage of their rigorous syntax to extract and translate the information they contain in an inductive manner that is supposed to prevent the apparition of semantical discrepancies between original and translated specifications.

Once the executable assertions have been computed, it remains to find a way to integrate them in the program under concern, so that its properties are checked at run-time. Several strategies have been proposed:

- *preprocessing*: The executable assertions are directly inserted in the source code at the appropriate places in a preprocessing step and get compiled with the original code in the later stages of the computation.
- *compilation-based approach*: The assertions are gathered in an independent executable unit of the programming language that can be called from the original source code after it has been included to it.

The *compilation-based approach* is usually cleaner then the *preprocessing* one and was chosen for the JML Assertion Checker, a tool included in the JML release. It is used to check automatically at run-time the validity of JML assertions (preconditions, postconditions and class invariants) specifying Java programs, while at the same time generating HTML documentation. These tasks are achieved in two steps: first, a *translator* converts the Java source code into a new version of the program embedding assertion checks. Then, the *runtime system* allows the programmer to specify further assertion checking options. Since the source code is not modified during this translation, the original functionalities of the classes and methods are preserved.

Providing for an efficient assertion checker is a significant motivation for the programmer to write specifications in his source code, because he can immediately check the validity of the properties he is interested in and is thus rewarded for the time he spent annotating the code.

2.4. Automatic test generation

2.4.1. Automation of unit testing

As the benefits of program testing for improving the development process gained always more adepts in the computer science community, tools and frameworks spread around in order to automatize to the extent possible the generation, execution and analysis of tests.

A tool for unit testing traditionally needs to complete three successive tasks: selecting test input data, executing the test and comparing the result of the test with the expected results. We will see in the following how this has been already achieved and will present at this occasion *Tennessee*, a tool that is able to check the compliance of C^{++} classes and functions with their specifications and which can automatically generate executable tests to check their behaviour in various situations.

2.4.2. First task: selecting input data

Two techniques are typically used in program testing to select relevant input data for the tests. The first one, *structural testing* (aka *program-based* or *white-box* strategy) consists in analysing the source-code and using the acquired knowledge (e.g. the program's control flow) to select convenient input data. Usually, this task is achieved by performing a symbolic execution of the program in order to determine its possible paths and the path traversal conditions. Then, these conditions can be used as filter for the test input data. Because it is quite intuitive and well-suited for automation, this strategy has encountered in the past the most success and the widest use in the industry.

The second strategy, *functional* testing (aka *specification-based* or *black-box* strategy) does not consider the internal structure of the program but relies on its specifications, usually written in the form of formal annotations, to deduce relevant test input data. This requirement for a formal specification is of course the main flaw of the strategy, because of the additional work it means for the programmer.

The analysis of specifications, on which relies the *black-box* approach, may however lead to a more accurate selection for input data which can improve the relevancy of the unit-tests. The extracted information is usually richer than the knowledge deduced from the low-level analysis of the source code of the program performed in the *white-box* approach. Moreover, supporting the analysis of abstract specifications makes it possible to reuse for program testing specifications that have been written previously for another purpose (such as formal proving [21]) and that are already available with the program.

2.4.3. Second task: producing a test oracle

A *test oracle* is a mechanism which produces the predicted outcomes that has to be compared with the actual outcomes of the program under test and that determines whether a test is successful or not. In the *specification-based* approach, the predicted outcome of the tests corresponds to the validity of the assertions specified in the annotations. Thus, the formal specifications themselves can be seen as the test oracle and its answer is the result of the execution of the tests corresponding to these specifications.

As a result, automating the test oracle amounts to automatically check whether the specifications for a target function are indeed true when the function is executed. The specifications have to be resolved to a truth value semantically equivalent to the conjunction of the results of their evaluation at run-time. We have already seen that tools such as the JML Assertion checker currently exist that efficiently achieve this translation without additional help from the programmer.

2.4.4. Third task: executing the tests

Once the input data and the expected results for the tests are known, it remains to start and control their execution. The code for assertion checking of the function under concern must be called at the right places: checks for preconditions validity in the prestate and for postcondition validity in

the poststate of the function. If class invariants are also present in the specification, they must be checked in both states of the method.

Two different approaches can be used to automate this task [22]:

- *In-line approach*: The code for the assertion checks is directly inserted in the body of the target function.
- *Wrapper approach*: Different new methods performing the assertion checks are defined and included in the source code. Then, calls to these functions are inserted at the appropriate places in the body of the original target function.

The main drawback of the first approach is that it is not always clear how to insert directly postcondition checks (and deal with the `return` statement) within the body of a function. The *wrapper* approach also makes it easier and cleaner to extend the framework in the future with additional checks by just including additional function calls in the body of the target function while their definition can be written at a separate place.

3. Related work

3.1. Automated testing

A large body of research has focused on automated testing to reduce the cost of software development without relying on the labor-intensive processes of manual testing. Tools automating the generation of test cases and using the specification-based approach have been implemented for many different formalisms: Z specifications [23] [24], UML statecharts [25] [26] or ADL specifications [27] [28]. In these cases however, the generated test cases are not written in a common programming language such as Java or C^{++}, which limited the diffusion of these tools. The TestEra Framework [29] addresses this problem by generating Java test cases from Alloy [30] specifications. Since an Alloy Analyzer (AA) is already implemented [31], it can be used to automatically generate method inputs and check correctness of outputs. But the programmer must be familiar with the Alloy specification language which is much different from Java.

The development of JML, introduced in the previous section, lead to the design of frameworks for specification-based testing entirely relying on the Java programming language, and therefore much more attractive for the programmers.

3.2. JML/JUnit and Korat

For the Java programming language, the combination of JML and JUnit has been widely used for specification-based testing. Cheon and Leavens [32] designed automatic translation of JML specifications into test oracles that can be passed to the JUnit framework. The generated test cases consist of a combination of calls to the tested methods with various parameters values. One drawback is however that the base instance and the parameters must be supplied by the user, who may well forget or ignore interesting values.

As improvement, Korat [33] enhances the JML/JUnit framework with capabilities for automatic test-case generation. This tool comes close to *Tennessee*: in both cases, possible candidates are automatically generated and checked against the method's preconditions. Only the candidates which pass this test are used as input values in the generated test-cases. The main drawback in this strategy is however that the domain of validity for the values initially selected may be extremely large, so that the probability to reach a candidate revealing an error remains small. As a result, both tools include a few strategies for restricting the set of values of the initial domain. The candidates removed are either values that have no chance to appear as input parameters in real-life, or that have no influence on the result for assertion check validity.

Given a formal specification for a method, Korat first constructs a Java predicate from the method's preconditions. Then, using a *finitization description*, it generates all nonisomorphic inputs satisfying the predicate for the tested method. A *finitization description* specifies bounds on valid domains for the input parameters. It can be defined by the user or computed automatically by Korat from the parameter declarations in the source file. Passed to JUnit, the generated test-cases are executed while the method postconditions are used as test oracle to check the validity of the output.

Tennessee also relies on the preconditions of the tested method specified as *Cpal* annotations in the source file to find valid input data for the instanciation of the test-cases. The Java predicate computed by Korat and implementing the semantics of the preconditions corresponds in *Tennessee* to the generated precondition check function. And the testing specifications such as the finitization description can be performed in the configuration files of *Tennessee*.

The main difference between the tools concerns the way strategic input values are initially chosen. In order to compute the Java predicate `PreOK` for the precondition validity, Korat actually expects that the implementation of the tested class complies with "good programming practice" and provides for a method (usually called `repOK` or `checkRep`) checking whether the class invariants for this class are satisfied or not, and returning the corresponding Boolean value. The predicate `PreOK` is directly resolved in a call to this method. With this strategy, the tool does not need any knowledge on the semantics of the preconditions and their validity is checked in a *black-box* manner.

In order to deduce constraints on the input parameters, Korat instruments the predicate `PreOK` so that the input parameters which are not read during the evaluation of `PreOK` are labelled. Whenever the result of the check is true and some parameters have been labelled, Korat deduces that they had no influence on the result and uses this information as feedback to reduce the state space of combination of parameter candidates.

Tennessee, on the other hand, takes advantage of the syntax of the preconditions written in *Cpal* to analyze them and try to deduce range constraints for some of the parameters. With this strategy, the search space can be directly restricted, before the method for precondition check is called for the first time. This may be of advantage compared to Korat, which, initially, has no way targetting a strategic combination of input values and only deduces the constraints gradually when execution performs. Moreover, while *Tennessee* supports testing sequences for methods on different base instances, the test cases constructed by Korat consist of a single instance construction and one method invocation on this instance.

3.3. Other tools

3.3.1. Testing tools for C/C^{++}

Tools and frameworks for unit testing that rely on similar approaches than JUnit have also been developed specifically for the programming languages C/C^{++}(CppUnit [34], CppTest [35], Boost [36] for a few popular Open source examples). Some of them, such as C++-test, also provide facilities for automated testing. This software developed (under license) by Parasoft can automatically generate and execute unit tests for C and C++ programs and provides furthermore easy ways to add user-defined test cases.

T++ [37] is a C^{++} language extension for system testing of C^{++} code which relies on compiler and debugging information to deduce relevant values for the test cases it automatically generates. For each class defined in the program, a set of *typicals* is computed that is deduced inductively, starting from the data-type definition and applying the rules of a simple *typical algebra*. The user can additionally define new sets of typicals or modify generated ones.
This strategy has a similar purpose than the search for critical values performed by *Tennessee*. However, without additional precisions from the user, T++ exclusively relies on information derivable by the compiler and based on the underlying data-types. This does not include the knowledge on the behaviour of the tested member that can be found in specifications such as the *Cpal* annotations and that may give hints about the values which have more chance to occur in real-life executions of the program.

In order to better targets relevant values, T++ allows the user to (re)define inside *tuning blocks* the set of typicals at any scope of the program. Test scripts can also be written to iterate over typicals sets.
On the contrary to *Tennessee*, the primary purpose of T++ is not to check the compliance of a program with its expected behaviour. Rather, it works on a lower level (not on the interface) and creates a knowledge base of the source code using the facilities of the compiler. This approach could however be advantageously integrated into tools such as *Tennessee* to improve their capability of generating relevant values for a C^{++} module, starting from the source code itself or from the specifications of its functionality.

3.3.2. Tobias

Tobias is a combinatorial testing tool for the automatic production of large test suites for Java programs. The generated test cases are derived from *test patterns* which give an abstract description of the test cases. They need to be associated to an automatic oracle which is typically based on executable specifications such as JML for Java. The main problem of Tobias is the *combinatorial explosion* which may happen as soon as the test cases consist of more than a couple of method calls. *Tennessee* prevents the occurence of combinatorial explosion by avoiding full traversing of the search space for the instanciation of the test case parameters and relies, whenever possible, on a strategic filtration of the irrelevant values.

3.3.3. Jartege

Jartege (*Java Random Test Generator*) is a tool derived from Tobias for the random generation of unit tests for annotated Java classes. Because the functionalities of Jartege come close to those of *Tennessee*, we compared the generation of test cases performed by both tools. The results of this evaluation is presented in Section 6.2 where Jartege is also described with more details.

4. Design

4.1. Introduction

4.1.1. The challenge

The benefits of integrating program testing methods relying on the philosophy of *Design By Contract* in the process of software development was discussed in the previous Section. The conclusion that a large part of this process is amenable to tool automation motivated the development of *Tennessee*, a tool that can be used to test in an automatic manner program modules written in the programming language C^{++}.

The requirements that an automatic testing tool relying on the *Design By Contract* approach should consider can be summurized into four main goals:

1. **Definition of the contract**

   The specification of the program, written by the client in form of annotations, needs to be univoquely interpreted on both sides, which imposes the use of an unambiguous specification language. This language should be expressive enough to allow an instinctive specification of the properties and behaviour of the methods under concern. Because traditional programming language such as C^{++} do not comply well with this goal, a formal abstract specification language is preferred.

   As a result, a new formal specification language called *Cpal* (*C^{++} Annotation Language*) was defined to express the conditions of the contract between the client of a C^{++} module and the classes and functions this latter contains. The grammar of *Cpal* is given in the Section 4.2.2 on page 30. The *Cpal* specifications are parsed by the *Parsor*, one of the main components of the tool *Tennessee*, which retrieves from them the information that will be used to generate the corresponding executable assertions.

2. **Translation of the contract**

   The way the translation from formal specifications to executable assertions is achieved by the testing tool deserves particular attention because the semantics of the abstract language (here *Cpal*) needs to be preserved to the extent possible in the target programming language (here C^{++}). This step is not straightforward, because complex logical constructs (such as quantifiers) supported in *Cpal* do not have direct counterparts in C^{++}. A strategy has therefore be designed, which specifies the behaviour the tool should adopt when faced to such situations.

3. **Check for the validity of the contract for a given program in the target module**

   In order to check the behaviour of the function when the program is executed, the translated assertions need to be integrated in the execution flow at the right place, so that properties

are indeed verified in the state (prestate, poststate) they are supposed to hold according to the specifications. The purpose of the translation of the contract in an executable form is to allow an *automatic* interpretation of it. A new program intergrating these checks has therefore to be computed and executed in place of the original one. It should simulate an authentic execution of the program while at the same time checking whether the properties of the contract are still valid or not. These checks should moreover remain *transparent* unless the violation of the validity of an assertion occurs.

4. **Check for the validity of the contract for any program in the target module**

   The last challenge a tool developed for checking validity of a specification should support is to confirm whether any program relying on the module the contract applies to is valid with respect to its specifications or not. In other words, examplary programs using the modules under contract should be automatically generated and checked against assertion validity.

4.1.2. *Tennessee*

Tennessee adresses each of these issues including the last one, generating automatic validity checks for an annotated module given as input to the framework (target module). The overall computation of *Tennessee* can be split into three parts, where each of these steps is performed by one of the three main components of *Tennessee*: the *Parsor*, the *Traductor* and the *Generator*. The architecture of *Tennessee* is illustrated on the Figure 4.1 on Page 29. First, the annotated source code is read by the *Parsor* which extracts and classifies the annotations embedded within it. Then, the *Traductor* analyzes this information and translates it in a valid executable form which is, as far as possible, reduced to a minimal form capturing without redundancy all the assertions specified in the source code. Finally, the *Generator* computes test cases on the basis of strategies that are supposed to increase the relevancy of the tests for catching errors in the initial program.

The execution of a test case generated by *Tennessee* leads to one of the following outcomes:

1. Normal program termination:
   The program under test satisfies all the assertions in its specification and completes its execution successfully.

2. Assertion violation exception:
   At least one of the assertions is violated and an assertion exception of the corresponding class is thrown.

3. External failure:
   No assertion exception is thrown but the program does not complete as expected.

The two last situations reveal problems in the module under test. The second case (detection of an assertion violation) signals that the specification is not fully satisfied and the error message computed by *Tennessee* summurizes the information that could be collected about the kind of assertion failure that was detected and the execution path which lead to its occurence. Hopefully, this information is sufficient to find the source of the error and correct it. The third case may be harder to solve because it comprises the (numerous...) remaining situations which lead to program failure and that cannot be documented by *Tennessee*.

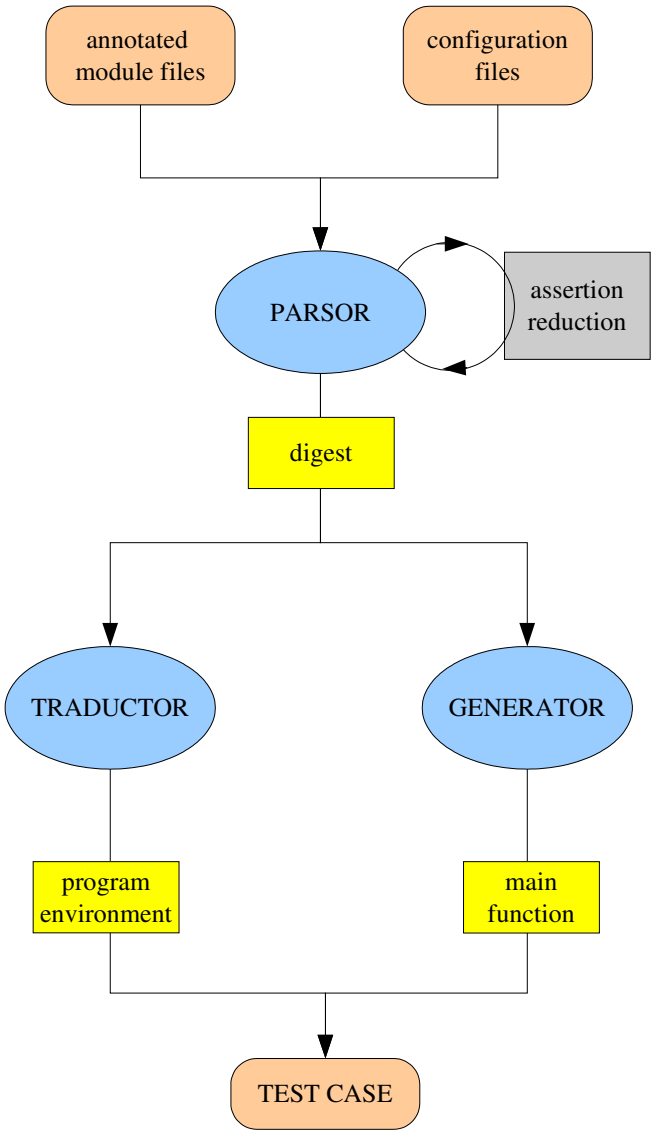

Figure 4.1.: Architecture of *Tennessee*

4.2. The annotation language *Cpal*

4.2.1. Motivation

Cpal, the C^{++} *annotation language*, is a new formal behavioral specification language for C^{++}. It is used to express the contract between the program and its client, following the philosophy of *Design By Contract*. The requirements on its expressivity are two-fold: on the one hand, as formal language, it should define enough *logical constructs*, so that the expressivity of the logic required for the annotations is captured. On the other hand, as behavioral specification language, it should enhance the logical constructs with *specification identifier*s that precise in which situations (prestate, poststate of a method) these conditions are supposed to hold.

The grammar of *Cpal* is given in the Section 4.2.2. Even if its expressivity is expected to meet the user's needs, there may still remain properties that the programmer would rather express in C^{++} instead of writing a formal expression for them. A class may for instance provide for methods that are used to check specific properties and whose calls could advantageously be integrated in the annotations, so that the checks belong to the specification for the method under concern. The language *Cpal* was therefore extended to allow the integration within the formal specifications of expressions written in legal C^{++} syntax that are later included in the generated tests.

4.2.2. Syntax of *Cpal*

4.2.2.1. Notational conventions

The syntax of *Cpal* is presented in form of a Backus-Nauer Form (BNF) grammar with following extensions:

- Terminal symbols are written in type-writer style: `terminal`
- Non-terminal symbols are written in italic: *non-terminal*
- Alternatives are specified with double vertical bars: ||
- Optional elements are surrounded by square brackets: [*optional element*]
- Informal definitions are written in standard font: some additional explanation

4.2.2.2. Grammar of *Cpal*

well-formed_C^{++}_expression ::= an expression which is well-formed with respect to the standard grammar of the C^{++} programming language.

cpp_literal ::= `integer-literal` || `character-literal` || `floating-literal` || `string-literal` || `boolean-literal`

A *cpp_literal* is a *well-formed_C^{++}_expression* belonging to a built-in type as defined in the standard C^{++} grammar.

arithsign ::= $+$ || $-$ || $*$ || $/$

number_expression ::= `integer-literal` || `floating-literal` || `boolean-literal` || *number_expression arithsign number_expression* ||

(*number_expression arithsign number_expression*)

name_expression ::= a *well-formed_C^{++}_expression* that is not a *number_expression*.

data-type ::= a *well-formed_C^{++}_expression* that is a valid C^{++} data-type (built-in or user-defined).

atom ::= [{*data-type*}] *name_expression*

old_atom ::= `OLD`{*atom*}

variable ::= *atom* || *old_atom*

term ::= *variable* || *number_expression*

primitive_condition ::= *boolean_condition* || *container_condition* || *cpp_condition*

cpp_condition ::= a *well-formed_C^{++}_expression* that is evaluated as a Boolean value

boolean_condition ::= *term boolean_sign term*

boolean_sign ::= $<$ || $<=$ || $>$ || $>=$ || $==$ || $!=$

container_condition ::= `Contained`(*term*, *container*)

container ::= a list of comma-separated *well-formed_C^{++}_expression*s that specify a *Cpal* container (see the definition in the Paragraph 4.2.3.2 on page 33).

composite_condition ::= *implication_condition* || *quantified_condition*

implication_condition ::= `Implic`(*predicate*, *predicate*)

predicate ::= *primitive_condition* || *composite_condition*

quantified_condition ::= *quantifier_id*(*data-type* `string-literal` ; ; *predicate* ; ; *predicate*)

quantifier_id ::= `Exists` || `Forall`

4.2.3. Supported logic

4.2.3.1. Overview

In order to be amenable to tool support, the specification language that is used to define properties of program functions should not suffer from semantical problems and the logic it supports should be clearly defined. The mathematical formalism underlying *Cpal* is *multisorted first-order logic*. *First-order logic*, also called *predicate logic*, corresponds to propositional logic enhanced with existencial and universal quantifiers (see [38] for a complete overview). Its expressivity was proven to allow the formalization of virtually all of mathematics, yet not reaching the power of more advanced formalisms such as high-order logics. For the expressivity of the specifications supported by *Tennessee*, it has been decided however to restrict to *first-order logic* enhanced with *sorts*. We see this choice as an acceptable trade-off between the degree of expressivity reached by the constructs of the *Cpal* language and the cost for their automatic analysis.

The formulae of first-order logic are defined by structural induction, starting from a set of *sorts* and *operators*. A *sort* is a set of values that share some common structural properties. Because *Cpal* is used to annotate programs written in C^{++}, it made sense to borrow to the extent possible elements from this programming language. The sorts defined in *Cpal* have for instance been mapped to the legal C^{++} data-types and given the same names. This relieves the programmer from the need to learn a new family of abstract sorts (required, for instance, when using specification languages such as the family of Larch languages [39]) and makes the annotation process more instinctive.

Besides the sorts, the second basic logical elements in the syntax of *Cpal* are the *operators*. An operator is a map from a tuple of sorts (its *domain sorts*) to a *range sort*. A *predicate* is an operator with range sort `bool`. For the definition of the common operators that have C^{++} counterparts, the C^{++} syntax is reused:

- comparison operators: $<$, $<=\ldots$
- arithmetic operators: $+$ (addition), $-$ (substraction), $*$ (multiplication)$\ldots$
- logical connectives: $\&\&$ (conjunction), $||$ (disjunction), ! (negation)

4.2.3.2. The *Cpal* predicates

Any independent condition in a *Cpal* specification is a *predicate*. The elements allowed as building blocks during the inductive construction of a predicate determine whether it belongs to the family of *primitive_condition*s or *composite_condition*s:

- A *primitive_condition* is a predicate that was constructed without using another predicate. Its composition is therefore restricted to elements that belong to smaller structural categories.
- A *composite_condition* contains at least one other predicate as subelement.

The *primitive_condition*s are the structurally simplest conditions in *Cpal*. They can be further divided into three different categories:

- *boolean_condition*:
  This condition is a binary relation between two *term*s and involves a C^{++} boolean sign as (infix) operator which has the same meaning as in C^{++}.

- *container_condition*:
  The constructor `Contained` was defined in *Cpal* to give facilities for the restriction of the domain a *term* can belong to. The semantics of `Contained` is that the *term* given as first parameter belongs to the set of values (the *container*) specified by the remaining parameters. Because the C^{++} standard library includes several container classes and iterators for them (`vector`, `map`...), our definitions of containers are closely related to these structures. The `Contained` operator is overridden several times to allow alternative definitions of the container part. The different signatures and their semantics are given on the Figure 4.2.

- *cpp_condition*:
  The only requirement for a *cpp_condition* to have a valid syntax is to be a well-formed C^{++} expression resolving to an element of the sort `bool`. With this category of predicates, C^{++} expressions can be integrated in the *Cpal* annotations.

More complex predicates can be constructed using the aforementioned predicates as building elements. These *composite_condition*s can be classified into two categories:

- *implication_condition*:
  The constructor `Implic` is a macro for the expression of an implication between two conditions. The condition `Implic`(a, b) where a and b are predicates, is semantically equivalent to the formula $a \rightarrow b$, which abbreviates $\neg a \vee b$.

- *quantified_condition*:
  The traditional notation for quantification in predicate logic is slightly modified in *Cpal*: the quantifiers, defined as ternary operators, implement *constrained* quantification. A *quantified_condition* is of the form:

  > *quantifier_id*(*data-type variable* ; ; *constraint* ; ; *predicate*)
  > where *quantifier_id* is `Forall` or `Exists`.

  In this notation, the first argument contains the declaration of the variable that gets bound by the quantifier, the second argument, *constraint*, restricts the domain of values the bound variable may belong to, and the third argument is the actual condition that should hold for the possible values of the bound variable. It is therefore semantically equivalent to one of the following logical expressions:

  - if *quantifier_id* is `Forall`: $\forall$ *variable*. *constraint* $\rightarrow$ *predicate*
  - if *quantifier_id* is `Exists`: $\exists$ *variable*. *constraint* $\wedge$ *predicate*

  The reason for constraining the quantification follows the constatation that, in most practical cases, one needs to restrict the permitted values of the bound variable in the specification anyway, so that it makes sense to do it directly in the quantification. If the constraint part is empty, no further restriction is set on the values of the bound variable which may have any value contained in its sort.

4.2.4. Semantical gap between *Cpal* and the user's expectations

The integration of elements borrowed from the native programming language C^{++} into *Cpal* specifications may threaten the semantical unambiguity required for the resulting assertions. The

syntax	semantics
`Contained(x,`*container*`)`	`x` $\in$ *container*, where *container* is an instance of a C^{++} container class
`Contained(x,it_0,it_end)`	`x` belongs to the container delimited by the two iterators
`Contained(x,minval,maxval)`	closed interval: minval $\leq$ x $\leq$ maxval
`Contained(x,-Infinity,maxval)`	semi-opened interval: x $\leq$ maxval
`Contained(x,minval,Infinity)`	semi-opened interval: x $\geq$ minval
`Contained(x,-Infinity,Infinity)`	opened interval: no restriction

Figure 4.2.: Supported *container* definitions for the constructor `Contained`

universe of values a variable of a given data-type may belong to is a typical situation in which a *semantical gap* can occur between the user expectations and the *Cpal* specifications. In the following, precisions on the specifications involving numeric expressions and on the quantifications for instances belonging to user-defined C^{++} classes are given.

4.2.4.1. Specifications involving numeric expressions

Problematic situations typically occur when range constraints are specified for numeric values. If, for instance, a variable of an integer type is involved in an abstract specification, then its domain, without additional constraint, intuitively corresponds to the integral values in the range $]-\infty,+\infty[$. The fact that the *Cpal* sorts are mapped to the C^{++} data-types adds however implicit constraints on the domain of the variables of these types. This comes from the fact that the C^{++} data-type `int` (as well as the other integral types) have actually fixed precision in the finite range `[INT_MIN, INT_MAX]`. Moreover, operators over these types obey rules of modular arithmetic. The specification `(INT_MIN * INT_MIN)>0` is for instance evaluated to `false`, since `INT_MIN * INT_MIN` is equal to zero in this formalism.

Attempts have been made in other specification languages to solve this problem. Chalin [40] defined for instance a new variant of JML that offers support for two arbitrary precision numeric data-types (that do not exist in the Java programming language) called `integer` and `real`. Designing a strategy to "close the semantical gap" in *Cpal* goes however beyond the scope of this thesis and the user is supposed to keep in mind the specificities of C^{++} data-types when writing *Cpal* annotations involving such values.

4.2.4.2. Quantification on class instances

The mapping from *Cpal* sorts to C^{++} *built-in* data-types allows an univoque evaluation of the domain of values under concern. The situation is more complicated when the universe of a class instance comes into play, for example in a quantification involving a class instance as bound variable. Checking whether *there exists* an instance having a given property, or whether *all instances* have this property needs to identify with precision the set of instances that constitute the universe of the bound variable at this step of the program execution.

Several alternatives exist for the definition of this universe. A few propositions:

- All potentially constructable instances, with respect to all the constructors defined in the program at the time of execution. Here, the objects are in their initial state, meaning that they have not been involved in other operations than those defined in the constructor which was used for their creation.
- Only the currently existing instances at this step of execution. Their state depends here on the operations they were already involved in.
- The instances that exist or have existed at some point of execution.
- All the valid instances according to the C^{++} definition, which additionally satisfy the class invariants.

The approach adopted in *Tennessee* corresponds to the first proposition: the universe of instances at a given point of execution of a program is the set of all instances that may be constructed with the available constructors defined in the class, and which differ in some of their attribute. That is, the checks focus on the differences in the instance attributes, not in their references. If, for instance, the class `Student` has no defined constructor and a single attribute `age` of type `int`, then the universe is the finite set of (`INT_MAX - INT_MIN` +1) `Student` objects, each of one having, for the attribute `age`, a different value contained in the interval [INT_MIN, INT_MAX].

4.3. The *Parsor*

4.3.1. Task

The annotated source code files given as input to *Tennessee* are first passed to the *Parsor*. Its task is two-fold: first, information about the signature of the target classes and functions is retrieved from their C^{++} definitions. In a second step the *Cpal* annotations surrounding these elements are parsed and their specifications classified and stored. The registered information is then passed as input value to the *Traductor*.

4.3.2. Classification of the specifications

The *Cpal* annotations are written either in additional configuration files or directly above the function or class definition they qualify in the source code. In the second case and in order to ensure that they are ignored by the C^{++} compiler, these annotations are embedded into C^{++} comment blocks (i.e. delimited by the signs `/*` and `*/`). Since the *Parsor* needs to distinguish them from other C^{++} comments, additional delimiters (`$++` and `++$`) surround the annotation block on top of a function or a class. The programmer must also add a delimiter of the form `/*$- - - -$*/` after the body of any annotated definition, so that the *Parsor* more easily detects its end. An example is given in Figure 4.3 on page 36 where the (shortened) definition of the member function `append` of the class `Node` is preceded by an annotation block containing two preconditions and one postcondition.

Such an annotation block is composed of a set of *Cpal specification_item*s. They are classified by the *Parsor* according to their *specification identifier*, in order to separate preconditions, postconditions and invariants. Then, inside each of these categories, *primitive_condition*s, *composite_condition*s and *cpp_condition*s are distinguished and each category registered.

```
 /*$++
pre:{${ n!=NULL }$};
pre:{${Forall(int x  ;; Into(x,all_vals());; x != n->val) }$};
post:{${ next!=NULL}$};
++$*/
void Node::append(Node* n)
{
   ...
}
/*$----$*/
```

Figure 4.3.: Annotation block for the function `Node::append`

4.4. The *Traductor*

4.4.1. Task

The *Traductor* is responsible for the generation of a new program environment embedding the assertion validity checks for the specifications extracted from the annotated source code and classified by the *Parsor* in the first step. The resulting program it generates only lacks its main function before it can be executed.

The general translation method of the *Traductor* is to operate on the scale of a function in the original source code. The target functions are sequentially considered by the *Traductor* which computes, for each one of them, a set of helper functions integrated in the modified program. A *helper function* is a function which does not exist in the source code and is created by *Tennessee* for testing purposes. The different helper functions computed for a target function are described in details in the Section 5.1.3 on page 51.

It is very important for *Tennessee* to perform a translation from *Cpal* annotations to C^{++} executable code that preserves the semantics initially intended by the programmer when writing the specifications. In other words, the translation must be *sound*, meaning that no false positive is produced. The term *false positive* qualifies an error detected by *Tennessee* which actually does not exist in the analyzed code. On the other hand, one hopes that the translation performed by the *Traductor* is also *complete*. Completeness means that all the errors contained in the program are detected by *Tennessee*, or in other words, that no *false negative* (existing error that is ignored by *Tennessee*) occurs. Because of the limited expressivity of the specification language *Cpal* and the complexity of today's programs, it is unlikely that a tool like *Tennessee* will ever be able to achieve completeness and detect all possible bugs. Therefore, we only require *Tennessee* to *tend* to completeness and we hope to be able to extend in the future the capablity of the tool and the expressivity of *Cpal* so that it can efficiently detect always more error-prone situations.

On the other hand, the guarantee of soundness remains a priority: while a false negative merely compels the programmer to do additional debugging work in order to find the error, a false positive (signalled error which does not exist in reality) may imply a long and useless debugging process that is mere time loss. The attention will therefore in the following be focused on the way *Tennessee* ensures soundness of the translation.

4.4.2. Execution failures

In the Paragraph 4.1.2 the three possible results of the execution of a test case generated by *Tennessee* were presented. We saw that if the execution does not perform successfully (so the checks

do not remain transparent), then this reveals an error in the program. With respect to the computation performed by *Tennessee* however, an exception signalling the occurence of an assertion violation in the target program can be considered as a normal outcome and only the third case, *execution failure* is not a normal result for *Tennessee*.

Typically, such problematic situations are program abortion or non-termination. If, for instance, a null pointer dereference occurs during the execution of the C^{++} code implementing the assertion checks for a target function, then the program execution is aborted. However, this does not imply that the contract implemented by the annotations for this target function is not fulfilled (and does not imply the contrary either). More precisely, consider a precondition of the form `f(x)==f(x)` where `x` is the `NULL` pointer. The evaluation of this condition in *Tennessee* will lead to program abortion, even if it is true according to the semantics of first-order logic. Such problems and the way *undefined expressions* in general should be interpreted were already adressed in the literature for other assertion checkers [22], [41]. The solutions usually chosen amount to support only an *approximation* of the semantics of first-order logic and try to detect the critical situations and give for them an evaluation that complies as far as possible with the truth value intuitively expected.

For now, *Tennessee* does not perform additional analysis of the specifications to try to find and prevent the occurences of such situations. A possible future extension would be, for instance, to reduce the logical constraints to a logically equivalent form in a prior step (e.g. using a constraint solver), so that conditions such as `f(x)==f(x)`, whose validity can be deduced by syntactic analysis, are recognized and removed from the set of specifications which are executed by *Tennessee*. For now, it is left to the programmer to avoid execution failures and any pre- or postcondition that is not evaluated to true by *Tennessee* when the check is performed is considered as a violated specification.

4.4.3. Detection of internal assertion failures

The translation may also be complicated by the occurence of *internal assertions*. As soon as a target function calls another annotated method which was also translated by *Tennessee* (the target function itself in case of recursion), then we say that the assertions of the nested method are internal assertions for the target function. In this case, it is necessary for *Tennessee* to distinguish between the (external) preconditions of the target function and the (internal) preconditions of the function it calls in its body. Indeed, these two kind of preconditions do not have the same meaning for the target function. While the responsibility for the violation of an external precondition is given to the client calling the target function, the function itself has to be blamed for a false internal precondition, because in this case the client of the nested method is actually the target function itself.

It is foreseen to implement in *Tennessee* the detection of internal precondition violation in future development, so that the new target function computed by the *Traductor* throws an assertion exception of a specific class (e.g. `internal precondition error`) whenever a precondition assertion exception is thrown by a nested method. For now however, *Tennessee* allows the user to choose between two different configurations: either the internal assertion violations are detected and signalled but without distinguishing them from the external assertions, or they are ignored, which may lead to the occurence of false negatives.

4.4.4. The translation process

The way the *Traductor* translates a *Cpal* specification (precondition, postcondition, or invariant) into executable form can be seen as a two-stage process: first, the *term*s occuring in the specification are extracted and the value they refer to precisely determined. This can lead to a renaming of

some of the variables and the definition of new elements in the translated conditions. The second step consists in translating the whole sentence of the specification into a semantically equivalent C^{++} expression (containing, if any, the renamed variables from the first step). This expression will be resolved at run-time into a Boolean value which corresponds to the result of the validity check initially specified in the annotations. We first describe with more precision the second step of the translation process before the translation of *term*s is adressed.

4.4.5. Translation of a *specification_item*

The way *Tennessee* translates a *specification_item* depends on the category this specification belongs to: a *primitive_condition* or *cpp_condition* can be directly mapped into a C^{++} Boolean condition. The same applies on an *implication_condition* after a small rewriting step. The remaining *composite_condition*s, which contain a *Cpal* operator not belonging to built-in C^{++} syntax need to be previously resolved before valid C^{++} code can be generated.

The general approach that was adopted to translate these conditions is to define for each *Cpal* operator a corresponding C^{++} function which returns a value of built-in C^{++} type `bool`. These functions are included in the new program environment. At run-time, a check for assertion validity of a *composite_condition* leads to the call of the corresponding function whose returned value directly confirms whether the assertion complies to the specification or not.

The following sections describe the translation process from the *Cpal* operators to the corresponding C^{++} functions.

4.4.5.1. Translation of the operator `Contained`

General translation method

The different possible definitions for the `Contained` operator (see the Figure 4.2 on Page 34) are translated into three different C^{++} functions with the same name (`Contained`) and returning a value of type `bool`, but accepting different kind of parameters (principle of *override*). Their signatures and meanings are as follows (where `T` is the parameterized type of the template function):

- `bool Contained(T val,` *container* `values)`:
  This function returns `true` whenever `val` belongs to `values`. By *container* is meant an instance of a container class as defined by the C^{++} standard library (`vector<T>`...).
- `bool Contained(T val, T first, T last)`:
  The Boolean value `true` is returned if `val` is greater than `first` and smaller than `last` (see the Figure 4.4 for the source code). Note that opened and semi-opened intervals can also be specified with the keyword `INFINITY`.
- `bool Contained(T val, iterator<T> first, iterator<T> last)`:
  The Boolean value `true` is returned if the value `val` is met while iterating on the values between `first` and `last`, which are supposed to be iterators of C^{++} *containers* as described in the first category.

4.4.5.2. Translation of the universal quantifier `Forall`

The *Traductor* may have to resolve specifications involving universal quantification, of the form:

`Forall(T x ;;` *constraint* `;;` *condition*`)`

```
template <class T>
bool Contained(T value, T first, T last)
{
  return ((first<=value)&&(value<=last));
}
```

Figure 4.4.: Translated function for the `Contained` operator accepting as parameters the extrema of the specified domain

```
bool Account::fcpp_forall_0(int amount)
{
    vector< int  > vtmp = all_vals();
    for(vector< int  >::iterator it = vtmp.begin(); it!=vtmp.end(); ++it)
    {
        if(!(  (*it)!=amount ))
        return false;
    }
    return true;
}
```

Figure 4.5.: Example of a *forall-function* generated by the *Traductor*

where the intended meaning is that all the possible values `x` of data-type `T` which satisfy *constraint* should make *condition* true. Here, *constraint* and *condition* are *Cpal predicate*s.

The *Traductor* computes a so-called *forall-function* for every universal quantification found in the *Cpal* annotations. In the body of this function, an iteration is started on the set of all possible values the bound variable may take and for every iteration step, the satisfiability of *condition* is checked. Whenever this check resolves to false, the value false is returned by the *forall-function*. If all the possible values for `x` have been tested, the Boolean value true is returned. The pseudo-code for this computation is given in the Figure 4.6.

The set of all possible values for `x` corresponds to the intersection of the domain of the data-type `T` it belongs to and the set of permitted values with respect to the contraint given as second parameter to the operator. The Figure 4.5 gives the source code of the *forall-function* generated by the *Traductor* for the following *Cpal* assertion:
`pre:$ FORALL(int x;; Contained(x,all_vals());; x!=amount $;`
where `all_vals()` is a function whose return value is of type `vector<int>`. Here the *constraint* is therefore a *container_condition* and the *container* a `vector<int>`.

4.4.5.3. Translation of the existencial quantifier `Exists`

The *Traductor* may also have to resolve specifications involving existential quantification, of the form:

`Exists(T x ;;` *constraint* `;;` *condition*`)`

The *Traductor* generates in this case an *exist-function* with a similar structure than the *forall-function*. The only difference with universal quantification is that evaluation stops whenever `true` is returned since in this case the whole existentially quantified expression resolves immediately to

```
checkTrue(FORALL(T x ;; x in container ;; condition)):
{
    for x in container
        if (condition == false)
            return false
        end if
    end for
    return true
}
```

Figure 4.6.: Algorithm for checking the validity of a universally quantified expression

```
checkTrue(EXISTS(T x ;; x in container ;; condition)):
{
    for x in container
        if (condition == true)
            return true
        end if
    end for
    return false
}
```

Figure 4.7.: Algorithm for checking the validity of an existencially quantified expression

`true`. The pseudo-code for the evaluation algorithm `checkTrue` of an existencially quantified expression is given in Figure 4.7.

4.4.6. Translation of the *term*s

4.4.6.1. The problem

As defined in the grammar of *Cpal* in Section 4.2.2, the elements that are called *term*s may comprise various different structures, from explicit numeric literals to variable names or function calls. In order for a translated assertion which involves such a *term* to be correctly executed, it is first necessary that this *term* belongs to the environment (the scope) of the (translated) target function.

If the *term* is a variable name, there are three different cases in which it belongs to the scope of a target function and can with assurance be used in its *Cpal* annotations:

- Global variables:
  Global variables that are defined in the program before the target function is called belong to its scope and can be used in its annotations. One problem may however occur when a global variable used in a postcondition is actually deleted by the target function itself and does not refer to a correct value anymore at the time the check for postcondition validity is executed. *Tennessee* does not detect such problems and it is left to the programmer to ensure that the global variables used in postconditions indeed exist in the program environment at this step of computation.

- Class attributes:
  If the target function belongs to a class, the class attributes are visible names for the function.

- Input parameters of the target function
  If the target function accepts input values, the name of the formal parameters can also be used in the annotations. On the contrary to the global variables (which may also be visible for other target functions and other classes) and to the class attributes (at least accessible to other methods of the class), the formal parameters are local variables of the target function and can therefore only appear in specifications applying to it.

This requirement that the *term* belongs to the visible environment of the target function is however not sufficient to ensure a correct translation of the *term* with respect to the semantics of the specification: it is also necessary to guarantee that a translated *term* indeed refers to the intended value specified in the *Cpal* annotation. As an example, consider the situation in which the value held by a *term* is the global variable x in the prestate of the target function. Then, using the name x in the postcondition may not lead to the check of the intended value since the test for the postcondition validity is executed in the poststate of the target function. If x is modified by the target function, then the name x refers to different values in the pre- and in the poststate, leading to an incorrect translation of the postcondition. This is all the more dangerous since these semantically incorrect conditions actually remain syntactically correct and would be therefore accepted by *Tennessee* and transparently integrated in the test-case.

4.4.6.2. Old atoms

In order to refer, in the poststate, to values that held in the prestate of a target function, *old_atom*s were defined in the *Cpal* grammar. When an *old_atom* is found in the specifications, *Tennessee* stores the value that the corresponding *atom* holds in the prestate of the target function and replaces in the translated assertions any occurence of this *old_atom* by the name of the variable containing its prestate value. Note that this replacement is only performed in assertions corresponding to postconditions since in preconditions we have equality between the *old_atom* and the value of the corresponding *atom*.

Additionally, if the name of a *function parameter* passed by value is specified in a postcondition, the *Traductor* transforms it in an *old_atom* and uses exclusively its prestate value. A *function parameter* passed by value is indeed a local variable for the target function which can be considered as a black-box operating on its parameters in a way that the *caller* is not supposed to know. Consequently, even if it is not explicitely specified in form of an *old_atom*, only the prestate value can be used by the client in the annotations.

4.4.6.3. Translation of class attributes

The unqualified name of a class attribute is only accessible to member methods of this class. Therefore, if a translated assertion which involves the name of a class attributes is manipulated by a function generated by *Tennessee* in the new program environment and which is not a method of the class the attribute belongs to, a compilation error will occur when the test is executed. To avoid this situation, all the new functions generated by the *Traductor* and used to test a target function are defined as methods of the same class and their declarations are inserted in the class definition in the new program environment.

4.5. The *Generator*

4.5.1. Overview

4.5.1.1. Vocabulary

Tennessee takes as input an annotated C^{++} module and returns executable tests that aim at revealing assertion failures in the module. Before discussing how *Tennessee*, thanks to its *Generator*, achieves the last step of its computation and generates the tests, it is necessary to define the main concepts used in our context of program testing.

- **Test harness and test-scope**
  A *test-harness* is a set of *test cases* (see the definition below) applying on a set of module members, which constitutes the *test-scope* of the test-harness. In other words, the *test-scope* represents the set of functions that have to be tested in the module. To each test-harness corresponds one and only one test-scope. In order to simplify its definition, a test-scope can be specified with the following elements, which all resolve to a set of target functions:
  - A list of target functions: all these target functions are directly inserted in the test-scope.
  - A target class: all the methods of the target class are inserted in the test-scope.
- **Test case and test-sequence**
  A *test case* is a single executable test computed by *Tennessee*. It contains a *test-sequence* which specifies the tested target functions and the order in which they are called. For a given test case corresponds one and only one such sequence, and the smallest test-sequence that may be specified consists of a single occurence of a target function. Note that all the target functions must belong to the test-scope of the current test-harness. A test-sequence can therefore be seen as the simulation of a unit of execution that could occur in a real-life application based on the same interface. Concretely, every main function computed by the *Generator* of *Tennessee* and embedded in the new program environment implements a test case.

4.5.1.2. State of the computation

After the *Traductor* has output the new program environment it still remains, before a test can be executed, to add a main function which contains the function calls corresponding to the test case for the module under concern. The *Generator*, the component in *Tennessee* responsible for this task, tries to automate to the extent possible the efficient generation of relevant test cases for the program under test.

The Section 2.4.1 pointed out the three main tasks a testing tool has to complete in order to generate a test case:

1. Selecting input data
2. Producing a test oracle
3. Executing the tests

Considering what has been achieved so far, three conclusions can be drawn:

- The first task still needs to be adressed.

- The second task was already performed in a previous step.
- The completion of the third task will be straightforward as soon as the first task is fulfilled.

The second conclusion comes from the fact that the assertion checks computed in *specification-based* approaches can be considered as a test oracle. In other words, the second task was completed by the *Traductor* and the test oracle is at this time available for the *Generator*. About the third conclusion, the integration of the test case in the form of a main function in the new program environment which embeds the assertion checks suffices to ensure that the execution of the target functions will be automatically compared at run-time with their expected behaviour. Moreover, any assertion violation will be automatically detected and a corresponding exception thrown. Eventually, the error message returned to the client should contain enough information to locate the code responsible for the failure and to retrace the execution path which lead to its occurence. Hopefully, this suffices to correct the code and make it more compliant with its specifications.

All in all, this shows that the task left to the *Generator* is actually the selection of input data for the initialization and configuration of the test case. This process can be split into three successive steps:

1. Generation of the test-sequence
2. Instanciation of the parameters of the target functions
3. Generation of the base instances for the target functions

Before we consider each step with more details, we first explain in the two following paragraphs the motivation and strategies that will orient the *Generator*'s computation.

4.5.1.3. Motivation and goals

Relying on the signatures of the target functions stored by the *Parsor* in an earlier step, it is not difficult for the *Generator* to compute an executable test case which as expected tests the module under concern. This merely amounts to compute an arbitrary sequence of calls to the target functions, to deduce from the data-types of their parameters some valid combinations of values and to apply the calls on instances constructed at the beginning of the test case with the available constructors for the classes defined in the module.

Unfortunately, such a test case has little chance to reveal a bug contained in the program, especially if the number of execution paths that lead to its occurence is not very large. On one hand, several target functions accepting various parameters are usually involved in the module under test. On the other hand, the domain of valid values that the current computer architectures support for the C^{++} built-in data-types are very large (e.g. 2^{32} different values of type `int`). Therefore, an arbitrary sequence of calls to the target functions with a random instanciation of their parameters is very unlikely to reproduce the specific execution paths that would reveal the presence of the bug.

What is actually expected from the *Generator* is to compute *relevant* test cases, i.e., test cases that have a non-negligible probability to detect a bug contained in the source program. In other words, the test cases should have a good chance to reproduce an execution path that leads to program failure. The only way to effectively achieve this goal is to rely for the test case generation on additional information about the module under concern and giving hints about, for instance, the most typical sequences of functions that may be called or the restricted range of values a

given parameter may belong to, so that the test case has a higher probability to simulate a real-life execution. Actually, any information which permits to reduce the degree of randomness of the instanciation of the test case parameters is useful and may increase its relevancy. Basically, three different kind of data which could help in this sense the generation of more realistic test cases can be distinguished:

1. Sequence of function calls
   The first question that must be solved in order to generate a new test case is the choice of the target functions that are called and the way these calls should occur. In other words, the *Generator* has to determine the sequence of function calls that will be contained in the test case and that should reflect a realistic execution path.

2. Involved base instance
   A base instance is the instance of a class defined in the module and on which is applied some methods in the test case. *Tennessee* is looking for information that could help the *Generator* to decide which base instance to use for a given call in the test case: population of instances existing at a given step of the computation, turnover, construction strategy (which constructor, which parameters...).

3. Input values for the parameters
   Any function call in the test case to a target function that accepts parameters raises the question of their instanciation. The *Generator* needs to know from which domain the values of a given parameter are taken or, on the contrary, from which domain these values cannot come. This information reduces the risk of generating *irrelevant* test cases, i.e. test cases that have no chance to simulate a real-life execution of the program.

Trying to gain more knowledge about these three kind of data raises however the question of the sources of information available to *Tennessee* besides the signature of the target functions and that may be queried for such additional information. The following paragraph, will present these different sources and describe the strategy of the *Generator* to retrieve information from them.

4.5.1.4. Sources of information

Two main sources of information that can help the *Generator* build relevant test cases for an annotated C^{++} module can be distinguished:

1. The specification
   Based on the information about the target functions already collected, *Tennessee* may know more details about the properties and functionality of the module than the limited information provided by the signatures of the target functions found in the source code (which is restricted to the name of the functions and of their formal parameters and the data-types of their parameters and of their return values). This precious additional source of information consists of the *Cpal* specifications that were parsed and processed by the *Parsor* in an earlier stage of *Tennessee*'s computation. In general, the kind of data useful for the test case generation and that typically appear in the annotations of a target function corresponds to the third category (input values for the parameters). The specifications that give hints about the permitted domain of values of a parameter will be called *range constraints* in the following. In the *Cpal* syntax, *boolean_condition*s and *container_condition*s comply particularly well with the expression of range constraints.

2. The client
   Even if the primary goal of the *Generator* is to automate the generation of relevant test cases using only the source code of a C^{++} module and its specification, it is not realistic to believe that all the knowledge of the client about his module have been translated in the annotations and/or are deducible in a systematic manner by *Tennessee*. Therefore, besides the capability of generating automatically test cases that follow a predefined strategy and that take into account the information extracted from the specification, *Tennessee* should also give the user the possibility to directly interfere in the test case building process. The general approach of *Tennessee* is to predefine all the parameters that are needed to generate test cases while giving the user the possibility to easily overwrite or extend them in order to configure the test cases according to his needs. This is achieved through configuration files which define all the variables that may be instanciated by the user to parameterize the tests. When a test case is computed, *Tennessee* always gives priority to the user-defined values to configure the tests and completes the missing parameters with their default values.

The following considers more in details the strategies used by the *Generator* to complete the three tasks identified at the beginning of this section.

4.5.2. Generation of the test-sequence

The most important parameter of a test case is its test-sequence, i.e., the sequence of target functions that are called for assertion validity checks. Following the general philosophy of *Tennessee*, the user can either let the testing tool generate a test-sequence in an independent manner with its predefined strategy, or control with more or less precision this computation, so that the test-sequence complies with his needs. We will first describe the default strategy of *Tennessee* and see in a second step in which manner the user can modify and orient this computation.

4.5.2.1. Predefined computation

For the test-sequence, the default behaviour of the *Generator* is to assume that all the methods initially registered by the *Parsor* (i.e. all the annotated methods defined in the C^{++} module) should be tested. The test-scope of the current test-harness is therefore instanciated with all the registered methods for the current module under test.

Then, it remains to precise the length of the test-sequence (the number of function calls it contains) and its composition (which functions are called and in which order). In order to avoid the systematic generation of test cases of the same length, *Tennessee* chooses determines randomly this parameter (but with a predefined upper bound of 50 function calls). The test-sequence is then generated with the corresponding number of calls to functions that are picked randomly from the test-scope.

4.5.2.2. User-defined parameterization

The user can influence three different aspects of the test case generation. First, the test-scope can be restricted to a subset of all the annotated functions present in the module, so that the tests apply only on the selected structures. While the second parameter, the length of the test-sequence, can be defined by overwriting a single variable in the configuration file, several possibilities are left to the user to configure its composition. *Tennessee* supports the specification of this sequence in form of a regular expression (which may include infinite iteration), so there is practically no restriction on the variety of sequences that may be generated, and their degree of randomness can range from undeterminism to full determinism.

4.5.3. Generation of input values

4.5.3.1. General strategy

With the signatures of the annotated functions, the *Generator* has access to the data-type of any given parameter of a target function called in the test-sequence. For instantiation of such a parameter, it remains however to determine its domain of validity for this target function, i.e., the values of this data-type that should be considered as valid candidates for this function call.

This step is particularly important because the values used as parameters for the function calls determine the execution path followed at run-time and may therefore significantly influence the relevancy of the generated test case. In the practice indeed, the role and purpose of a function usually restricts the domain of relevant values for a given parameter to a subset of the whole domain defined by its data-type, so that it would be enough for the *Generator* to pick values exclusively from this subset when instantiating the parameter.

As a result, the *Generator* tries in a first step to find constraints on the domain of validity of the parameter which needs to be instantiated. Its strategy is still to rely on a random selection of the value, but only after the validity domain from which this value is taken has been reduced to its minimum, which is the intersection of all the domains of validity specified in the annotations for this parameter. In the following, the term critical value of a parameter is used to qualify any value this parameter could have in an execution path leading to the function call it is involved in (all the *reachable* values for this parameter).

The goal of the *Generator* will be therefore to determine with as much precision as possible the set of critical values for every parameter it must instantiate. For that, the sources of information previously pointed out will be used (the configuration files instantiated by the user and the *Cpal* specifications). Note that for the annotations, not all the *Cpal* specifications should be taken into account by the *Generator*: since parameter instanciation has to be completed before the execution of the target function (in its prestate), only preconditions can be used for the deduction of constraints on the function parameters.

All in all, the general strategy of the *Generator* to instantiate a parameter of a given C^{++} data-type can be summarized by the following steps:

1. Determine the data-type of the parameter under concern and initialize the set of critical values to the domain of validity for this data-type according to the C^{++} definition.
2. Check if the user specified restrictions for this parameter in the configuration files. If it is the case, restrict the set of critical values accordingly.
3. If the parameter belongs to a numeric built-in type, check if preconditions of the target function involve range constraints for this parameter. If yes, restrict the set of critical values accordingly.
4. If the parameter is a class instance, apply the strategy for instance generation (described below).

After the parameters of a function call have been instantiated, and before this combination of parameters is accepted as valid candidate for the test case instanciation, *Tennessee* checks whether the calls for the target function instantiated with these parameters fulfill all the preconditions for this target function. If this last test is successful, the combination of input values is used to instantiate the parameters of the function call in the test case.

4.5.3.2. User-defined parameterization

1. Generation of input values of a C^{++} built-in type

   If the user wants the *Generator* to integrate a specific constraint when computing the set of critical values for a parameter of a C^{++} built-in type occuring in the test-sequence, he has the possibility to specify this restriction in a configuration file. There are several, more or less direct ways to define this constraint, but the preferred one is to define a *Tennessee-container*. This extended notion of container corresponds to any set of data of a given C^{++} data-type that is defined in a configuration file. A *Tennessee-container* for the built-in data-type `int` can for instance be defined as a list of values explicitely written in the configuration file, as an interval (defined by its two extremal values), as a *container* defined in the C^{++} standard library, as the return value of a function call....

2. Generation of input values of a C^{++} complex type

   Tennessee considers as complex data-type any type that does not belong to the numeric built-in types predefined in the C^{++} programming language. When the parameter under concern is such an instance of a class, the user can also define a *Tennessee-container* to restrict its domain of validity, but its syntax may be a bit different. The valid instances (the critical values) which may be used can be defined directly or indirectly. In the first case, their name can be given if they already exist (in which case the user must ensure that they indeed belong to the visible scope of the target function in the source program) or they can be constructed explicitely in a piece of C^{++} code inserted by *Tennessee* in the generated test case. In the second case (indirect specification), the set of instances can be defined by indicating to the *Generator* the signature of a constructor which should be used for the construction of critical values along with, if necessary, additional information (for instance in the form of previously defined *Tennessee-container*s) about the way its parameters should be instantiated. It is also possible to require *Tennessee* to copy an existing instance (for which *Tennessee* calls the copy constructor).

Note that many of the parameters involved in these strategies may also be left unspecified by the user, so that *Tennessee* fills them with their default values. That can add a profitable degree of randomness to the computation of input values while still guaranteing that the generated values comply with the specification of the user. Finally, when the *Tennessee-container* corresponding to the critical values for the parameter under concern has been defined, additional variables in the configuration file allow the user to link this *Tennessee-container* to the parameter in the test-sequence it should be applied on.

4.5.3.3. Predefined generation of input values

Because *Tennessee* wants to relieve the programmer from any task other than the annotation of the source module, the default strategies are optimized in order to take into account the additional knowledge that can be deduced from the *Cpal* specifications.

1. Generation of input values of a C^{++} built-in type

   Basically, the default strategy for the generation of input values of a built-in type is random instanciation. However, if the parameter under concern belongs to a numeric built-in type,

Tennessee tries to find additional information in the *Cpal* specifications in order to restrict the range of critical values for this parameter. This is possible whenever the parameter is involved in *range constraints* in the preconditions contained in the annotations for the target function. In this case, the *Generator* analyses the range constraints and restricts the set of critical values accordingly.

2. Generation of input values of a C^{++} complex type

Generating a class instance automatically actually means calling an appropriate constructor for its construction. Performing this call means, for the *Generator*, to first select one of the possible constructor signatures that may be implemented in the module, and then to instantiate its parameters if necessary. The Figure 4.8 on page 49 gives an overview of the strategy used for input value generation that we describe in the following.

Stage 1: Selecting the appropriate constructor As usual, *Tennessee* first checks in the configuration file if the user specified a strategy that could be used to construct an instance of this class. If nothing is found, *Tennessee* checks if the parsor found such a constructor in the first stage of *Tennessee* computation, i.e., if a constructor for this class belongs to the current test-scope. If yes, it is used for the construction of the instance. If several constructors are available, *Tennessee* selects randomly one of them before every new instance generation. If no contructor is available, *Tennessee* uses the default constructor. This however only works if no other constructor is defined in the implementation, or if the default constructor has been overwritten in the source files. It is left to the programmer to guarantee that one of these conditions hold.

Stage 2: Generating, if necessary, input values for the selected constructor After learning the signature of the selected constructor, *Tennessee* can start generating input values for its parameters. If the constructor does not belong to the current *scope*, no constraint can be found for these parameters which are consequently instantiated with random values. Otherwise, the default strategy for input value generation can be reused here, at least for non-complex data-types. When complex data-types are involved, generating them requires to start again as subprocedure the current process (finding a correct constructor,...) so that a recursive computation is entered that may repeat itself indefinitely. In order to avoid running out of resource when such a situation occurs, the variable `gen_depth` is defined in the configuration file to specify the maximal number of nested computations of complex input values allowed by *Tennessee*. If this depth is reached and a new computation of a complex input value is still required, its construction is performed using the default constructor of the corresponding class, which does never accept parameters and therefore leads to the end of the recursion. Note however that this requires the call to the default constructor to be legal, which is assumed by *Tennessee* and must be ensured by the programmer.

Moreover, if several instances from the same class have to be generated at different depth levels of the same recursion path, *Tennessee* ensures that the same instance is not taken more than once, since this would lead to a loop in the recursion path (hence a non-terminating computation).

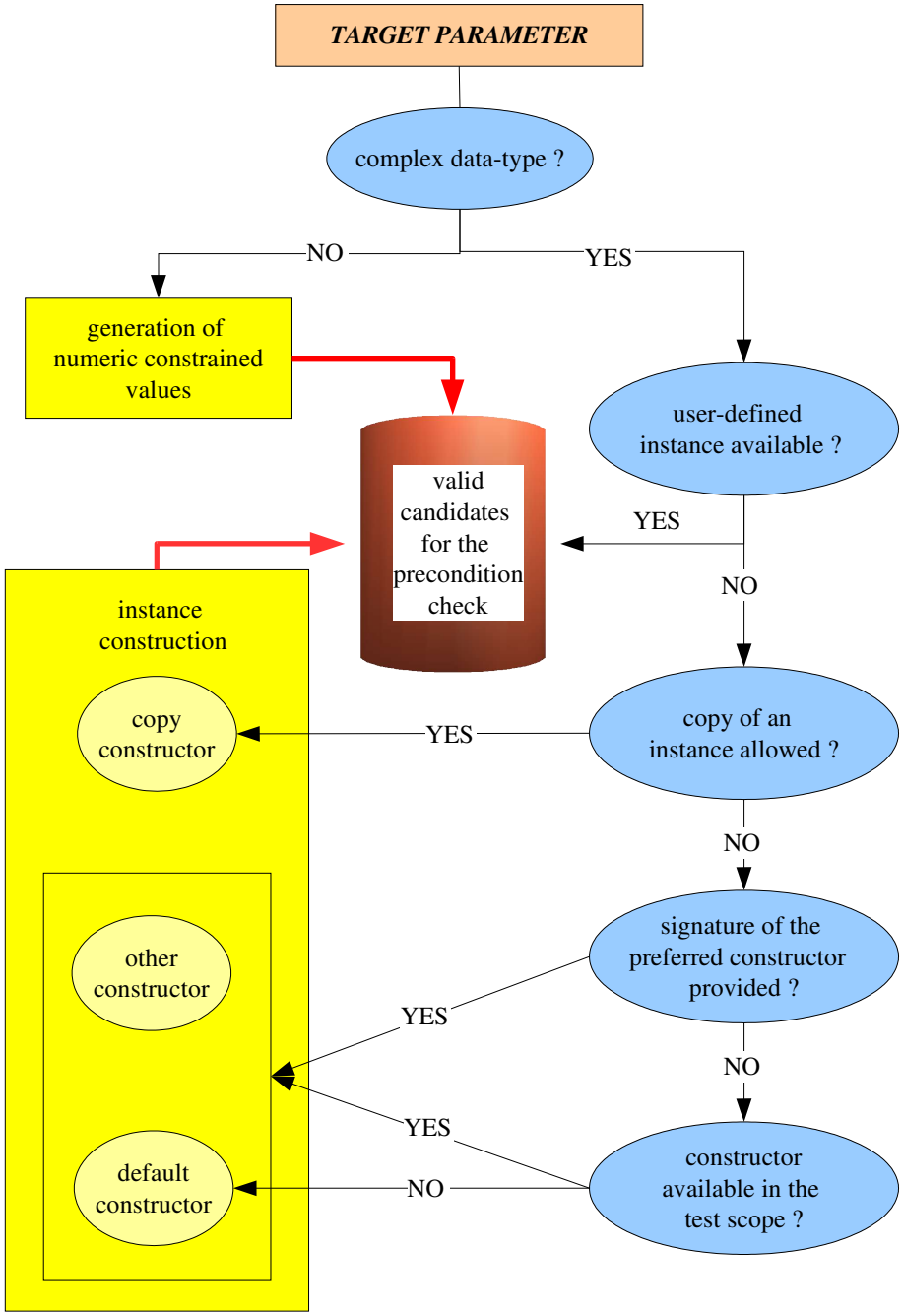

Figure 4.8.: Computation of valid candidates for the precondition check

4.5.4. Generation of base instances

After the test-sequence has been determined and its parameters instantiated, the last task left to the *Generator* it the determination of the instances on which the function calls in the test-sequence have to be applied. Selecting and/or creating these instances is done in a way similar to the definition of parameters of a complex type which we discussed in the paragraph for the instanciation of parameters. The user can define a specific base instance to use for every function call or for several consecutive function calls of the test-sequence. When nothing was specified in the configuration files, *Tennessee* constructs base instances using the same strategy as the one described previously (in the Stage 2). The only difference is an additional check performed by *Tennessee* to control that an object generated in a previous step as complex input value for a function call is not reused as base instance for this same function call. This is achieved by accepting as base instance only defined objects that have never been used, or newly generated ones.

5. Implementation

5.1. The new program environment

5.1.1. Approach

The new program environment computed by the *Traductor* contains for each target function a set of helper functions that implement the assertion checking code translated from the annotations. Most of these helper functions do not exist in the original program and are newly created by *Tennessee*. To guarantee that the implementation of these functions is correctly included in the new program environment, all the helper functions corresponding to the same target function are defined in a new header file. If the target function is a class method, its helper functions are defined as methods of the same class and their declarations are added to the class definition during the translation step.

Once these new functions have been correctly inserted in the translated program environment, it still remains to integrate calls to them in the execution flow so that the checks are performed at run-time when the corresponding state of computation of the target function (prestate or poststate) is reached. This is achieved by replacing in the new program environment the original target function by a wrapper function. A wrapper function corresponds to a function whose body has been modified but without changing its signature. This strategy offers a convenient way to conserve the initial execution flow of the program (thus simulating a realistic situation), while integrating checks that are executed at run-time when the wrapper function is invisibly called in place of the initial target function.

5.1.2. Assertion checks

The checks for assertion validity embedded in the helper functions of the new program environment are executed at run-time and an exception is thrown whenever an assertion failure is detected. The exception contains an error message which records as much information as possible about the failure that occured. This exception is then caught by the wrapper function and the subsequent behaviour of the program is configurable by the user. If the default strategy is not modified, the program aborts at the first assertion failure and logs the error message.

5.1.3. Generated helper functions

5.1.3.1. Introduction: the target function $clname :: fcname$

In the following, we assume that the general pattern of the signature of the target function, defined in the source file `mainfile.cpp` is as follows:

$$T\ clname :: fcname(T_1\ p_1, \ldots, T_n\ p_n)$$

where T is the data-type of the return value of the target function and $T_1, \ldots, T_n$ are the data-types of its parameters.

For every target function such as $clname :: fcname$, a definite set of helper functions is systematically computed by *Tennessee*:

- The **renamed original function** in `UTIL_clname_fcname.h`
- A **precondition checking function** in `UTIL_clname_fcname.h`
- A **postcondition checking function** in `UTIL_clname_fcname.h`
- A **class invariant checking function** in `UTIL_clname.h` (if still not existing)
- The **wrapper function** replacing the original target function in the new version of `mainfile.cpp`

We describe these helper functions in the following, along with an example, the target function `Node::append_node` belonging to the C^{++} class `Node` which implements a linked list and contains two private data fields:

- `int val;`
- `Node* next;`

Example: the target function `Node::append_node`

```
/*$++
  pre:{${ n!=NULL }$};
  post:{$${ OLD{length()}==length()-1 }$};
  pre:{$${ FORALL(int x;;Into_vec(x,all_vals());; x!=n->val }$};
++$*/
int Node::append_node(Node* n)
{
    if(next==NULL)
    {
        next = n;
        return 1;
    }

    else
    {
        Node* tmp = next;
        while(tmp->next!=NULL)
        {
            tmp = tmp->next;
        }
        tmp->next = n;
        return 1;
    }
}
/*$----$*/
```

5.1.3.2. The renamed original function

The original function is copied in the new program environment but its name is modified, so that the new signature has the following pattern:

$$T\ clname :: ORIG\_clname\_fcname(T_1\ p_1, \ldots, T_n\ p_n)$$

Example: signature of the renamed original function `Node::ORIG_Node_append_node`

```
int Node::ORIG_Node_append_node(Node* n)
```

5.1.3.3. The precondition checking function

$$void\ clname :: PRE\_clname\_fcname(T'_1\ p'_1, \ldots, T'_m\ p'_m)$$

The execution of the helper function PRE_clname_fcname aims at checking whether the preconditions specified in the annotations are satisfied when the target function is executed. Since the name of the formal parameters of the target function may appear in the annotations, the input parameters of the precondition checking function are the same as the target function. Every well-formed *Cpal* condition given in the annotations of the source file is translated into a well-formed equivalent C^{++} condition. Then, all these conditions are conjuncted in order to be checked together during execution of the wrapper function corresponding to the target function. In the wrapper function, a string literal is defined for every condition and contains the message that will appear in the log file whenever the corresponding condition is not satisfied at the time the tests are executed.

Example: precondition checking function `Node::PRE_Node_append_node`

The annotations for the function `Node::append_node` contain two preconditions:

```
pre:{${ n!=NULL }$};
pre:{${ FORALL(int x;; Into_vec(x,all_vals());; x!=n->val }$};
```

The first precondition requires the parameter of the method to be a valid pointer. It is a *boolean_condition* according to the *Cpal* grammar (4.2.2.2) and is directly translated to a legal C^{++} Boolean condition. The second precondition evaluates to true if and only if the value held by the node the parameter refers to is not already the value of a node present in the current list. The presence of the quantifier `Forall` triggers the creation of the file `UTIL_FORALL_Node_-append_node.h` in the new program environment, in which the function translating the check for the quantified precondition (and returning a Boolean value holding the result of this check) is defined. The corresponding C^{++} condition added in the wrapper function is reduced to a call to this function.

```
void Node::PRE_Node_append_node(Node * n)
{
    vector<char> vchar;
    vector<string> vstr;

    vchar.push_back(n!=NULL);
    vstr.push_back("n!=NULL");
```

```
    vchar.push_back(Node::fcpp_forall_0(n));
    vstr.push_back("FORALL(int x;; Into_vec(x,all_vals());; x!=n->val");

    bool res = true;
    int i=0;
    for(vector<char>::iterator it = vchar.begin();it!=vchar.end();++it)
    {
        res = (*it) && res;
        if(res==false)
            {
                const char* cond_str = vstr[i].c_str();
                throw pre_err("Node::append_node",cond_str);
                res = true;
            }
        i++;
    }
    return;
}
```

5.1.3.4. The postcondition checking function

$$void\ clname :: POST\_clname\_fcname(T\ ret, T'_1\ p'_1, \ldots, T'_m\ p'_m)$$

This function works in a similar manner as the precondition checking function but the number and type of its input parameters depend on the signature of $clname :: fcname$ and on the annotations. If the target function returns a value of type T, the first parameter of $clname :: POST\_clname\_fcname$ is instanciated at run-time with this return value, so that it can be used in the annotations (through the keyword `RET`). Additionally, the annotations for postconditions may also contain old values which refer to values in the prestate. These are therefore also passed as input parameters to the postcondition checking function. The original input parameters of the target function is the minimal set of old values stored by *Tennessee*.

The postcondition checking function `Node::POST_Node_append_node`

One postcondition is specified for the method `Node::append_node`, which checks whether the length of the list was indeed incremented by one during the `append_node` operation:

```
post:{${ OLD{length()}==length()-1 }$};
```

On the code given above, the parameters of this helper function consist of the return value `RET` of the original target function, and of two *old_value*s: the first one corresponds to the prestate value of the parameter of the original function (automatically stored by *Tennessee*) and the second one to an *old_value* specified by the user in the annotations.

```
void Node::POST_Node_append_node(int RET,
        Node* old_param_Node_append_node_0_1, int old_f_length$$0)
{
    vector<char> vchar;
```

```
    vector<string> vstr;

    vchar.push_back(old_f_length$$0==length()-1);
    vstr.push_back("old_f_length$$0==length()-1");

    bool res = true;
    int i=0;
    for(vector<char>::iterator it = vchar.begin();it!=vchar.end();++it)
    {
        res = (*it) && res;
        if(res==false)
        {
            const char* cond_str = vstr[i].c_str();
            throw post_err("Node::append_node",cond_str);
            res = true;
        }
        i++;
    }
    return;
}
```

5.1.3.5. The class invariant checking function

$$void\ clname :: CLASS\_clname\_checkinv()$$

If the target function is a class member, the specifications that are defined at the scope of the whole class are tested before the checks for the preconditions and postconditions of the target function are executed.

Example: the class invariant checking function `Node::CLASS_Node_checkinv`
No class invariant is specified for the class `Node`, so the generated class invariant checking function `Node::CLASS_Node_checkinv` is empty.

5.1.3.6. The wrapper function replacing the original target function

$$T\ clname :: clname\_fcname(T_1\ p_1, \ldots, T_n\ p_n)$$

The wrapper function is executed instead of the initial source target function whenever this latter is called. First, the class invariants and function preconditions are checked (i.e., the corresponding helper functions ar called). If these prestate checks succeed, the expressions specified as old values are stored in new variables before the initial source function (renamed in $clname :: ORIG\_clname\_fcname$) is called, and its return value (if any) stored in a new variable. After this call, execution has reached the poststate and the remaining checks for validity of postconditions and invariants are executed.

Whenever one of these checks fails, an exception is thrown by the corresponding helper function. In order to describe as precisely as possible the detected error and from which helper function it was thrown, different categories of exceptions are defined. Additionally, every exception comes with an error message that gives further hints about the problem, and which is copied at run-time

in the log file by the exception handler. If the user specified that the program should not be aborted after an assertion error, the error messages are listed sequentially in the log file.

If the user specifies it, *Tennessee* can also keep trace of the generated successful function calls performed so far during a test case.

Example: the wrapper function `Node::append_node`

Here, the default behaviour is adopted in which the computation is stopped whenever an exception for an assertion failure is caught by *Tennessee*.

```
int Node::append_node(Node* n)
{
    int RET;
    FILE* inputf;
    try
    {
        /**** check class invariant ****/
        CLASS_Node_checkinv();

        /**** check preconditions ****/
        PRE_Node_append_node(n);

        /**** store old values ****/
        Node* old_param_Node_append_node_0_1 = n;
        int old_f_length$$0 = length();

        /**** execute original function and store return value ****/
        RET = ORIG_Node_append_node(n);

        /**** check postconditions ****/
        POST_Node_append_node(RET,
                    old_param_Node_append_node_0_1, old_f_length$$0);

        /**** check class invariant ****/
        CLASS_Node_checkinv();
    }

    /**** exception handler for class invariant exceptions ****/
    catch(inv_err e)
    {
        const char *fct = (e.fct).c_str();
        const char *cond = (e.cond).c_str();
        abort();
        inputf = fopen("LOG_NOTOK_Node_append_node.txt","a");
        fprintf(inputf,"Node::append_node(n):
                        Error INV [Node::append_node]:
                              \%s doesn't hold !\n",cond);
        fclose(inputf);
    }
```

```
    /**** exception handler for precondition exceptions ****/
    catch(pre_err e)
    {
        const char *fct = (e.fct).c_str();
        const char *cond = (e.cond).c_str();
        abort();
        inputf = fopen("LOG_NOTOK_Node_append_node.txt","a");
        fprintf(inputf,"Node::append_node(n):
                        Error PRE [Node::append_node] :
                              \%s doesn't hold !\n",cond);
        fclose(inputf);
    }

    /**** exception handler for postcondition exceptions ****/
    catch(post_err e)
    {
        const char *fct = (e.fct).c_str();
        const char *cond = (e.cond).c_str();
        abort();
        inputf = fopen("LOG_NOTOK_Node_append_node.txt","a");
        fprintf(inputf,"Node::append_node(n):
                        Error POST [Node::append_node]:
                              \%s doesn't hold !\n",cond);
        fclose(inputf);
    }

    return RET;
}
```

5.2. The predefined test-case generation

Once the *Traductor* has generated the new program environment, the *Generator* can compute a new main function that will be inserted in the new program environment in place of the initial one. This main function implements a test case for the annotated C^{++} module which was given as input to *Tennessee*.

In the remaining paragraphs, we continue our example from the previous section and assume that the following test-sequence was specified in the configuration file as a sequence composed of the three following function calls repeated three times consecutively:

```
Node::length
Node::append_node
Node::length
```

where, as previously, the signatures of the target functions are as follows:

```
int Node::length()
```

`int Node::append_node(Node*)`

Further assumptions for this example are:

- The *scope* contains, besides the two target functions, the constructor `Node::Node(int)`
- A set of critical values for the parameter of the constructor `Node::Node(int)` was deduced from its *Cpal* annotations. It corresponds to the interval $[0;$`INT_MAX`$]$ of values of type `int`
- All the other parameters in the configuration files have been left unspecified and the values predefined in *Tennessee* must therefore be used for them.

The generation of a test-case is then achieved by performing following steps:

1. **Computation of input values for the parameter of** `Node::append_node(Node*)`

   The target function `append_node` is called three times in the test-sequence. Therefore, three different instances of the class `Node` have to be computed and the situation corresponds to the case explained in Paragraph 2 in which a constructor for this class belongs to the test-scope. Because a set of critical values is available for the parameter of this constructor, only values contained in it will be used for the three instance creations. Then, these three instances are pushed in a vector that will be used later on for the iteration. The C^{++} code generated by the *Generator* for this step is given in the following:

   ```
   /***** VECTOR for the parameters of $$c$$Node$$f$$append$$ *****/
   Node inst_Node_append_0(3546);
   Node inst_Node_append_1(76998);
   Node inst_Node_append_2(55);

   vector<Node*> vecgen_Node_append_0;
   vecgen_Node_append_0.push_back(&inst_Node_append_0);
   vecgen_Node_append_0.push_back(&inst_Node_append_1);
   vecgen_Node_append_0.push_back(&inst_Node_append_2);
   ```

2. **Generation of the base instance** The same strategy for the instance generation will be reused by *Tennessee* for the creation of the base instance on which the function calls will be applied. In the default strategy of *Tennessee*, the same base instance is indeed used for all the function calls in the test-sequence. This adds the code shown below to the main function for the test-case. The reason why a vector of `Node` pointers is computed is to handle the situation in which a different instance is required for each new sequence.

   ```
   /***** VECTOR for the instances of $$c$$Node$$ *****/
   Node inst_Node_0(857);

   vector<Node> vecgen_Node_0;
   vecgen_Node_0.push_back(inst_Node_0);
   vecgen_Node_0.push_back(inst_Node_0);
   vecgen_Node_0.push_back(inst_Node_0);
   ```

3. **Emission of the function calls** A loop is generated that calls the sequence of functions three times where a different parameter is used in every iteration for the parameter of `Node::append(Node*)`:

```
/***** Function calls *****/
for (int i=0; i<3; i++)
{
vecgen_Node_0[i].Node::length();
vecgen_Node_0[i].Node::append(vecgen_Node_append_0[i]);
vecgen_Node_0[i].Node::length();
}
```

We give in Annex A the source-code for the whole main function corresponding to this test-case. It can be found in Annex A. When they get embedded in the translated program environment previously generated by the *Traductor*, the function calls executed in this main function trigger the checks for assertion validity contained in the included helper functions so that at the end, the expected test-case generation performed by *Tennessee* is indeed successfully completed.

6. Evaluation

6.1. The module *Node*

6.1.1. Case study

The C^{++} module *Node* was chosen as first case study for the evaluation of *Tennessee*. The specification for this module, which implements a linked list (the class `Node`), is given in form of *Cpal* annotations in the source files. *Tennessee* was applied to this module in order to generate 100 test cases involving various sequences of typical list operations. Executing these test cases lead only for a few of them to a transparent evaluation of the underlying test-sequence. The faulty situations pointed out by the others permitted to discover a bug in the implementation of the method `Node::append` and a flaw in the specification for this method. These results are described with more details in the following paragraphs after a brief description of the module *Node* and the target functions selected for this experiment.

6.1.2. The class `Node`

An object of the class `Node` represents an element of a linked list. Every `Node` instance owns three private attributes:

- `name`: a `string` value representing the name of the instance
- `val`: a numeric value of type `int` corresponding to the weight of the instance
- `next`: a pointer of type `Node*` to the next element in the list

The interface of the class `Node` provides several public methods. We will focus on two typical operations involving list elements: the method `Node::append` and the method `Node::retrieve`, described in the following table:

Signature	Functionality	Return value
`int Node::append(Node* n)`	Add the element with pointer `n` at the end of the list whose head is `this`	`0`: success `1`: failure
`Node* Node::retrieve(int x)`	Remove the first element in the list whose weight is equal to `x`	A pointer to the removed instance (or 0)

The specifications for the two target functions we focus on are given in form of *Cpal* annotations and mention the two following methods which also belong to the class `Node` (but not to the test-scope of our test-harness) :

Signature	Functionality
`int Node::length()`	Returns the length of the list.
`vector<int> all_vals()`	Returns a vector containing the values of all the node instances contained in the list.
`vector<Node*> all_nodes()`	Returns a vector containing pointers to all the node instances contained in the list.

The annotations for the methods `append` and `retrieve` are as follows:

The method `Node::append`

```
/*$++
inv:{${ sum_vals()>=0 }$};
pre:{${ n!=NULL }$};
pre:{${ !Contained(n,all_nodes()) }$};
post:{${ OLD{{int}length()}<length() }$};
++$*/
int Node::append(Node* n)
{...}
/*$----$*/
```

The first specification is a class invariant precising that in the prestate and poststate of the method, the sum of the values of all the elements in the linked list whose head is the current base instance remains positive. Then, a precondition states that only an existent `Node` instance should be appened to the linked list, and that this element should not be already contained in the list. Finally, a postcondition checks that the length of the list was strictly increased by the append operation.

The method `Node::retrieve`

The method `Node::retrieve` removes the first element in the list whose weight is the value given as parameter. A pointer to the removed instance is returned by the method, or `NULL` if no element holding this weight was found in the list. The specification for this method consists of the class invariant already described previously, one precondition and two postconditions:

```
/*$++
inv:{${ sum_vals()>=0 }$};
pre:{${ Contained(newval,all_vals()) }$};
post:{${ !Contained(RET,all_nodes()) }$};
post:{${ OLD{length()}>length() }$};
 ++$*/
Node* Node::retrieve(int newval)
{...}
/*$----$*/
```

The precondition checks whether the weight given as parameter is indeed held by an element present in the list. The first postcondition resolves to true if the element the returned value points to is not contained in the list anymore after the execution of the function (so the retrieval was successful). Note that even if the attribute `next` of the last element in the list is equal to `0`, the implementation of the method `all_nodes` does not add the null pointer in the returned vector, so that the postcondition `post:{${ !Contained(RET,all_nodes()) }$};` is only violated in case an existent node instance holding the value `newval` was already present in the list in the prestate and is still there in the poststate although the operation `retrieve` should have removed it. Finally, the last postcondition checks that the length of the list was strictly decreased during the retrieval operation.

6.1.3. The test cases

We checked with *Tennessee* the compliance of the methods `append` and `retrieve` with their specification. The annotated source code was given as input to *Tennessee* along with a configuration file in which a few additional test parameters were set. The variable `ABORT` was for instance set to 1, so that any detected error triggered abortion of the execution of the program.

The generated test cases consisted in sequences of calls to the two methods, where the sequences had various length and composition. The values given as parameters and the objects used as base instances were also instanciated in several different ways to cover a lot of different situations. However, the critical cases which were detected mainly occured in sequences of operations involving only a few different `Node` instances. This comes from the fact that many specifications deal with relations between existing `Node` instances rather than independent properties of these objects.

The three possible outcomes that may result from the execution of a test-case in *Tennessee* (success, assertion violation or execution failure) appeared as the result of some test cases of this test-suite:

Result of the test case execution	Number of test cases
Success	26
Assertion failure (invariant violation)	72
Execution failure	2

6.1.4. First problem: an assertion failure detected by *Tennessee*

All the assertion failures detected concerned the same assertion, the class invariant `inv:{${ sum_vals()>=0 }$};` and revealed the same bug. An example of a failing sequence was:

```
Node* n00 = new Node(INT_MAX);
Node* n01 = new Node(67);
n00->append(n01);
```

Executing this program in the new program environment computed by the *Traductor* of *Tennessee* raises an assertion exception showing the message:

```
Caught INVARIANT POSTSTATE exception:
            Node::append([, 67]);
            sum_vals()>=0
```

which means that the class invariant `sum_vals()>=0` did not hold in the poststate of the method `Node::append` when a `Node` instance of weight `67` was given as input parameter.

The reason for this assertion failure is not obvious, since the weights of both elements in the list of head `n00` are positive and the result of `sum_vals()` should be the sum of all those weights. Displaying the result of the call to the method `sum_vals()` after `n01` was appended actually reveals that the sum resolves to `-2147483582`. Therefore, the problem comes from an integer overflow caused by the limited size of the numerical data-types in the C^{++} implementation. The fact that a large majority of test cases detected this error and that the failure occured in most of the cases after the execution of at most five function calls is due to the short execution path leading to occurence of this failure and to the default strategy of *Tennessee* which always, when possible, takes as first candidates for the instantiation of numeric parameters the values that are near to the extrema of their domain (here `INT_MAX`).

We performed a new series of 100 tests after an upper bound of 1.000.000 was added to the parameter of the constructor of `Node` and of the methods `append` and `retrieve` in form of a precondition (and after the source of execution failure had been removed as will be explained in the next paragraph). Because of the range constraint in the precondition, the weights of the generated instances were smaller, which prevented the occurence of this assertion failure. In this case indeed, no test case revealed again this failure.

6.1.5. Second problem: a flaw in the specification

The execution of two test cases let the computer hang, which was classified by *Tennessee* as an *execution failure*. In general, the fact that the program does not complete normally without generating an assertion exception reveals a flaw in the annotations or in their translation. That was confirmed in this case and we proved that both execution failures had the same origin.

One of the error-prone test case consisted in the following sequence of operations:

```
Node n00(0);
Node n01(1);
Node n02(2);
Node n03(3);
Node n04(4);
Node n05(5);

n03.append(&n04);
n04.append(&n05);

n00.append(&n05);
n00.append(&n03);

n00.display();
```

The way the `Node` instances are linked together generates a loop (see Figure 6.1.5). However the user, by explicitely adding the precondition `pre:{${ !Contained(n,all_nodes()) }$};` to the specification of the method `append`, actually intended to avoid this situation. The problem with this assertion is that it only takes into account the `Node` instance given as parameter when checking whether it is already contained in the list. All the possible other `Node` objects

previously appened to this parameter instance are however ignored and the appending operation leads to the formation of a loop whenever one of these objects is also in the current list.

This is exactly the situation which was revealed by this test case and which is not covered by the specification: appending `n05` and `n03` to `n00` while ignoring that `n05` was previously appened to `n03` generates a loop between `n03`, `n04` and `n05`.

In order to complete the specification so that *Tennessee* detects this situation when checking for assertion validity, we refined the incomplete precondition. The requirement is actually to allow the appending operation only when we are sure that the object given as parameter and all the other instances possibly linked to it are not already contained in the current list. This can be specified with the following *Cpal* assertion involving universal quantification, where `newnode` is the name of the formal parameter of the method `append`:

```
pre:{$${
    FORALL(
        Node* n;;
       Contained(n,newnode->all_nodes());;
       !Contained(n,all_nodes())
        )
    }$};
```

Informally, this resolves to true if every instance `n` contained in the list of head `newnode` is not contained in the list whose head is the current base instance (i.e., `this`).

The two error-prone test cases were executed again in the refined program environment and this time the faulty situations were indeed detected as precondition violations. For the first test case, following exception was thrown:

```
Caught PRECONDITION exception:
           Node::append([, 3]);
           FORALL(
               Node* n;;
               Contained(n,newnode->all_nodes());;
               !Contained(n,all_nodes())
               )
```

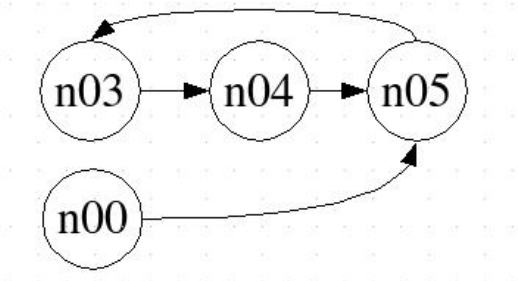

Figure 6.1.: Faulty situation for the method `append`

6.2. Comparison with Jartege

6.2.1. Jartege

Jartege is a tool for the random generation of unit tests for Java classes annotated in JML. For each new sequence of tests it performs, Jartege creates an object of the Java class `ClassTester` which is responsible for the automatic generation of the constructor and method calls. As soon as the Java classes that need to be tested have been given to the current `ClassTester` object, Jartege can start generating test cases in which objects from these classes are constructed and available methods applied on them.

Additionally, the user has the possibility to initialize the instance of `ClassTester` with parameters that control random aspects of the computation. *Weights* may for instance be attached to the tested methods and correspond to their probability to be called in test cases. Similarly, a probability function allows the user to specify the frequency of creation of a new instance for a new function call in the sequence (instead of reusing an existing instance).

Because they share similar goals, we decided to compare the performance of Jartege and *Tennessee* for different aspects of test case generation. The fact that the target programs of both tools are written in different programming languages (Java for Jartege, C^{++} for *Tennessee*) was not an issue, because both languages rely on the paradigm of *Object-Oriented programming*. Classes and methods that we tested could therefore be translated from one language to the other in a straightforward manner.

6.2.2. User-defined configuration of the test cases

In *Tennessee*, no testing class such as the `ClassTester` has been defined, but all the test parameters take the form of variables which can be directly instanciated by the user in configuration files and that are interpreted by *Tennessee* during its computation. We do not consider a strategy preferable to the other and both approaches seem to reach the same goals.

One important difference concerns however the way the user can configure the test cases. The facilities in *Tennessee* seem to be much more elaborate. In Jartege, if the few methods provided by the class `ClassTester` do not allow the user to configure the test cases according to his needs (e.g. if he wants to apply all the function calls to a unique, specific instance), then the only alternative which remains is to write these additional specifications in a class called `Fixture` and containing the methods `setUp` (resp. `tearDown`) that will be automatically called by Jartege at the beginning (resp. at the end) of every new test case.

On the contrary, the previous sections showed that practically all the different steps of the test case generation can be individually parameterized at different degrees of randomness in the configuration files provided by *Tennessee*. Even if, for instance, the two variables `initialize_string` and `finalize_string` can also hold code that is inserted at the beginning and end of every test case, it would be, for the aforementioned example, preferable in *Tennessee* to specify more directly (and more easily) which instances should be used for specific function calls in a test case and to parameterize them with more details if necessary.

We believe that this large variety of configuration facilities provided by *Tennessee* is a significant advantage of the tool. It reduces to the minimum the code the user must write in order to configure particular aspects of the generated test cases and gives him the opportunity to test precisely and without delay specific properties of the module under concern. And of course, the more general strategies such as fixtures are still supported if the programmer does not want to give a more detailed configuration. Moreover, integration of new strategies for the control of the parameter generation can also be achieved easily in *Tennessee*: it usually merely amounts to find the right

variables that need to be instanciated in the configuration files and to define a procedure that performs all these instanciations in a single step for later use of the same strategy.

The strategy of weight assignment provided by Jartege for the target functions has, for instance, not been directly implemented in *Tennessee* yet. However, the user has already different other possibilities to control the generation of the function call sequences computed by *Tennessee* for a test case, with a configurable randomness degree that ranges from *undeterminism* to *full determinism*. Thus, a probabilistic choice of the methods is all the same achievable in *Tennessee*, yet in a less direct manner than in Jartege (a new strategy for this facility could actually be integrated in *Tennessee* in the near future).

6.2.3. Optimization of the input values generation

6.2.3.1. Approaches supported by the tools

When generating input values for the parameters of a method whose behaviour has to be tested, it is important to avoid the values for which the function does not satisfy its preconditions (we will from now on call these values pre-invalid parameters). A straightforward way to achieve this goal is to select randomly from the set of all possible values some combinations of parameters, to check the precondition validity of the method instanciated with these values, and to keep for the test case only these combinations for which the method passed the test. This is precisely the strategy implemented by default in Jartege for any parameter of a Java primitive type. Unfortunately, the stronger the preconditions, the less probable it will be for the tool to catch pre-valid parameters when they are chosen from a large set of possible candidates such as the valid domains of the primitive types in Java.

The solution proposed by Jartege to reduce the overhead of the search for pre-valid parameters is to let the user provide these values: for any Java class to be tested (e.g. `Account`), a class called `JRT_Account` may be defined, which contains a private field of type `Account`, instanciated with the current object on which the target function is applied. If the user wants to instanciate a parameter of this method (e.g. the first parameter of the method `credit`, which is of type `int`), then he has to add to `JRT_Account` the method with signature `int JRT_credit_int_1()` and define it so that it returns a pre-valid value. Like all the other strategies presented in the previous sections that rely on the information provided by the user, this method may indeed greatly improve the relevancy of the generated test cases which are directly instanciated with critical values. This strategy has its counterpart in *Tennessee*, where sets of values can be generated in several different ways, including as return value of a user-defined function.

However, we consider this strategy as suboptimal, because the programmer has to write methods such as `int JRT_credit_int_1()` which, usually, only contains the transcription in executable code of the preconditions already given in form of annotations for the method under test. *Tennessee* already optimizes this selection with the strategy decribed in the Paragraph 4.5.3.1 on page 46, and that we call Pre-valid strategy.

6.2.3.2. Experiments

We checked the efficiency of the method supported by *Tennessee* by measuring the proportion in Jartege and in *Tennessee* of pre-valid values that were generated when 1000 candidates were selected by the test case generation tool. Because we focus on the computation of the test case generation tool in the prestate of the target function chosen for this experiment, the behaviour of this method does not matter. In order to test the generation of integer values we selected a method without side-effect and accepting a single value of type `int` as parameter. We modified

the standard behaviour of the tools, in order to count the number of generated pre-invalid values during the test case computation. Then, we asked the tools to compute a test case involving 1000 function calls to the selected target function.

This experiment was repeated for several different values of range constraints specified for the parameter of the target function in its annotations and we measured the proportion of pre-valid values that were generated for each experiment. More precisely, the ranges we specified were centered on 0 and covered a specific proportion of the whole range of Integer values supported by the system, from 0.5% to 100%. Three series of experiments were performed: two with *Tennessee* and one with Jartege. For *Tennessee*, we compared the results obtained with the use of the random strategy or the Pre-valid strategy (which is also the default strategy) for the input values generation. For Jartege, the default behaviour was tested.

6.2.3.3. Results

The results obtained for this three series of experiments are given in the Table 6.1 and illustrated in the Figure 6.2. The fact that, without regard to the range constraint, the Pre-valid strategy in *Tennessee* always lead to a proportion of 100% of pre-valid values seems to confirm the correctness of the computation of the set of critical values. As expected, the results for the random strategy are gradually worse when the range of permitted values gets smaller. In our experiment, less than half of the generated values were pre-valid parameters down to a range smaller than a quarter of the whole range of Integers (which typically still contains about half a billion of different possible values). The results we obtained for Jartege are better than the random strategy of *Tennessee*: the permitted range needs to shrink to 5% of the whole Integer range before less than half of the generated values are pre-valid, thus attesting the efficiency of the heuristic used by Jartege for the parameter instanciation.

As a conclusion for this experiment, we can point out the advantage of the Pre-valid strategy in *Tennessee*, which significantly improves the generation of relevant input values when enough information is found in the specification. For now, the annotation analysis is still in its infancy, but this could be developed in order to capture more constraints that could be used when generating input values. On the other hand, relying on the random strategy when no constraint can be deduced from the specification is not very satisfactory and show poor results. A good alternative would be to integrate the heuristic used by Jartege into *Tennessee* and use it instead of the Random strategy in order to reach the better results obtained by Jartege's default strategy for parameter instanciation.

6.2.4. Bug-finding

In a second experiment, we compared the relevancy of the test cases generated by Jartege and *Tennessee*. Our goal was to check whether both tools could find the same error(s), and how much time they needed before the failure was discovered. What we mean by time here is the length of the test case (i.e. the number of function calls) that are executed before the error occurs. Note that here we only count function calls that are present in the test case and that have consequently been instanciated with pre-valid parameters. Therefore, there should be no difference in the results between the different strategies used by the generation tools for the input values computation.

We selected as target functions the set {`credit`, `debit`, `setMin`, `cancel`} from the class `Account` and started with both tools the generation of test cases with a maximal length of 500

Coverage of the total range of Integers (in %)	Pre-valid parameters generated (for 1000 generated values)		
	Jartege	*Tennessee* (Random strategy)	*Tennessee* (Pre-valid strategy)
0.5	102	11	1000
1	196	19	1000
5	670	90	1000
10	882	189	1000
12.5	920	247	1000
25	999	443	1000
50	1000	756	1000
100	1000	1000	1000

Table 6.1.: Proportion of generated pre-valid values

Function call	`min`	`balance`
`Account a(10000,0)`	0	10000
`a.credit(5343893)`	0	5353893
`a.credit(2171330)`	0	7525223
`a.credit(2541720)`	0	10066943
`a.setMin(8805375)`	8805375	10066943
`a.credit(8678499)`	8805375	18745442
`a.cancel()`	8805375	10066943
`a.cancel()`	8805375	7525223

Table 6.2.: Sequence computed by *Tennessee* and revealing the invariant violation

function calls each, taken randomly from the set of target functions and applied on a single instance constructed in the same manner with both tools.

Executing the test cases revealed the occurence of an invariant violation for the class `Account`, that was detected by both tools. The three following steps show the minimal sequence that triggers this error (see the Figure 6.2 for a concrete example of a failing sequence generated by *Tennessee*):

1. A `credit` operation raises the `balance` of the current instance from b_1 to $b_2 > b_1$.
2. A `setMin` operation sets the `min` attribute to the value $b_3 > b_1$.
3. The operation `cancel` is applied on the instance, which gets as new `balance` the value b_1.

After this sequence, we are in a state where `balance`$= b_1 < b_3 =$`min`, which violates the class invariant $getMin() \leq getBalance()$.

This error was the first one discovered by both tools in many of the generated test cases.

We generated with both tools 100 test cases containing 500 function calls picked randomly from the set of target functions and counted the number of test cases which found the error. While Jartege detected the error 38 times, 77 test cases discovered it in *Tennessee*.

The difference between this results may partly result from the different strategies implemented by the tools for the parameter instanciation. We saw in the previous paragraph that *Tennessee*

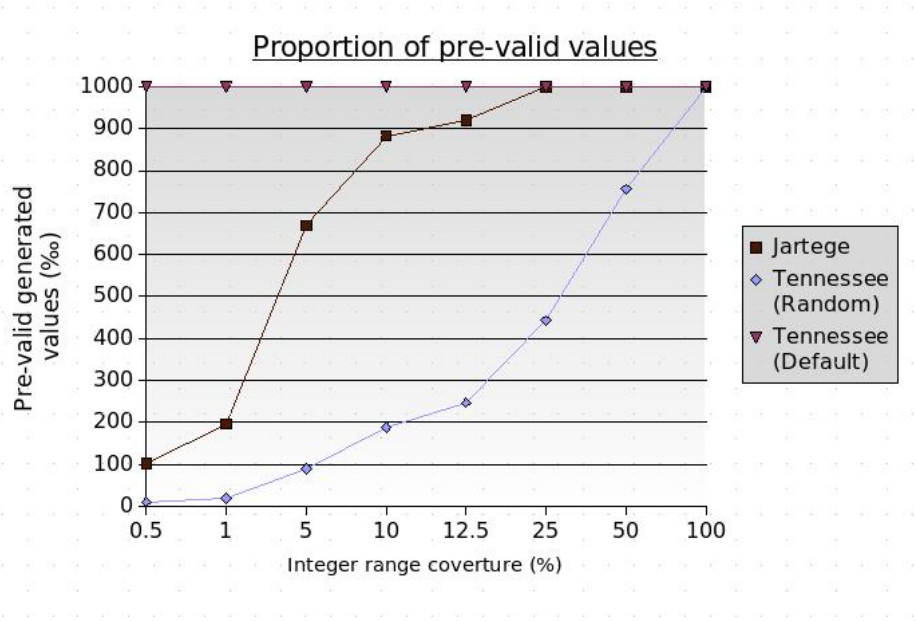

Figure 6.2.: Proportion of pre-valid generated values in Jartege and Tennessee (with the Random or Default strategies)

generates *directly* integers that satisfy the range constraints in the preconditions. In Jartege however, this filtration is performed at run-time and a limit exists for the maximal time allowed for the search of a given parameter satisfying the preconditions of a target function. A method with an integer parameter and a strong range constraint may therefore have a smaller probability to be called in the test case than a method with more permissive preconditions or with no parameter. The fact that in our example all the target functions except `cancel` are restricted by range constraints may imply that `cancel` is called with a higher probability than the other target functions, thus maybe preventing the sequence leading to the failure to occur as often as in the test cases generated by *Tennessee*. This assumption should however be confirmed by additional experiments such as the measurement of the probability for the function `cancel` to be called in a test cases generated by Jartege with comparison to the other target functions.

From these experiments, we can conclude that *Tennessee*, which applies on C^{++} modules and was developed *from scratch*, seems to provide more features and functionalities than Jartege. This, on one hand, lets the client configure with more precisions test cases adapted to his needs. On the other hand, the simplier approach provided by Jartege, which applies on Java programs and relies on the widely developed concepts of JML and JUnit, is easy to use and the limited facilities given to the user to adapt the test strategy proves quite efficient and may be sufficient for small applications.

7. Perspective and future work

Even if the two initial goals of *Tennessee*, the automation and optimization of the test case generation for C^{++} modules have been reached, its capabilities are still limited and a lot of work remains to be done to improve its efficiency.

The specification language *Cpal* supports only a limited syntax and expressivity. Quantification, for instance, can only be applied on variables, although it would be of advantage to support more advanced specifications that involve expressions of higher-order logics. As far as the syntax is concerned, the specification of logical connectives could also be extended, so that the traditional signs of *first-order logic* are allowed besides the C^{++} operators that have to be used for the moment (e.g. $\backslash/$ for conjunction instead of &&).

There is no limit on the ways the user could be allowed to configure a test case. First, the *strategies* currently supported by *Tennessee* and definable in the configuration files could be extended. Even if we consider the fine-grained instanciation allowed in configuration files as a clear advantage of *Tennessee*, we also noticed that, in the practice, setting up a strategy adapted to a specific module can become quite time-consuming. On the other hand, once such a strategy has been defined, it usually remains valid and useful in the long-run. It would be therefore of advantage to simplify the definition of procedures that set up in one step several overwritten parameters, so that a whole strategy can be immediately ready for the tests, and the switching from one strategy to the other simplified. Additionally, new approaches for the user-defined configuration, such as the definition of *weights* supported by Jartege (presented in Paragraph 6.2.2), could also be integrated in *Tennessee*.

The reporting and logging of assertion failures is another possible development. The current state only allows a coarse understanding of the failures, but more work on this aspect could help and accelerate the debugging process. Besides, advanced features could be added: with a large series of test cases, it would be for example interesting to deduce the minimal sequence of function calls leading to the occurence of a specific error. It seems however even more important for now to implement the distinction between internal and external assertion violations discussed in Paragraph 4.4.3, so that the responsibility for a failure is correctly assigned to the one who violated the contract.

Finally, the automatic filtration of input values should deserve attention in future development. The analysis of the *Cpal* annotations performed by the *Parsor* to deduce constraints on the parameters could certainly be improved. On one hand, other *black-box* approaches could optimize this step: a constraint solver or a model checker could for instance be applied on the specifications in a preliminary step to deduce more information about the valid domains of the parameters. On the other hand, it is not excluded to also integrate in the future mechanisms belonging to the field of *white-box* testing into *Tennessee*. Instrumenting the source code to find the privileged paths and derive relevant values for parameter instantiation has indeed already proved successful for the automatic generation of test cases [42] and we believe that ignoring *white-box* techniques would constitute sooner or later a non-negligible limit to the overall performance achievable by *Tennessee*.

8. Conclusion

In this thesis, we described *Tennessee*, a framework for unit testing and automatic test case generation for C^{++} modules. The framework was developed in order to achieve full automation of the test case generation process while improving to the extent possible the relevancy of these tests, so that they detect with high probability and in reduced time any faulty behaviour of the tested program.

The tasks that needed to be completed to adress this challenge were organized into three main steps performed by three different components (the *Parsor*, the *Traductor* and the *Generator*) relying on various features such as the definition of a new specification language called *Cpal*. The syntax of *Cpal*, designed to be intuitive for the C^{++} programmer, borrowed elements from this programming language while enhancing its expressivity by more advanced logical constructs (implication, quantifiers...).

The behavioral properties written in *Cpal* and analyzed by the *Parsor* are used to solve two key issues in unit testing: which value to choose as input data for the test cases and how to decide the validity of the test result (the test oracle problem). Filtering the valid input values according to the *Cpal* preconditions allows *Tennessee* to generate test cases that contain function calls simulating realistic situations with respect to the program specification. With this strategy, the computation of irrelevant test cases is significantly reduced. The test oracle problem is adressed by the *Traductor* which converts the abstract specifications into executable assertions checked at run-time when the function is called in the test case.

With this procedure, any annotated C^{++} module given as input to *Tennessee* triggers the generation of executable test cases that can be used to check its behaviour. *Tennessee* has however been further developed to become a framework which supports advanced interaction with the programmer. The motivation behind this approach is to allow the user to perform a fine-grained configuration of the generated test cases, which relies on additional knowledge about the module that *Tennessee* cannot directly derive from the specifications. Taking into account these user-defined *strategies* shall lead to the generation of test cases whose relevancy is significantly increased in comparison with the achievments of *Tennessee*'s standard computation. The drawback of this method is that it requires additional work from the programmer. However, the complete framework provided by *Tennessee* as well as the predefined configuration files that the programmer just needs to complete reduces this task to its minimum and justifies the interactive design adopted for *Tennessee*.

The experiments we performed to evaluate its performance demonstrated the capability of *Tennessee* to generate test cases that indeed implement the *Cpal* abstract specifications and whose execution can reveal assertion failures contained in the module. The comparison with a similar existing tool for the Java programming language also proved the benefit of the predefined strategy of *Tennessee* which can directly generate, in case of numeric parameters, input data simulating with more accuracy a realistic call to the target function.

Appendix A.

Example of a generated test case

The test-sequence for this example consists of three repetitions of the following three function calls:

```
Node::length
Node::append_node
Node::length
```

Source code of the generated test case:

```
int main(int argc, char* argv[])
{
    /***** VECTOR for the parameters of $$c$$Node$$f$$append$$ *****/
    Node inst_Node_append_0(3546);
    Node inst_Node_append_1(76998);
    Node inst_Node_append_2(55);

    vector<Node*> vecgen_Node_append_0;
    vecgen_Node_append_0.push_back(&inst_Node_append_0);
    vecgen_Node_append_0.push_back(&inst_Node_append_1);
    vecgen_Node_append_0.push_back(&inst_Node_append_2);

    /***** VECTOR for the instances of $$c$$Node$$ *****/
    Node inst_Node_0(857);

    vector<Node> vecgen_Node_0;
    vecgen_Node_0.push_back(inst_Node_0);
    vecgen_Node_0.push_back(inst_Node_0);
    vecgen_Node_0.push_back(inst_Node_0);

    /***** Function calls *****/
    for (int i=0; i<3; i++)
    {
        vecgen_Node_0[i].Node::length();
        vecgen_Node_0[i].Node::append(vecgen_Node_append_0[i]);
        vecgen_Node_0[i].Node::length();
    }

     return 0;
}
```

Bibliography

[1] B. Beizer. *Software Testing Techniques, 2nd ed.* 1990. 15

[2] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969. 17

[3] J. M. Spivey. *The Z Notation: A Reference Manual, 2nd ed.* International Series in Computer Science. Prentice Hall, NewYork, NY, 1992. 18

[4] J. Dawes. *The VDM-SL Reference Guide*. Pitman, 1991. 18

[5] Bertrand Meyer. *Eiffel: The Language*. Object-Oriented Series. Prentice Hall, NewYork, NY, 1992. 18

[6] Igor D. D. Curcio. Asap - a simple assertion preprocessor. *SIGPLAN Notices*, 33(12):44–51, 1998. 18

[7] Carolyn K. Duby, Scott Meyers, and Steven P. Reiss. CCEL: A metalanguage for C++. In *USENIX C++ Technical Conference Proceedings*, pages 99–115, Portland, OR, 10–13 1992. USENIX Assoc. Berkeley, CA, USA. 18

[8] Pedro Guerreiro. Another mediocre assertion mechanism for c++. In *TOOLS '00: Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 33)*, page 226, Washington, DC, USA, 2000. IEEE Computer Society. 18

[9] Mike A. Martin. Effective use of assertions in c++. *SIGPLAN Notices*, 31(11):28–32, 1996. 18

[10] D. Bartetzko, C. Fischer, M. Moller, and H. Wehrheim. Jass - java with assertions, 2001. 18

[11] M. Lackner, A. Krall, and F. Puntigam. Supporting design by contract in java, 2002. 18

[12] I. Nunes. Design by contract using meta-assertions, 2002. 18

[13] Reinhold Plosch. Design by contract for python. In *APSEC '97: Proceedings of the Fourth Asia-Pacific Software Engineering and International Computer Science Conference*, page 213, Washington, DC, USA, 1997. IEEE Computer Society. 18

[14] Reinhold Plosch and Josef Pichler. Contracts, from analysis to c++ implementation. In *Proceedings of TOOLS 30*, pages 248–257. IEEE Computer Society, 1999. 18

[15] Gary T. Leavens and Yoonsik Cheon. Preliminary design of Larch/C++. In U. Martin and J. Wing, editors, *Proceedings of the First International Workshop on Larch, July, 1992*, pages 159–184. Springer-Verlag, New York, N.Y., 1993. 18

[16] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report 98-06i, 2000. 18

[17] G. Leavens, Y. Cheon, C. Clifton, C. Ruby, and D. Cok. How the design of jml accommodates both runtime assertion checking and formal verification, 2003. 18

[18] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for Java. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI'2002)*, volume 37, pages 234–245, June 2002. 18

[19] Bertrand Meyer. *Object-oriented Software Construction, 2nd ed.* Prentice Hall, New York, NY, 1997. 19

[20] David S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995. 19

[21] J. F. H. Winkler and S. Kauer. Proving assertions is also useful. *SIGPLAN Not.*, 32(3):38–41, 1997. 20

[22] Y. Cheon and G. Leavens. A runtime assertion checker for the java modeling language. 2002. 21, 37

[23] Hans-Martin Hörcher. Improving software tests using Z specifications. In Jonathan P. Bowen and Michael G. Hinchey, editors, *ZUM'95: The Z Formal Specification Notation*, volume 967 of *Lecture Notes in Computer Science*, pages 152–166. Springer-Verlag, 1995. 23

[24] J. M. Spivey. *The Z notation: a reference manual*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1989. 23

[25] Jeff Offutt and Aynur Abdurazik. Generating tests from UML specifications. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723, pages 416–429. Springer, 1999. 23

[26] James Rumbaugh, Ivar Jacobson, and Grady Booch, editors. *The Unified Modeling Language reference manual*. Addison-Wesley Longman Ltd., Essex, UK, UK, 1999. 23

[27] Juei Chang, Debra J. Richardson, and Sriram Sankar. Structural specification-based testing with adl. In *ISSTA '96: Proceedings of the 1996 ACM SIGSOFT international symposium on Software testing and analysis*, pages 62–70, New York, NY, USA, 1996. ACM Press. 23

[28] Juei Chang and Debra J. Richardson. Structural specification-based testing: automated support and experimental evaluation. In *ESEC/FSE-7: Proceedings of the 7th European software engineering conference held jointly with the 7th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 285–302, London, UK, 1999. Springer-Verlag. 23

[29] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs, 2001. 23

[30] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *ESEC/FSE-9: Proceedings of the 8th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 62–73, New York, NY, USA, 2001. ACM Press. 23

[31] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002. 23

[32] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. Technical Report 01–12, 2001. 23

[33] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133, New York, NY, USA, 2002. ACM Press. 23

[34] CppUnit. Located at:
`http://cppunit.sourceforge.net/cppunit-wiki/FrontPage`. 24

[35] CppTest. Located at:
`http://cpptest.sourceforge.net`. 24

[36] Boost. Located at:
`http://www.boost.org/libs/test/doc/index.html`. 24

[37] J. Gil and B. Holstein. T++: A test case generator using a debugging information based technique for source code manipulation. In *TOOLS '97: Proceedings of the Tools-23: Technology of Object-Oriented Languages and Systems*, page 272, Washington, DC, USA, 1997. IEEE Computer Society. 24

[38] Melvin Fitting. *First-order logic and automated theorem proving*. Springer-Verlag New York, Inc., New York, NY, USA, 1990. 32

[39] Larch Family Languages. Located at:
`http://www.boost.org/libs/test/doc/index.html`. 32

[40] Patrice Chalin. Back to basics: Language support and semantics of basic infinite integer types in jml and larch. Technical Report CU-CS 2002-003.1, Computer Science Department, Concordia University, October 2002. 34

[41] Patrice Chalin. Logical foundations of program assertions: What do practitioners want? In *SEFM '05: Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, pages 383–393, Washington, DC, USA, 2005. IEEE Computer Society. 37

[42] Yoonsik Cheon, Myoung Kim, and Ashaveena Perumendla. A complete automation of unit testing for Java programs. In Hamid R. Arabnia and Hassan Reza, editors, *Proceedings of the 2005 International Conference on Software Engineering Research and Practice (SERP '05), Volume I, Las Vegas, Nevada, June 27-29, 2005*, pages 290–295. CSREA Press, 2005. 71

[43] Parasoft. Located at:
`http://www.parasoft.com`.

[44] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[45] David Gries and Fred B. Schneider. Avoiding the undefined by underspecification. In Jan van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, number 1000, pages 366–373. Springer-Verlag, New York, N.Y., 1995.

[46] D. L. Parnas. Predicate logic for software engineering. *IEEE Trans. Softw. Eng.*, 19(9):856–862, 1993.

[47] Bart Jacobs, Joachim van den Berg, Marieke Huisman, Martijn van Berkum, Ulrich Hensel, and Hendrick Tews. Reasoning about Java classes. In *Proceedings, Object-Oriented Programming Systems, Languages and Applications (OOPSLA'98)*, pages 329–340, Vancouver, Canada, 1998.