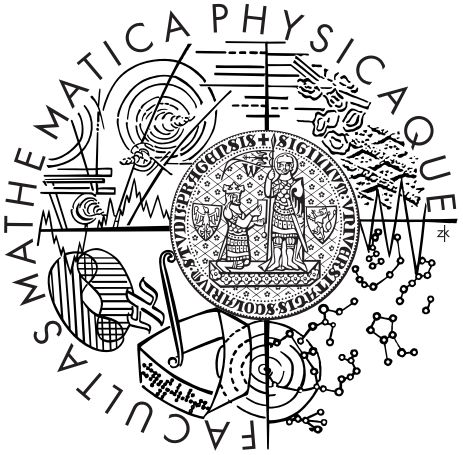
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Tomáš Trojánek

Capturing Performance Assumptions using Stochastic Performance Logic

Department of Distributed and Dependable Systems

Supervisor of the master thesis: doc. Ing. Tůma Petr, Dr.
Study programme: Computer science
Specialization: Software systems

Prague 2012

I owe many thanks to my supervisor doc. Ing. Tůma Petr, Dr. for numerous pieces of advice, corrections and the time he dedicated to this thesis. His guidance and interest helped me to move forward and develop a fine understanding of the subject.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In ..... date ..... signature of the author

Název práce: Záznam předpokladů o výkonu pomocí stochastické výkonnostní logiky

Autor: Tomáš Trojánek

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: doc. Ing. Tůma Petr, Dr.

Abstrakt: U softwarových projektů se výkonnostní testování používá k objektivnímu zhodnocení rychlosti daného díla. V ideálním případě je k projektu přidružena sada testů, která může být opakovaně spouštěna za účelem ověření, zda jsou veškeré výkonnostní nároky stále dodrženy. V dnešní době nejrozšířenější způsob, jak tyto testy provádět, staví na absolutních hodnotách naměřeného času. Test spustí vybranou jednotku aplikace a následně porovná dobu jejího běhu s předem stanovenou časovou hranicí. Tento přístup má však nevýhody, které značně nabourávají spolehlivost takovýchto testů. Není totiž zřejmé, jak určit ony časové hranice, které rozhodují o úspěchu nebo selhání. A pakliže jsou tyto hranice určeny, jsou závislé na konkrétní hardwarové konfiguraci. Tato práce se proto jako řešení zmiňovaných problémů pokouší ukázat odlišný způsob, který je založený na relativním porovnávání výkonu. Za pomoci logiky, která staví na práci publikované zadávající katedrou, se vybrané jednotky aplikace porovnávají navzájem a výsledky testů se tak stávají odolnější vůči změnám prostředí. Zavedená teorie je v práci také implementována a ověřena na vybraných ukázkových případech.

Klíčová slova: stochastická logika, výkonnostní testování, relativní předpoklady

Title: Capturing Performance Assumptions using Stochastic Performance Logic

Author: Tomáš Trojánek

Department: Department of Distributed and Dependable Systems

Supervisor: doc. Ing. Tůma Petr, Dr.

Abstract: Performance testing is a mean used to evaluate speed of software projects. In an ideal state a project has a set of tests attached to it and such set may be repeatedly executed in order to verify that all performance expectations are satisfied. The most widespread method of constructing these tests nowadays is based on measuring absolute time values. A test executes a chosen application unit and then compares the time it took to complete with a precise bound, which has been determined in advance. However, this approach has several disadvantages that affect reliability of such tests. First of all, the way in which those precise bounds should be established is not clear. And even if it is, then the bounds are tied to a certain hardware configuration. As a remedy, this thesis demonstrates a whole another approach, which is based on relative performance comparison. Using a logic built on top of a research published by the issuing department, chosen application units are compared together in a manner that makes results of such tests more reliable even to a change of hardware configuration. The presented theory is also implemented and verified on selected use cases.

Keywords: stochastic logic, performance testing, relative assumptions

Contents

Introduction

Motivation

Software testing in general is a recommended best practice if not a required attribute of a well-executed software project. It has to be planned in advance, designed in the same manner as the code itself and finally implemented according to a prescribed set of project guidelines. As a result, projects with sufficiently rich test support achieve higher quality in production [1] and their acceptance by clients is smoother, because requested functionalities are tested throughout the whole development. A thorough test suite gives project manager a comprehensible overview of the state of the project under development and has a potential to make the project cheaper by an order of magnitude, because important architectural flaws are detected earlier [2].

In general, software testing may be divided into several orthogonal areas of interest, two of which are relevant for this thesis – functional testing and performance testing. Both are employed to ensure that the resulting application meets its original specification and to increase its quality. Functional testing is aimed at verification of software's behavior. It is commonly understood by developers and utilized by the popular test driven development methodology (TDD, [3]). On the other hand, performance testing is used to either verify speed requirements of application units or to verify high-level attributes of application's architecture, e.g., the ability to handle load. The number of software project types that can benefit from this kind of testing is not limited – from small embedded systems to big internet applications. Yet, because of its difficult configuration and the amount of effort necessary for a good-quality integration into a project, performance testing is often neglected or skipped altogether.

This thesis is focused on performance testing that evaluates speed of application units – known under the simple name of *performance unit testing*. Similar to regular functional unit tests, performance unit tests usually follow the same structure:

1. **Arrangement** – Setup of an environment required by the system under test (SUT, [4]).
2. **Action** – Execution of the SUT. In context of performance testing this may mean measuring and collecting time data.
3. **Assertion** – Comparison of the yielded result with an expected value.

In order to successfully implement such test, all these three phases must be clearly specified in advance. Additionally, there are several very important requirements

for the test as a whole – it has to be stable, deterministic and repeatable. This means that if the test is launched several times with the same input data and in the same environment, its result should always be the same. A test whose result is changing nondeterministically is irritating during development and it is only matter of time before programmers start to ignore its failed assertions completely (as described in [5]). Hence the essential purpose of the test is voided.

In order to achieve the reliability described in the previous paragraph, it is first necessary to choose an approach that prescribes how performance assumptions should be captured and how the tests should be designed and evaluated. The capturing of assumptions means that each test should have a formal specification of what is fast and what is slow. The guidelines for design and evaluation tell how to correctly determine whether all relevant assumptions are met.

The classical and most widespread approach nowadays uses absolute time bounds to capture the assumptions. The design of such tests usually follows the arrangment-action-assertion structure with assertion part being possibly a single line of code that compares the measured execution time of the SUT to some given bound.

More complex variations of this scheme are possible. For example, SUT's execution may be measured repeatedly and only a chosen statistic (mean, median) is then compared to the bound. Similarly a certain level of tolerance may be introduced to protect the test from failing just because of a small deviation.

But even the best tailored scheme cannot compensate for important disadvantages of this approach. They all stem from the way the bounds are determined. With software applications, performance is tightly coupled to the hardware and to the environment they are being executed in. When setting up an absolute bound, developers have to bear in mind that the test will only be valid on a specific configuration. This may be a violation of software testing rules that were stated earlier, because for some computers the test will be always successful, for some it will be always unsuccessful and the most trouble-making scenario is that it will be sometimes successful and sometimes unsuccessful.

However, this problem is not relevant if the test is created according to a specification that clearly states what performance is required on what configuration. Other remedy from unreliable results is to design the test for a configuration of a server that provides continuous integration ([6]). Tests are then executed only on that machine and when the results are collected, they are sent back to the programmer who is the author of the last commit in a revision control system. But this is still only one solution out of many that attempt to artificially increase the usability of such testing.

A whole other approach is described and researched in this thesis. It is based on the article [7] that proposes usage of *relative time bounds* for capturing performance assumptions. Bounds expressed in this relative manner aim at strong hardware independence. The design process of a test starts with specification of a special logical formula that describes in which relation is performance of two or more application units expected to be. The next step is to measure the actual performance of the tested units in absolute time values. Measured data creates an input for statistical tests that evaluate the logical formula. This yields the result of the test. The concept of specifying bounds in a relative way simplifies the

general task of performance testing, because tests become only loosely coupled to some particular platform and may be deployed to multiple machines without affecting their results. It is also easier for programmers to find natural bounds for tested units. Any sorting algorithm, for example, may be compared to the Quick Sort or the Bubble Sort and thus a comprehensible overview of its speed is created. Even non-technical people like customers who ordered an application from a software company are able to say that the result should not be slower than some similar already available reference application.

Goals

This thesis follows up with research from the original article [7] and presents the following contributions:

- A new customized logic for performance formulas is introduced. This enables logical formulas that capture performance assumptions to be more modular and facilitate diverse requirements of real world software projects.
- Researched knowledge is verified by an implementation of evaluation engine library that enables performance unit testing in the .NET Framework environment. The design of the library is described together with a worked example.
- The library is applied to selected open source projects in order to demonstrate its benefits. The use cases are analyzed in detail and general best practices are argued as a conclusion.

Organization

The text is structured as follows.

- **Chapter 1** defines Customized Stochastic Performance Logic (CSPL) used later in the thesis to express performance assumptions. The logic is a customization of the logic defined in [7]. Several concepts unnecessary for the practical implementation are stripped off and new relation operators are introduced to also enable the classical absolute time bound approach.
- **Chapter 2** introduces two interpretations for the CSPL logic. The first one has only a theoretical purpose and is used to explain the logic. The other assigns semantics based on execution time sampling and statistical tests. This interpretation is designed so that it may be implemented and used by the thesis.
- **Chapter 3** is focused on UnitRacer – a practical implementation of performance unit testing library based on the theory from the first two chapters. The design of the library is discussed and demonstrated on examples. The chapter pays special attention to evaluation of hypotheses using statistical tests and describes how new tests can be added and thus extend the library.

- **Chapter** 4 presents several use cases that illustrate how UnitRacer can be used in software projects. The proposed usage starts with testing architectural components and shows benefits of such early testing. The text continues with discussion of regression testing and testing of regular units during the construction phase of development. Another topic on which the chapter is focused analyzes how performance of application-level scenarios can be put into a relation with industry standards in order to ensure that a quality result is delivered to stakeholders.

- **Conclusion** summarizes the thesis and argues known drawbacks of the implemented library. There is also a discussion of several areas suggested for further research.

Chapter 1

Customized Stochastic Performance Logic

1.1 Background

An important part of this thesis is a practical implementation of an engine library that enables relative performance unit testing of real-world projects. In order to deliver such library, it is first necessary to establish a solid theory based on which the implementation would be possible. Hence this chapter starts with the basics and introduces and precisely defines a mean of capturing performance assumptions.

A programmer may be in a situation where he needs to make a decision between some two similar approaches. Because performance is important to his project he decides to compare these two approaches together using a performance unit test and then choose the faster one.[1] This will not only provide him with a clear answer, it will also serve as an important form of documentation (*Test as Documentation*, [4, p. 23]) – as long as the test will be a part of the project, it will clearly show why the decision to favor one approach over some other has been made.

A starting point of each relative performance test is to capture assumptions about speed of the compared units. If the goal is simply to choose the faster unit, the assumption may read like:

> *"Unit A is faster than unit B."*

In other scenarios the assumptions are more complex. For example, a programmer may want to verify his implementation of the Merge Sort algorithm, which is known to be slower on average than the Quick Sort but faster than the Bubble Sort algorithm.[2] In such case he can state:

[1] The process of writing a performance unit test may be skipped if relative performance of the two approaches is already documented by some established authority (e.g. MSDN network for .NET Framework platform). At the time of writing this thesis, such official comparisons were, however, very rare to find.

[2] The statement is based on amortized computational complexities of the three algorithms. Values computed for their standard forms are stated in, e.g., [8].

> *"When sorting arrays of ten thousand integers, the mean performance of the Merge Sort algorithm is slower than the mean performance of the Quick Sort and faster than the mean performance of the Bubble Sort algorithm."*

It is apparent that the assumption from the first example is much more vague than the second one. Still even the second one may be interpreted in multiple ways. To this end, it is important to establish a precise theory that will enable programmers to state their assumptions in an exact way, with no room left for ambiguity. In particular, the problematic areas are:

- **Units** – The subject of a test should be clearly specified. In context of performance unit testing the units are understood to be class or instance methods. Such methods should be made available to the evaluation engine. This, among other things, means that the engine is not responsible for creating instances upon which the methods are executed. Consequently, the precise way of obtaining the instance is also part of the test specification.
- **Input data** – It is common that a unit will have different performance on different input data. When executed on a large data set the performance will be slower than with small data. In relative performance testing it may be therefore necessary to provide the compared units with the exactly same input data to ensure a fair result. When units are executed repeatedly this requirement may be in some situations made weaker by requiring only some input data generator that produces data from the same class (e.g. integer arrays of the same length and with the same amount of transpositions for sorting algorithms).
- **Measured attributes** – Speed is not a well-defined parameter. When comparing sorting algorithms, the Quick Sort has higher worst time performance than the Merge Sort ([8]). The mean performance is, however, in the opposite relation (ibid.). Telling which algorithm is better therefore depends on which attribute is more important in the given situation.
- **Evaluation** – In order to make it possible for anyone to reproduce such test, it is also necessary to define, with all the previous points set in place, a precise evaluation process. The process should then always be a part of the test specification.

In this thesis, the task of establishing a precise theory will be split into two parts. The first is in this chapter and focuses on introduction of a special logic, designated for making relative performance statements in form of logic formulas. The other part provides an interpretation for the logic so that the formulas may be evaluated. Both parts together are an answer to the problematic areas from the previous paragraph.

1.2 SPL Legacy

The logic presented in this chapter is tightly based on *Stochastic Performance Logic* (SPL) from the article [7]. Therefore, it is recommended for the reader to

get familiarized with the article first and then return back to this text, where a modification of the logic – *Customized Stochastic Performance Logic* (CSPL) – is described.

The main purpose of customizing SPL into CSPL is to prepare a theoretic background for implementation of an evaluation engine based on that logic. CSPL has been created from SPL by removing some parts that are not necessary for the engine. New parts have been added and some parts have been modified as well.

First of all, the concept of *workload parameter* [7, p. 2] has been left out from CSPL. In SPL workload parameter determines class of input data for methods under test. It is a mean that enables programmers to express their performance assumptions in a way that takes the nature of input into account. Because SPL has knowledge of input data classes, its formulas may be quantified over them, i.e., over workload parameter domains. For the evaluation engine from this thesis, quantification has not been identified as an essential component, because it may be accomplished by repeatedly executing a performance unit test that takes different input data generators as a parameter. An example of simulating workload parameter quantification in CSPL can be found at the end of section 1.3. Because workload parameters are left out from CSPL, *method workload* [7, p. 2] is left out as well.

The concept of *method performance* (ibid.) is present in CSPL but is redesigned so that it does no longer use workload parameters.

Perhaps the most important change is the addition of new relation operators. CSPL introduces the new operators in order to make the evaluation engine based on it more universal and support also the classical absolute time bounds based performance testing. Thanks to the following theory and its interpretation, the performance testing of a unit against a time bound will be more reliable and easier to express. This will be especially true when comparing the evaluation engine provided with this thesis to a framework that a programmer would otherwise manually write for his performance tests (as seen in projects that have been selected for the use case demonstration of the library in chapter 4).

1.3 The Logic

One of the most important tasks of a performance testing theory is to precisely define what a tested unit is. In previous sections it has been shown that not only the unit itself affects the measured performance, but also the way it is obtained and what input data it is executed with. When capturing performance assumptions it is necessary to specify all these things. For CSPL, concepts like unit, its state and input data could be defined. But since it is possible to leave all these specifics for a particular interpretation of the logic, only a single concept – *performance sampler* – will be introduced.

Definition. *Let* Ω *be a sample space, then* performance sampler *is defined as a random variable* $\Psi : \Omega \to \mathbb{R}^+$.

The definition is very general, but as it has been said, it will be up to CSPL interpretations to assign some meaning to it. For purposes of better clarification, performance sampler may be interpreted as follows.

Example 1. *Let* M *be a method,* G *a generator of random input arguments and* E *state of the environment that affects* M*'s performance. Then observations of performance sampler* $\Psi_{M,G,E}$ *are interpreted as execution times of method* M *in environment* E *with arguments provided by* G*.*

To describe performance sampler from another point of view, it is useful to compare it with SPL concepts. SPL started with *workload parameters* which are used to specify dimensions for random input arguments. A workload parameter with value 10000 could mean that random arrays of 10000 integers should be generated. The next concepts, *workload class* and *method workload*, used sample space Ω and workload parameters to compose a random variable whose observations are object instances that are fit as input arguments for the tested method. Finally, *method performance* is defined as a random variable whose observations are execution times of the tested method with random input arguments from *method workload*.

In CSPL, as illustrated by example 1, the task of creating random input arguments is completely hidden (in the example by a random generator G). And because there is no quantification over classes of input arguments, it is up to a particular interpretation of the logic to specify what will be encapsulated under the symbol Ψ. In the previous example it was a mechanism composed of three components – M, G and E, but anything that can be regarded as a stochastic process $\Omega \to \mathbb{R}^+$ is valid.

With performance sampler in place, it is possible to proceed to the definition of CSPL logic.

Definition. CSPL *is a many-sorted first-order logic with:*

- *Function symbols*
  - FunPS – *for* performance functors.
  - FunT – *for performance observation transformations with arity* $\mathbb{R}^+ \to \mathbb{R}$.
- *Predicates*
  - $\leq_{tl,tr}$, $=_{tl,tr}$ *with arity* $(\Omega \to \mathbb{R}^+) \times (\Omega \to \mathbb{R}^+)$, *where* $tl, tr \in FunT$.
  - $\leq_t$, $\geq_t$ *and* $=_t$ *with arity* $(\Omega \to \mathbb{R}^+) \times \mathbb{R}$, *where* $t \in FunT$.
- *Axioms*

  (1) For each $\Psi \in FunPS$ *and each* $tl, tr \in FunT$ *such that* $\forall o \in \mathbb{R} : tl(o) \leq tr(o)$*, there is an axiom*

$$\Psi \leq_{tl,tr} \Psi$$

(2) For each $\Psi_M, \Psi_N \in FunPS$ *and each* $tm, tn \in FunT$*, there is an axiom*

$$(\Psi_M \leq_{tm,tn} \Psi_N \ \wedge \ \Psi_N \leq_{tn,tm} \Psi_M) \quad \leftrightarrow \quad \Psi_M =_{tm,tn} \Psi_N$$

(3) For each $\Psi \in FunPS$*, each* $\mu_0 \in \mathbb{R}$ *and each* $ts, tb \in FunT$ *such that* $\forall o \in \mathbb{R} : ts(o) \leq tb(o)$*, there are axioms*

$$\Psi \leq_{tb} \mu_0 \quad \rightarrow \quad \Psi \leq_{ts} \mu_0$$

and

$$\Psi \geq_{ts} \mu_0 \quad \rightarrow \quad \Psi \geq_{tb} \mu_0$$

(4) For each $\Psi \in FunPS$*, each* $\mu_0 \in \mathbb{R}$ *and each* $t \in FunT$*, there is an axiom*

$$\Psi \leq_t \mu_0 \ \wedge \ \Psi \geq_t \mu_0 \quad \leftrightarrow \quad \Psi =_t \mu_0$$

The function symbols *FunT*, predicates $\leq_{tl,tr}$ and $=_{tl,tr}$ and the first two axioms are the same as in SPL. The meaning of the rest is explained in the following paragraphs.

The function symbols *FunPS* are CSPL's abstractions of a performance measuring process designated to measure execution times of some specific method.

New predicates $\leq_t$, $\geq_t$ and $=_t$ are introduced in order to facilitate the absolute time bounds based performance testing approach. In the next chapter, an interpretation will specify how these predicates should be evaluated by statistical tests. The symbol Ψ represents a method; the number μ_0 represents an absolute time bound. By using these predicates it will be possible to express assumptions similar to those from the following examples.

Example 2. *"In order for a platform to meet application's minimal requirements, reseting values of an integer array with one million elements to* 0 *in a for-cycle should not be slower than* 10*ms."*

$$\Psi_{ArrayReset} \leq_{id} 10ms$$

Example 3. *"According to a specification of a process scheduling algorithm, the mean length of a continuous computational time that is granted to each process in a round-robin manner should never be shorter than 10 milliseconds or greater than 30 milliseconds."*

The symbol $\Psi_{Process}$ *in the following equation stands for a random variable that observes lengths of time slices that some selected process spends in continuous computation.*

$$\Psi_{Process} \geq_{id} 10ms \ \wedge \ \Psi_{Process} \leq_{id} 30ms$$

Just as in SPL, the examples use lambda notation [9] for compact representation of performance transformation functions (symbol *id* is a shortcut for $\lambda x.x$).

Finally, CSPL introduces two new axioms – (3) and (4). They are used to emphasize the purpose of the logic and to ensure that the logic meets basic expectations. This is especially useful for interpretations based on hypothesis testing, because the axioms are partly tailored to fit such interpretation. The

third axiom may be regarded as a multiplication by a constant, where *a)* further improvement of a method's performance or *b)* further deterioration of a method's performance does not make it break its relation with the original bound to an opposite one. The fourth axiom is similar to the second one and shows the correspondence between $\leq_t$, $\geq_t$ and $=_t$.

Before concluding this chapter, it still remains to show that SPL's quantification over *workload parameters* may be achieved also in CSPL. The idea is very simple as it is based on a straightforward formula expansion. In SPL it is possible to quantify over finite subsets of *workload parameter* domains. In CSPL this would mean quantification over a finite number of *performance samplers*, where each sampler measures the same method, only with different input parameters. Therefore, when a quantifier would be applied to some SPL formula, in CSPL the quantification get expanded into N formulas, where N is the size of the finite subset of the *workload parameter* domain over which it is quantified. The result of those N formulas would be then compared together and a result would be decided based on the type of the quantifier. The idea is illustrated by the following examples that capture the same assumptions – the first example is formulated in SPL, the other in CSPL.

Example 4. *"On integer arrays with 1, 2, 3 and 4 elements, the Bubble Sort algorithm is faster than the Quick Sort algorithm."*

Captured in SPL, the formula reads like:

$$\forall n \in \{1, 2, 3, 4\} : P_{Bubble}(n) \leq_{p(id,id)} P_{Quick}(n)$$

Example 5. *"On integer arrays with 1, 2, 3 and 4 elements, the Bubble Sort algorithm is faster than the Quick Sort algorithm."*

Captured in CSPL, the specification is as follows.

Let there be four performance samplers Ψ_{B,G_1}, Ψ_{B,G_2}, Ψ_{B,G_3} *and* Ψ_{B,G_4} *for the Bubble Sort algorithm executed on data from random integer array generators* G_1, G_2, G_3 *and G4, where each generator* G_N *generates arrays of* N *elements. Performance samplers* Ψ_{Q,G_1}, Ψ_{Q,G_2}, Ψ_{Q,G_3} *and* Ψ_{Q,G_4} *for the Quick Sort algorithm are defined similarly.*

Let there be four formulas:

$$\Phi_1 : \Psi_{B,G_1} \leq \Psi_{Q,G_1}$$

$$\Phi_2 : \Psi_{B,G_2} \leq \Psi_{Q,G_2}$$

$$\Phi_3 : \Psi_{B,G_3} \leq \Psi_{Q,G_3}$$

$$\Phi_4 : \Psi_{B,G_4} \leq \Psi_{Q,G_4}$$

Then the assumption is considered satisfied if and only if all the formulas Φ_1, Φ_2, Φ_3 *and* Φ_4 *are evaluated as true.*

To give a meaning to the CSPL logic from this chapter, it is still necessary to define a semantics for it. An interpretation based on statistical tests is introduced in the next chapter.

Chapter 2

Interpretations

2.1 Overview

The objective of this chapter is to give the CSPL logic an interpretation so that it is possible to evaluate its formulas. Like in [7], first an interpretation that has only a theoretical purpose is described in section 2.2. It is named *expected value based interpretation* and it shows in a very simple manner the CSPL's intended use. The interpretation builds on a theory presented in the article [7], so only CSPL specific semantics and theorems are to be found here in the thesis.

On the other hand, section 2.3 is focused on *sample value based interpretation* that does have a practical use. It even forms a cornerstone of the CSPL evaluation engine described later in chapter 3. Similar to the *expected value based interpretation*, this interpretation is also based on a theory from the article [7] and thus only CSPL specific additions are presented here.

The thesis, however, moves beyond the original vision of the article by generalizing the concept of the introduced *sample value based interpretation*. Section 2.4 discusses how an arbitrary number of interpretations based on statistical tests can be used when deciding validity of a formula – an approach supported by the implemented evaluation engine.

2.2 Expected Value Based Interpretation

As justified by [7], comparison of expected values is a straightforward natural way to compare random variables. In SPL the random variable in question is *method performance* – a function parametrized by a set of *workload parameters*. In CSPL *method performance* is replaced by *performance sampler*. The difference between those two is minimal, because as soon as *method performance* is assigned its parameters it becomes $\Omega \to \mathbb{R}$, while *performance sampler* is defined as $\Omega \to \mathbb{R}^+$ (section 1.3). Therefore the expected value based interpretation described here will use results from [7] in its definition.

Each function symbol $f_{PS} \in FunPS$ is interpreted as a *performance sampler* (section 1.3) whose observations are execution times of some method M for which input arguments have been obtained from G_M – a generator of random arguments for M. Other attributes that affect M's performance (including, but not limited to the state of the object on which the method is invoked or more generally

the state of the entire application) are considered to be an implicit part of M's specification. These are, together with specification of G_M, programmer's task to decide. Broader discussion of this may be found in the last section of this chapter.

Each function symbol $f_T \in FunT$ is interpreted as an arbitrary performance observation transformation function $\mathbb{R}^+ \to \mathbb{R}$. In *expected value based interpretation*, it is possible to afford such loose definition because the interpretation is just theoretical and it is inconsequential if some f_T changes the distribution of a random variable to whose observations it is applied. Note that this will not be the case for *sample based interpretation*.

The interpretation of relation operators $\leq_{tl,tr}$ and $=_{tl,tr}$ is also taken from [7]. Only *method performance* is replaced by *performance sampler*, but otherwise the definition remains unchanged.

The relation operators $\leq_t$, $\geq_t$ and $=_t$ are defined as follows:

Definition. *Let $\Psi : \Omega \to \mathbb{R}^+$ be a performance sampler, $t : \mathbb{R}^+ \to \mathbb{R}$ a performance observation transformation function and $\mu_0 \in \mathbb{R}$ a bound. Then the relations $\leq_t, \geq_t, =_t$: $(\Omega \to \mathbb{R}^+) \times \mathbb{R}$ are interpreted as follows:*

$$\Psi \leq_t \mu_0 \quad \textit{iff} \quad E(t(\Psi)) \leq \mu_0;$$

$$\Psi \geq_t \mu_0 \quad \textit{iff} \quad E(t(\Psi)) \geq \mu_0;$$

$$\Psi =_t \mu_0 \quad \textit{iff} \quad E(t(\Psi)) = \mu_0,$$

where $E(X)$ stands for the expected value of a random variable X and $t(X)$ for a random variable that is created from X by applying function t on each observation.

The last definition concluded description of semantics for the expected value based interpretation. However, it is still left to demonstrate the consistency of the semantics with all CSPL axioms. A proof for the first and the second axiom is not presented here as it would require only a slight modification of Theorem 1 from [7, p. 4].

The third and the fourth axiom are new to CSPL and therefore new proofs are necessary. First a lemma is introduced to cover CSPL-agnostic properties of expected value. With the lemma in place, a theorem finishes the job of showing that the interpretation of $\leq_t$, $\geq_t$ and $=_t$ is consistent with the both axioms.

Lemma 1. *Let $X : \Omega \to \mathbb{R}$ be a random variable, $t, t_s, t_b : \mathbb{R} \to \mathbb{R}$ arbitrary functions over real numbers and $\mu_0 \in \mathbb{R}$. Then the following holds:*

$$(\forall o \in \mathbb{R} : t_s(o) \leq t_b(o)) \ \wedge \ E(t_b(X)) \leq \mu_0 \quad \to \quad E(t_s(X)) \leq \mu_0$$

$$(\forall o \in \mathbb{R} : t_s(o) \leq t_b(o)) \ \wedge \ E(t_s(X)) \geq \mu_0 \quad \to \quad E(t_b(X)) \geq \mu_0$$

$$E(t(X)) \leq \mu_0 \ \wedge \ E(t(X)) \geq \mu_0 \quad \to \quad E(t(X)) = \mu_0$$

Proof. The validity of the first two formulas may be easily derived from the definition of the expected value. Let $f(x)$ be the probability density function of a random variable X. Because $f(x) \geq 0$, it holds that

$$E(t_s(X)) = \int_{-\infty}^{+\infty} t_s(x) f(x) dx \leq \int_{-\infty}^{+\infty} t_b(x) f(x) dx = E(t_b(X)) \leq \mu_0,$$

similarly

$$E(t_b(X)) = \int_{-\infty}^{+\infty} t_b(x)f(x)dx \geq \int_{-\infty}^{+\infty} t_s(x)f(x)dx = E(t_s(X)) \geq \mu_0.$$

For a case when X is a discrete random variable, the proof remains the same only the integral is replaced by a sum.

The validity of the third formula flows directly from the total ordering over real numbers. □

Now it is possible to move to the theorem that concludes the formal definition of expected value based CSPL interpretation.

Theorem 1. *The interpretation of relation operators $\leq_t$, $\geq_t$ and $=_t$ is consistent with axioms (3) and (4).*

Proof. The proof is easily obtained from lemma 1 by assigning Ψ as X and interpreting t, t_s and t_b as performance observation transformation functions. □

This section has illustrated that the expected value based approach is a valid interpretation of CSPL logic. The approach has however no practical use, because it is very difficult or even impossible to determine expected value of execution times for an arbitrary method. A remedy to this problem is presented in the following section.

2.3 Sample Based Interpretation

A more practical interpretation that this section introduces is based on statistical tests. Instead of requiring knowledge of expected values for random variables and their transformations, it will be now sufficient just to collect a sample of execution times. This is a much welcomed improvement, because in real-world software projects programmers cannot spend development time determining expected execution times of methods they write. With the sample based interpretation the user input is reduced merely to a selection of a statistical test that should be used for evaluation. The exact range of statistical tests offered is a matter of a particular evaluation engine. This thesis aims at high modularity and the evaluation engine provided (called UnitRacer, chapter 3) is freely extensible with new tests. However, in order to make one of the most useful tests ready for the engine, this section needs to provide a required theory first. It focuses on definition of CSPL interpretation that is based on statistical tests for mean of normally distributed random variables. This theory may be used in the future as a template for definitions and validity proofs of other statistical tests – those may be either for random variables with different distribution or for normally distributed variables but a different measure (e.g. variance). A further discussion of this topic is in section 2.4.

In order for the following interpretation to be well defined, it is first necessary to fix a set of observations. These observations will be parameters of the interpretation (similarly to [7, p. 4]). This is a necessity since the interpretation is based on hypothesis testing and with different sets of observations a hypothesis

may get rejected or not rejected. Since it is possible to use any set of execution time observations this parametrization is not an obstacle for the practical implementation. Formally, each performance unit test based on this interpretation will first measure execution times, then construct the interpretation theory with measured samples and finally, using this theory, evaluate the test.

The following definition of *experiment* is nearly the same as the one from [7, p. 4], except *method performance* is replaced with *performance sample*.

Definition. *Let $\mathcal{O}_\Psi = \{\Psi^1, \ldots, \Psi^n\}$ be a set of n observations of a performance sampler Ψ, where Ψ^i denotes i-th observation. Then experiment $\mathcal{E}$ is a collection of such $\mathcal{O}$ sets.*

Example 6. *When comparing performance of the Bubble Sort and the Quick Sort algorithms, each one of them has been executed five times and their performance has been measured in milliseconds. This yielded the following sets of observations – $\mathcal{O}_{Bubble} = \{42, 53, 64, 48, 55\}$ and $\mathcal{O}_{Quick} = \{24, 15, 62, 22, 21\}$.*

The collection $\mathcal{E} = \{\mathcal{O}_{Bubble}, \mathcal{O}_{Quick}\}$ is therefore an example of an experiment*.*

Now it is possible to define the *sample based interpretation* of CSPL, parametrized for some experiment $\mathcal{E}$. The definition is very close to the one from the previous section, except for the interpretation of performance observation transformation function and relation operators ($\leq_{tl,tr}$, $=_{tl,tr}$, $\leq_t$, $=_t$ and $\geq_t$).

Performance observation transformation functions are defined in *sample value based interpretation* as follows:

Definition. *Each function symbol $f_T \in FunT$ is interpreted as a function $\mathbb{R}^+ \to \mathbb{R}$ in one of the two following forms:*

- *$f_T = ax$, where $a \in \mathbb{R}^+$*
- *$f_T = x + b$, where $b \in \mathbb{R}$*

These specific forms have been chosen, because they allow a programmer to capture performance assumptions of the most common scenarios and at the same time, when they are applied to a normally distributed random variable the result is again a random variable with normal distribution.[1] The later fact is especially important as it allows the interpretation to satisfy CSPL axiom (3), as shown further in the text.

The interpretation of relation operators $\leq_{tl,tr}$ and $=_{tl,tr}$ with arity $(\Omega \to \mathbb{R}^+) \times (\Omega \to \mathbb{R}^+)$ is the same as in [7, p. 4], except *method performance* is again replaced by *performance sampler*. Otherwise theorems and their proofs are identical to what would otherwise be written here. This means that the Welch's t-test [11] is used for evaluation of these relations.

In order to interpret the new relation operators $\leq_t$, $=_t$ and $\geq_t$ with arity $(\Omega \to \mathbb{R}^+) \times \mathbb{R}$ it is necessary to select a statistical test in the same manner as [7] has selected the Welch's t-test, but with the difference that the new test must be a one-sample test for mean of a random variable with normal distribution. Based on these requirements, the Student's t-test has been chosen.

[1] The statement comes from a more general attribute of normally distributed random variables that their linear functions also have normal distributions, e.g., [10].

Definition. *Let $\Psi : \Omega \to \mathbb{R}^+$ be a performance sampler, $t : \mathbb{R}^+ \to \mathbb{R}$ a performance observation transformation function, $\mu_0 \in \mathbb{R}$ a bound and $\alpha \in [0, 0.5]$ a fixed significance level.*

For a given experiment $\mathcal{E}$, the relations $\leq_t$, $=_t$ and $\geq_t$ are interpreted as follows:

- $\Psi \leq_t \mu_0$ *iff the null hypothesis*

  $$H_0 : E(\{t(\Psi^1), \ldots, t(\Psi^n)\}) \leq \mu_0$$

  cannot be rejected by one-sided Student's t-test at significance level α based on the observations gathered in the experiment $\mathcal{E}$;

- $\Psi \geq_t \mu_0$ *iff the null hypothesis*

  $$H_0 : E(\{t(\Psi^1), \ldots, t(\Psi^n)\}) \geq \mu_0$$

  cannot be rejected by one-sided Student's t-test at significance level α based on the observations gathered in the experiment $\mathcal{E}$;

- $\Psi =_t \mu_0$ *iff the null hypothesis*

  $$H_0 : E(\{t(\Psi^1), \ldots, t(\Psi^n)\}) = \mu_0$$

  cannot be rejected by two-sided Student's t-test at significance level 2α based on the observations gathered in the experiment $\mathcal{E}$,

where $E(\{t(\Psi^1), \ldots, t(\Psi^n)\})$ stands for the mean value of performance sampler observations from set $\mathcal{O}_\Psi$ in experiment $\mathcal{E}$ transformed by function t.

With the definition in place, it is now necessary to prove that the introduced semantics satisfies all CSPL's axioms. Theorems and proofs for axioms (1) and (2) are formulated in [7], the rest follows.

For purposes of theorem proving, the one-sample Student's t-test rejects the null hypothesis $\overline{X} \geq \mu_0$ against the alternative hypothesis $\overline{X} < \mu_0$ with significance level α if

$$\frac{\overline{X} - \mu_0}{\frac{\sigma}{\sqrt{n}}} < t_{\nu,\alpha}$$

and rejects the null hypothesis $\overline{X} = \mu_0$ against the alternative hypothesis $\overline{X} \neq \mu_0$ with significance level α if

$$\left| \frac{\overline{X} - \mu_0}{\frac{\sigma}{\sqrt{n}}} \right| < t_{\nu,\alpha/2}$$

where σ is the sample standard deviation, n is the sample size and $t_{\nu,\alpha}$ is the α-quantile of Student's distribution with $\nu = n - 1$ levels of freedom.

First a lemma is introduced that will make reasoning about Student's t-test easier. Its goal is to show that a shift by a constant has no impact on the result of such test.

Lemma 2. *Let $\{x^1, \ldots, x^n\}$ be a set of n observations collected from a normally distributed random variable X. Let μ_0 be a bound. Then the Student's t-test rejects the null hypothesis*

$$H_0 : X \geq \mu_0$$

based on observations $\{x^1, \ldots, x^n\}$ iff the Student's t-test rejects null the hypothesis

$$H_{S0} : X \geq 0$$

based on observations $\{x_S^1, \ldots, x_S^n\}$, where $x_S^i = x^i - \mu_0$.

Proof. The proof compares t-score values of Student's t-test equations for H_0 and H_{S0}. Because

$$\begin{aligned}
\overline{X_S} &= \frac{1}{n}\sum_{i=1}^{n} x_S^i \\
&= \frac{1}{n}\sum_{i=1}^{n}(x^i - \mu_0) \\
&= \frac{1}{n}\sum_{i=1}^{n} x^i \; - \; \frac{1}{n}\sum_{i=1}^{n}\mu_0 \\
&= \overline{X} - \mu_0
\end{aligned}$$

and

$$\begin{aligned}
\sigma_S &= \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_S^i - \overline{X_S})^2} \\
&= \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x^i - \mu_0 - \overline{X} + \mu_0)^2} \\
&= \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x^i - \overline{X})^2} \\
&= \sigma
\end{aligned}$$

it is possible to write

$$t = \frac{\overline{X} - \mu_0}{\frac{\sigma}{\sqrt{n}}} = \frac{\overline{X_S} - 0}{\frac{\sigma_S}{\sqrt{n}}} = t_S$$

which concludes the proof. □

The next lemma describes properties of performance observation transformation functions.

Lemma 3. *Let $t_s, t_b \in FunT$ be performance observation transformation functions such that $\forall o \in \mathbb{R}^+ : t_s(o) \leq t_b(o)$, then they must be in one of the following forms:*

(a) $t_s = a_s x,\ t_b = a_b x,$ *where* $0 < a_s \leq a_b$

(b) $t_s = a_s x,\ t_b = x + b_b,$ *where* $0 < a_s \leq 1, b_b \geq 0$

(c) $t_s = x + b_s,\ t_b = a_b x,$ *where* $b_s \leq 0, a_b \geq 1$

(d) $t_s = x + b_s,\ t_b = x + b_b,$ *where* $b_s \leq b_b$

Proof. The lemma will be proven by individual points, one by one. For each point it will be proved that the relation $\forall o \in \mathbb{R}^+ : t_s(o) \leq t_b(o)$ holds and that for opposite constraints it does not.

(a) ⋆ $t_s = a_s x,\ t_b = a_b x$, where $0 < a_s \leq a_b$

$$\begin{aligned} t_s &= a_s x && ; x \in \mathbb{R}^+, 0 < a_s \leq a_b \\ &\leq a_b x \\ &\leq t_b \end{aligned}$$

⋆ $t_s = a_s x,\ t_b = a_b x$, where $0 < a_b < a_s$

$$\begin{aligned} t_s &= a_s x && ; x \in \mathbb{R}^+, 0 < a_b \leq a_s \\ &> a_b x \\ &> t_b \end{aligned}$$

(b) ⋆ $t_s = a_s x,\ t_b = x + b_b$, where $0 < a_s \leq 1, b_b \geq 0$

$$\begin{aligned} t_s &= a_s x && ; x \in \mathbb{R}^+, 0 < a_s \leq 1 \\ &\leq x && ; b_b \geq 0 \\ &\leq x + b_b \\ &\leq t_b \end{aligned}$$

⋆ $t_s = a_s x,\ t_b = x + b_b$, where $1 < a_s, b \in \mathbb{R}$
First it will be proven that there is a $x_i \in \mathbb{R}$ such that $t_s(x_i) = t_b(x_i)$.

$$\begin{aligned} a_s x_i &= x_i + b_b \\ (a_s - 1)x_i &= b_b && ; a_s > 1 \\ x_i &= \frac{b_b}{a_s - 1} \end{aligned}$$

Because intersection x_i really exists and t_s grows faster than t_b ($1 < a_s$), it is true that $\forall \bar{x} > x_i : t_s(\bar{x}) > t_b(\bar{x})$ which is the sought-after contradiction.

⋆ $t_s = a_s x,\ t_b = x + b_b$, where $0 < a_s \leq 1, b_b < 0$
If $a_s = 1$, then

$$\begin{aligned} t_s(1) &= a_s \cdot 1 && ; a_s = 1 \\ &= 1 && ; b_b < 0 \\ &> 1 + b_b \\ &> t_b(1). \end{aligned}$$

Otherwise there is again a point of intersection $x_i = \frac{b_b}{a_s - 1}$ and arguments similar to those from the previous bullet apply.

(c) $\star$ $t_s = x + b_s,\ t_b = a_b x$, where $b_s \leq 0, a_b \geq 1$

$$\begin{aligned} t_s &= x + b_s && ; b_s \leq 0 \\ &\leq x && ; x \in \mathbb{R}^+, a_b \geq 1 \\ &\leq a_b x \\ &\leq t_b \end{aligned}$$

$\star$ $t_s = x + b_s,\ t_b = a_b x$, where $b_s > 0, a_b > 0$

If $a_b \leq 1$, then functions t_s and t_b never intersect and because $b_s > 0$ the function t_s yields higher values for any $x \in \mathbb{R}^+$.

When $a_b > 1$, then there is a point of intersection $x_i = \frac{b_s}{a_b - 1}$ and because t_b grows faster than t_s and $b_s > 0$, it implies that $x_i > 0$. Therefore $\forall 0 < \bar{x} < x_i : t_s(\bar{x}) > t_b(\bar{x})$.

$\star$ $t_s = x + b_s,\ t_b = a_b x$, where $b_s \in \mathbb{R}, 0 < a_b < 1$

If $b_s \geq 0$, then functions t_s and t_b never intersect and because $0 < a_b < 1$ the function t_s grows faster and yields higher values for any $x \in \mathbb{R}^+$.

Otherwise functions t_s and t_b have an intersection $x_i = \frac{b_s}{a_b - 1}$ and because t_s grows faster than t_b, it holds that $\forall \bar{x} > x_i : t_s(\bar{x}) > t_b(\bar{x})$.

(d) $\star$ $t_s = x + b_s,\ t_b = x + b_b$, where $b_s \leq b_b$

$$\begin{aligned} t_s &= x + b_s && ; b_s \leq b_b \\ &\leq x + b_b \\ &\leq t_b \end{aligned}$$

$\star$ $t_s = x + b_s,\ t_b = x + b_b$, where $b_s > b_b$

$$\begin{aligned} t_s &= x + b_s && ; b_s > b_b \\ &> x + b_b \\ &> t_b \end{aligned}$$

□

The following theorem verifies the validity of axiom (3) in the sample based interpretation of CSPL.

Theorem 2. *The interpretation of relation operators $\leq_t$ and $\geq_t$ is consistent with with axiom (3) for a fixed experiment $\mathcal{E}$.*

Proof. Let $\Psi \in FunPS$, $\mu_0 \in \mathbb{R}$ and $t_s, t_b \in FunT$ such that $\forall o \in \mathbb{R}^+ : t_s(o) \leq t_b(o)$. Then there are two formulas whose validity needs to be proven

$$\Psi \leq_{tb} \mu_0 \quad \rightarrow \quad \Psi \leq_{ts} \mu_0$$

and

$$\Psi \geq_{ts} \mu_0 \quad \rightarrow \quad \Psi \geq_{tb} \mu_0$$

but due to their similarity, only the second one will be demonstrated.

To make the proof more compact, a notation for expected values is introduced as follows:

$$\overline{\Psi_s} = E(\{t_s(\Psi^1), \ldots, t_s(\Psi^n)\}) \text{ and}$$
$$\sigma_s = \sqrt{Var(\{t_s(\Psi^1), \ldots, t_s(\Psi^n)\})},$$

where E marks sample mean and Var sample variance. Symbols $\overline{\Psi_b}$ and σ_b are defined in a similar way.

The goal of this proof is hence to show that when the null hypothesis $H_{0s} : \overline{\Psi_s} \geq \mu_0$ cannot be rejected by the Student's t-test, then also the null hypothesis $H_{0b} : \overline{\Psi_b} \geq \mu_0$ cannot be rejected.

First, lemma 2 is used to shift μ_0 and all observations of Ψ by $-\mu_0$. As a result, further notation is hereby established:

$$\overline{\Psi_s^\triangle} = E(\{t_s(\Psi^1 - \mu_0), \ldots, t_s(\Psi^n - \mu_0)\}),$$
$$\sigma_s^\triangle = \sqrt{Var(\{t_s(\Psi^1 - \mu_0), \ldots, t_s(\Psi^n - \mu_0)\})}$$

and symbols $\overline{\Psi_b^\triangle}$ and $\sigma_b^\triangle$ are again defined similarly.

Lemma 3 is used to divide the proof into four cases based on possible t_s and t_b forms. Individual cases are proven by showing that the t-score for H_{0b} is bigger than the t-score for H_{0s}.

(a) $t_s = a_s x,\ t_b = a_b x$, where $0 < a_s \leq a_b$

$$tscore_s \leq tscore_b$$
$$\frac{\overline{\Psi_s^\triangle} - 0}{\frac{\sigma_s^\triangle}{\sqrt{n}}} \leq \frac{\overline{\Psi_b^\triangle} - 0}{\frac{\sigma_b^\triangle}{\sqrt{n}}}$$
$$\frac{\overline{\Psi_s^\triangle}}{\sigma_s^\triangle} \cdot \frac{\sqrt{n}}{1} \leq \frac{\overline{\Psi_b^\triangle}}{\sigma_b^\triangle} \cdot \frac{\sqrt{n}}{1}$$
$$\frac{a_s \overline{\Psi^\triangle}}{a_s \sigma^\triangle} \leq \frac{a_b \overline{\Psi^\triangle}}{a_b \sigma^\triangle}$$
$$1 \leq 1$$

(b) $t_s = a_s x,\ t_b = x + b_b$, where $0 < a_s \leq 1, b_b \geq 0$

$$tscore_s \leq tscore_b$$
$$\frac{a_s \overline{\Psi^\triangle}}{a_s \sigma^\triangle} \leq \frac{\overline{\Psi^\triangle} + b_b}{\sigma^\triangle}$$
$$\overline{\Psi^\triangle} \leq \overline{\Psi^\triangle} + b_b$$
$$0 \leq b_b$$

(c) $t_s = x + b_s,\ t_b = a_b x$, where $b_s \leq 0, a_b \geq 1$

$$\begin{aligned} tscore_s &\leq tscore_b \\ \frac{\overline{\Psi^\triangle} + b_s}{\sigma^\triangle} &\leq \frac{a_b \overline{\Psi^\triangle}}{a_b \sigma^\triangle} \\ \overline{\Psi^\triangle} + b_s &\leq \overline{\Psi^\triangle} \\ b_s &\leq 0 \end{aligned}$$

(d) $t_s = x + b_s,\ t_b = x + b_b$, where $b_s \leq b_b$

$$\begin{aligned} tscore_s &\leq tscore_b \\ \frac{\overline{\Psi^\triangle} + b_s}{\sigma^\triangle} &\leq \frac{\overline{\Psi^\triangle} + b_b}{\sigma^\triangle} \\ \overline{\Psi^\triangle} + b_s &\leq \overline{\Psi^\triangle} + b_b \\ b_s &\leq b_b \end{aligned}$$

□

Theorem 2 has proved the validity of the sample based interpretation with CSPL axiom (3). It is still left to show that even the axiom (4) is valid in this interpretation.

Theorem 3. *The interpretation of relation operators $\leq_t$, $\geq_t$ and $=_t$ is consistent with axiom (4) for a fixed experiment $\mathcal{E}$.*

Proof. Let $\Psi \in FunPS$, $\mu_0 \in \mathbb{R}$ and $t \in FunT$. Similarly to the proof of theorem 2 a notation is first introduced:

$$\begin{aligned} \overline{\Psi_t} &= E(\{t(\Psi^1), \ldots, t(\Psi^n)\}) \text{ and} \\ \sigma_t &= \sqrt{Var(\{t(\Psi^1), \ldots, t(\Psi^n)\})}, \end{aligned}$$

where E marks sample mean and Var sample variance.

By assigning interpretation of the relation operators $\leq_t$, $\geq_t$ and $=_t$ to the equation from axiom (4)

$$\Psi \leq_t \mu_0 \ \wedge\ \Psi \geq_t \mu_0 \quad \leftrightarrow \quad \Psi =_t \mu_0$$

the following formula is the obtained:

$$\frac{\overline{\Psi_t} - \mu_0}{\frac{\sigma_t}{\sqrt{n}}} \leq -t_{\nu,\alpha} \quad \wedge \quad \frac{\overline{\Psi_t} - \mu_0}{\frac{\sigma_t}{\sqrt{n}}} \geq t_{\nu,\alpha} \qquad \leftrightarrow \qquad \left| \frac{\overline{\Psi_t} - \mu_0}{\frac{\sigma_t}{\sqrt{n}}} \right| \geq t_{\nu,\alpha}$$

And since the absolute value may be expanded to

$$t_{\nu,\alpha} \leq \frac{\overline{\Psi_t} - \mu_0}{\frac{\sigma_t}{\sqrt{n}}} \leq -t_{\nu,\alpha}$$

it is apparent that the axiom is valid just by replacing its symbols with their corresponding interpretations. □

2.4 Extensibility

As indicated in the previous section, the evaluation engine provided with this thesis is freely extensible with interpretations that are based on different statistical tests. This is possible as long as all the new interpretations share together some common rules. Namely, they must have the same interpretation of both function symbols $FunT$ and $FunPS$. On the other hand, as long as they can prove axioms (1) – (4), they are free to supply custom statistical tests as a mean of interpreting the relation operators.

The interpretation from the previous section uses the Welch's t-test to compare together means of random variables and the Student's t-test to compare a mean with an absolute time bound. In other scenarios it could be useful to have an interpretation for other than normally distributed random variables. These can be either tests for other specific distributions or non-parametric tests.

It is also possible for the interpretation to be aimed at a whole another measure. While the Welch's t-test compares means, other tests may compare median or even the distribution of random variables. These tests are, however, left unimplemented in the thesis and are a subject of potential future research.

2.5 Interpretation of Performance Sampler

There is a reason why the interpretation of function symbols from *FunPS* in section 2.2 is so cautious. It is because performance of a method is tightly coupled to the state of its environment. As described in introduction, it is obvious that performance is expected to change when switching hardware platforms. The same holds, e.g., for the state of the object on which the method is invoked. Consider the following example:

Example 7. *A method, whose performance is highly dependent on the state of its object. Written in* `C#`*.*

```
public class TestedClass {
    private int _counter = 0;
    public void TestedMethod() {
        Thread.Sleep(++_counter);
    }
}
```

The *TestedMethod* will get slower and slower over the time. Therefore, when a programmer is about to specify what exactly is behind a function symbol from *FunPS* he has to pay close attention, otherwise the resulting test would not be repeatable. In general, it is a question of a particular scenario as of what should be part of a function symbol and what not. It depends on expectations that test designers have towards the test.

The general rule of thumb is that anything that is not supposed to affect the result of some CSPL formula should not be a part of Ψ's specification. For example, hardware configuration or operating system are natural candidates for things that should be left out. On the other hand the state of the object from example 7 should definitely be considered.

Chapter 3

UnitRacer

While the previous two chapters were focused on introducing the theoretical aspects of the CSPL logic, the goal of this chapter will be to put the theory into practice. A library called UnitRacer, implemented for purposes of this thesis, is analyzed here in the text and by showing its various aspects and functionalities, it will be demonstrated how CSPL may be utilized in real-world software projects. The compiled library and its source codes are to be found on the DVD that accompanies this thesis[1], but its architecture, important design decisions and examples are described here in the following sections. To exercise UnitRacer in practice and demonstrate its benefits, chapter 4 presents several use cases that show how the library could have been integrated into selected software projects and possibly increased their quality.

The UnitRacer library has been developed for the .NET Framework platform. It is composed of several assemblies that together provide the required functionality. Some assemblies are just supporting ones, but there are two that have a special meaning for applications that want to employ UnitRacer in their quality assurance process. The first assembly is called simply `UnitRacer.dll` and is to be referenced from the application's test project in order to allow expressing and evaluating CSPL formulas. Best practices for writing performance unit tests with UnitRacer are presented later in this chapter. The other important assembly is an executable named `UnitRacer.ReportInspector.exe`. It allows detailed analysis of reports that are an optional by-product of CSPL formula evaluation. The inspector can be used to display a histogram of measured samples or a timeline that illustrates how execution times of some unit evolved during the sampling process.

To make expressing performance assumptions using CPSL formulas easy for programmers, the UnitRacer library utilizes some advanced programming techniques. The library also aims at comprehensible API and one of its main goals is to be usable even for developers that are not familiar with the concept of the CSPL logic. In order to achieve this, the task of writing test specification has been separated from the task of evaluating it. Thus a programmer only needs to write a simple equation and it is up to the evaluation engine hidden in the library to compute the result. By using .NET's feature called *lambda expressions*, it is even possible to provide a decent compile-time checking of formulas. The exact

[1] See appendix DVD.

notation and use of this capability is described further in this chapter. Another important feature is the run-time compilation of reflection expressions which eliminates a major performance penalty, while still enabling a programmer to use delegates and capture the test by a single short formula.

The library has been designed from the very start to be easily extensible and highly modular. Any of its essential components may be replaced at any time by a different implementation or a new component may be added without removing the original one. This is especially true for the collection of statistical tests that the library offers. It ships with the Student's t-test and the Welch's t-test on the basis of results from the previous chapter, but new tests may be added at will. In order to make the process of writing tests easier, the library also comes with an API for simple random data generation. The programmer's task is therefore simplified into just expressing the test itself without having to write any helper scaffolding first. But perhaps the most important advantage of the library is that its tests are stable and repeatable. This not only favors the library to other performance testing frameworks, but it also promotes performance testing as a whole because it removes the unreliability of tests with nondeterministic results.

This chapter is structured as follows. The top-level overview of the UnitRacer library as a software project is in section 3.1. Then the architecture is presented in section 3.2 and succeeding sections deal with descriptions of the most important components such as the component for capturing CSPL formulas using lambda expressions, the component for internal representation of those formulas by a syntactic tree or the component for evaluation and reporting. Section 3.7 is focused on ReportInspector, a UnitRacer support application that enables analysis of evaluated tests through a graphical interface. The chapter is concluded by section 3.8 which shows a worked example of a typical performance unit test with further notes on its portability and overall value.

3.1 Project Overview

The UnitRacer project has been written in the `C#` programming language. The decision to choose this language and the .NET Framework has been made, because they both provide several state of the art features that other mainstream platforms and programming languages do not. This in return allowed the library to become more simple and easier to use for its end users. The development has been carried out in Visual Studio 2010 Premium, which is a proprietary IDE by Microsoft. A free alternative is Visual Studio 2010 Express or MonoDevelop 3.0 – both of them are capable of opening the Visual Studio solution file (`.sln`) and compiling the project.

Compiled .NET assemblies that are the main artifacts of project's build process are targeted against .NET Framework 4.0. This requirement cannot be lowered to some older version of the .NET Framework, because only in the 4.0 version some advanced features used by UnitRacer are available. The project has been prepared from its very beginning to be independent of the Microsoft's .NET implementation and to enable its execution also under the Mono project. The version of Mono runtime tested for compatibility is 2.10.9. As of hardware, the compiled assemblies may be executed anywhere where the .NET Framework

or Mono are present, there are no additional requirements. The project is, however, dependent on two third-party libraries – *Math.NET Numerics* [12] and *ZedGraph* [13] which are distributed together with the project. The libraries are available under the MIT/X11 and the LGPLv2 license respectively.[2] There are no other such libraries or licenses that would somehow affect or restrict the way, in which the core project binaries are allowed to be distributed.

The source codes of the project are structured by Visual Studio compilation units called *projects*. This may be misleading since the word "project" is already used in its natural meaning to denote the UnitRacer library as a whole. Because of that, Visual Studio projects will be, whenever mentioned in an ambiguous context, prefixed with the mandatory "Visual Studio" words. Compilation output of each such Visual Studio project is a .NET assembly – a linkable library. UnitRacer has four such projects:

- **UnitRacer** – The main project whose assembly is to be referenced by applications that want to perform UnitRacer tests. It contains everything from grammar for CSPL formulas to evaluation engine that is able to decide their validity and output an XML report. Section 3.2 is focused on a detailed analysis of UnitRacer's architecture and design.
- **UnitRacer.ReportInspector** – Project that is compiled into an executable that allows analysis of formula evaluation reports through a graphical interface. It is built on top of the WinForms graphical framework. More about the ReportInspector including a short demonstration of its capabilities is to be found in section 3.7.
- **UnitRacer.Tests** – A collection of unit tests for units from the UnitRacer Visual Studio project. Tests use the Visual Studio Unit Testing framework which also makes it possible to measure code coverage.
- **UnitRacer.Utils** – This project bears no core logic of UnitRacer, it only contains some minor classes and helper methods that are missing from the .NET Framework and make coding easier and more expressive.

In order to build the project and produce all of its compilation artifacts, it is recommended to use the *MSBuild* utility that ships with the .NET Framework 4.0. By issuing the following command, all UnitRacer Visual Studio projects will be compiled.

```
MSBuild.exe /t:Rebuild /p:Configuration:Release UnitRacer.sln
```

On the Mono platform, the same can be achieved using the `xbuild` program or through the MonoDevelop IDE. Since the library is composed only from a small amount of assemblies, no special build script has been created.

During the development, selected quality assurance means have been employed. First of all, the Test Driven Design methodology helped to achieve loose coupling between project's components. The overall test coverage has settled at

[2]The licenses are included in compliance with their respective terms on the attached DVD.

approximately 45%[3] with about 130 unit tests. Secondly, all public types, methods and properties do have their XML documentation. And finally, with about 2200 lines of code, the core assembly has achieved maintainability index of 83 which is a good result, since the metric ranges from 0 to 100 and 0 - 9 denotes code that is hard to maintain, 10 - 19 code that is moderately maintainable and 20 - 100 code that is easy to maintain [14]. Positive values have been achieved also for cyclomatic complexity, depth of inheritance and class coupling.

3.2 Architecture

The goal of this section is to present the top-level architecture of the UnitRacer library. Perhaps the most important aspect is, how individual components are separated one from another based on their natural responsibilities. The library is not very demanding on other architectural aspects such as various measures of performance or security, although they have been taken into account during the initial process as well. Accordingly, this section will focus mainly on the mentioned separation of concerns in the library.

Briefly said, UnitRacer enables evaluation of CSPL formulas for purposes of performance unit testing. In practice, this is a complex process of which the formula evaluation is only one of many parts. First of all, it is necessary for the user to design a test. More specifically, to determine what units are to be tested, what input data should the units be executed on, what exact CSPL formula captures best the tested assumption and how the result should be interpreted.

The unit is typically a method on an instance of some class. The state of the instance is also part of the specification as described in the previous chapter, but once obtained, the only requirement on the unit is that it should be able to accept input data and let others execute it. This leads to a natural abstraction that treats the unit as a functor object – an object with a single method that takes an arbitrary input and does some computation on it. Functor object is a useful concept since it encapsulates the essence of what a tested unit is and enables it to be passed around. Diagram 3.1 gives a closer view of the functor representation in UnitRacer (detailed description follows in section 3.3).

Input data for functors is obtained from generator objects since this approach enables to have an arbitrary generation logic implemented for various scenarios. The generator also allows to create as many input arguments as necessary. This is important since performance testing based on statistical tests will need to create a sample of certain amount of measurements. It is up to the generator whether it will generate identical objects or whether there will be some random in the process.

The specification of a CSPL formula is on the other hand a relatively straightforward task. Once the test designer knows what performance assumption should be tested, the only thing left to do is to rewrite it in a syntax imposed by the logic.

[3] Measured by Visual Studio tools for the Visual Studio Unit Testing framework.

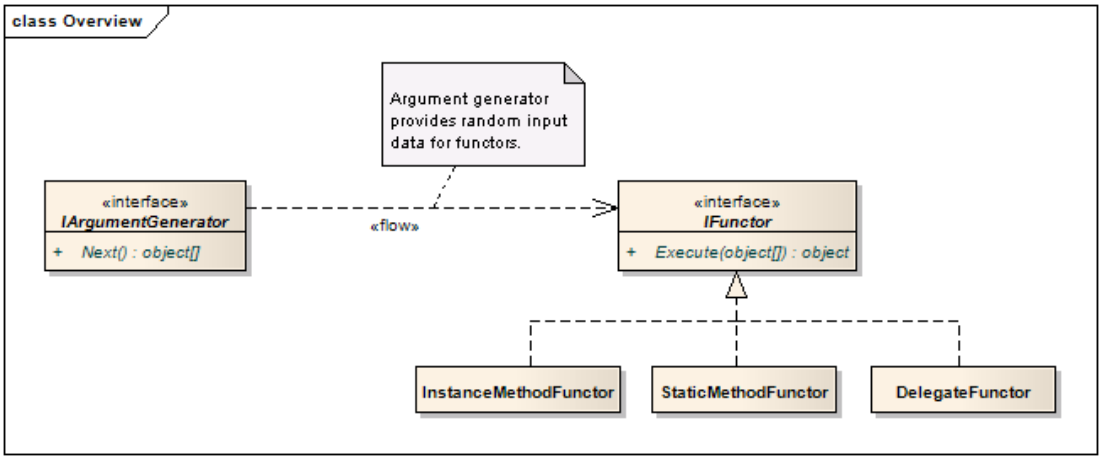


Figure 3.1: *IFunctor* and *IArgumentGenerator*.

Similarly, the interpretation of a test result is also not complicated. CSPL formulas evaluate either to *true* or *false*. It is common to write the test so that the formula is expected to hold – therefore *true* value is expected after the evaluation.

To sum up all previous test design tasks a brief list is provided in the spirit of the *arrangement-action-assertion* paradigm. The list also serves as a design template for a typical UnitRacer test (see figure 3.16 in section 3.8 for a working code snippet).

1. arrangement
   - Create *functor objects* that will represent the tested methods in the course of the test.
   - Get *input data generators* for the functors.
   - Capture the assumed relation between the units using a *CSPL formula*.
   - Obtain an instance of the *evaluation engine*.
2. action
   - Pass functors, input data generators and the CSPL formula to the evaluation engine and let it decide validity of the formula.
3. assertion
   - Check that the result of the evaluation is *true*, e.g., the formula holds.

The list is especially important because it shows how responsibilities are divided between the user and the library. In order to fulfill its responsibilities, UnitRacer is composed of several major components, where each component handles one separate concern. The individual components will be introduced in the following paragraphs.

The first component deals with the way users specify their tests. It is encapsulated by an interface because there may be an arbitrary number of particular strategies as how this is possible. The component provided with the library by default, enables users to specify CSPL formula with a lambda expression – a special feature of the `C#` programming language. But other implementations could

choose to parse plain text, XML or some other format. A common responsibility of these components is to collect the CSPL formula, functors and input argument generators. Since collecting these objects may be understood as a formal specification of an experiment of some kind, the component is called *Experiment*. The interface of this component requires, that no matter how the specification of the test has been collected, the component can parse it and convert it into a standardized format recognized by the evaluation component.

The internal representation of a test is defined in the *Grammar* component. The component has abstract data types for CSPL formulas so that their syntactic tree may be traversed and evaluated. It also provides the user with interfaces for various kinds of functors – starting with functors for instance and static methods to functors for *delegates*, another feature of the `C#` language. Abstractions are also ready for input argument generators and evaluation strategies that prescribe how the performance of some particular unit should be measured. This component is best described as a package of various data types that all together provide a common domain language for the entire library.

A test specification, as captured by the *Experiment* component and expressed by classes from the *Grammar* component, needs to be somehow evaluated. That is where the *Evaluation Engine* component comes in place. It exposes through a well defined interface a set of methods that take a test specification as their input and decide whether the CSPL formula of the test is satisfied. The component may be particulary complex as it needs to do the actual measurement and evaluation. The particular range of offered functionalities is a matter of an individual implementation, but every single one must, for example, let users extend and modify the set of statistical tests that the engine should use for evaluation of CSPL formulas.

One functionality that all implementations of the *Evaluation Engine* component have to support is generation of evaluation reports. The feature itself is provided by another stand-alone component called *Reporting*, but a precise communication between these components has to be adhered by both. Reporting may come in many different forms, but there is again one implementation made ready in the library. This is an XML reporting component that captures important data about test evaluation into an XML file that may be later opened by the ReportInspector application.

The following figure shows how all the major components of the library cooperate together:

To put components into a better perspective, the activity diagram in figure 3.3 shows how they communicate. The interaction starts with the user, who specifies the test in some input-dependent way, thus creating an experiment instance. For illustration, this can be an experiment parsed from a lambda expression that references all necessary functors and input data generators and has a form of a CSPL formula (more about this kind of experiment in section 3.4). The user then passes this experiment to an evaluation engine, which immediately requests the experiment to build an input-independent representation of the CSPL formula. With such representation it is possible to use a single evaluation logic that is custom to the engine without the necessity to understand multiple input formats. In general, the engine will measure execution times of methods referenced from

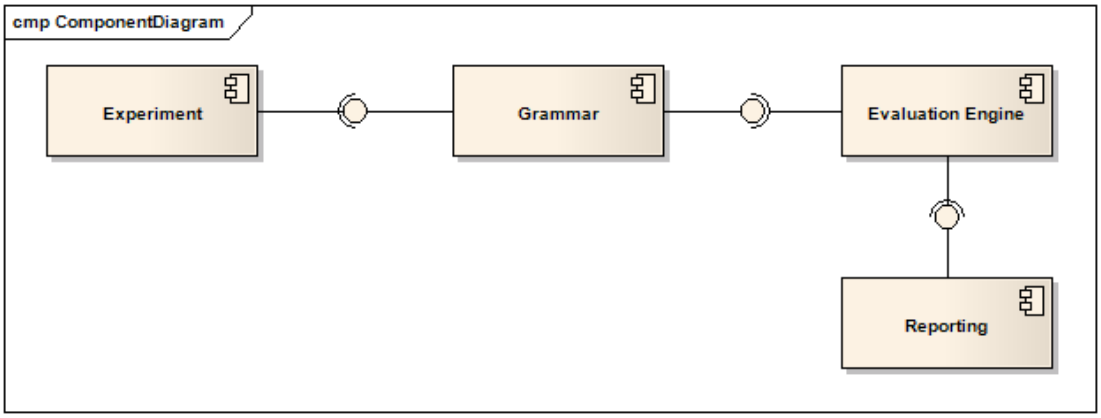


Figure 3.2: Major UnitRacer components

the formula and use statistical tests to decide formula's validity. Finally, it passes all necessary information to a report generator so that, e.g., an XML report may be written to a file.

Both diagrams 3.2 and 3.3 show the UnitRacer architecture with a great amount of abstraction. The individual components are described further in the following sections.

3.3 Grammar

The explanation of the individual components starts with *Grammar* since all other components are either directly or indirectly dependent on this one.

As written earlier the *Grammar* component is responsible for defining a common domain language for the UnitRacer library. The most fundamental concepts are associated with interfaces like `IFunctor`, `IPerformanceSampler`, `ISampling-Strategy`, `IArgumentGenerator` and most importantly `IFormula`.

3.3.1 Functors

The `IFunctor` interface, as indicated earlier, is an abstraction for a tested unit. Whether it is an instance or static method or a delegate, its particular form is hidden behind this interface (figure 3.4).

The library comes with a `FunctorFactory` class that offers factory methods that allow users to easily obtain `IFunctor` instances that represent units of their choice. The `FunctorFactory` relies on type inference, a special feature of the `C#` language, in a similar manner that the popular *Moq* [15] library does. The usage is illustrated by example 8. Similar may be achieved for static methods and delegates.

Example 8. *Imagine a class* `MD5Hash` *that has an instance method with signature* `string GetHash(string plaintext)`. *Let* `md5` *be an instance of the* `MD5Hash` *class, then the following line may be used in UnitRacer to obtain an* `IFunctor` *that represents the* `GetHash` *method of that instance.*

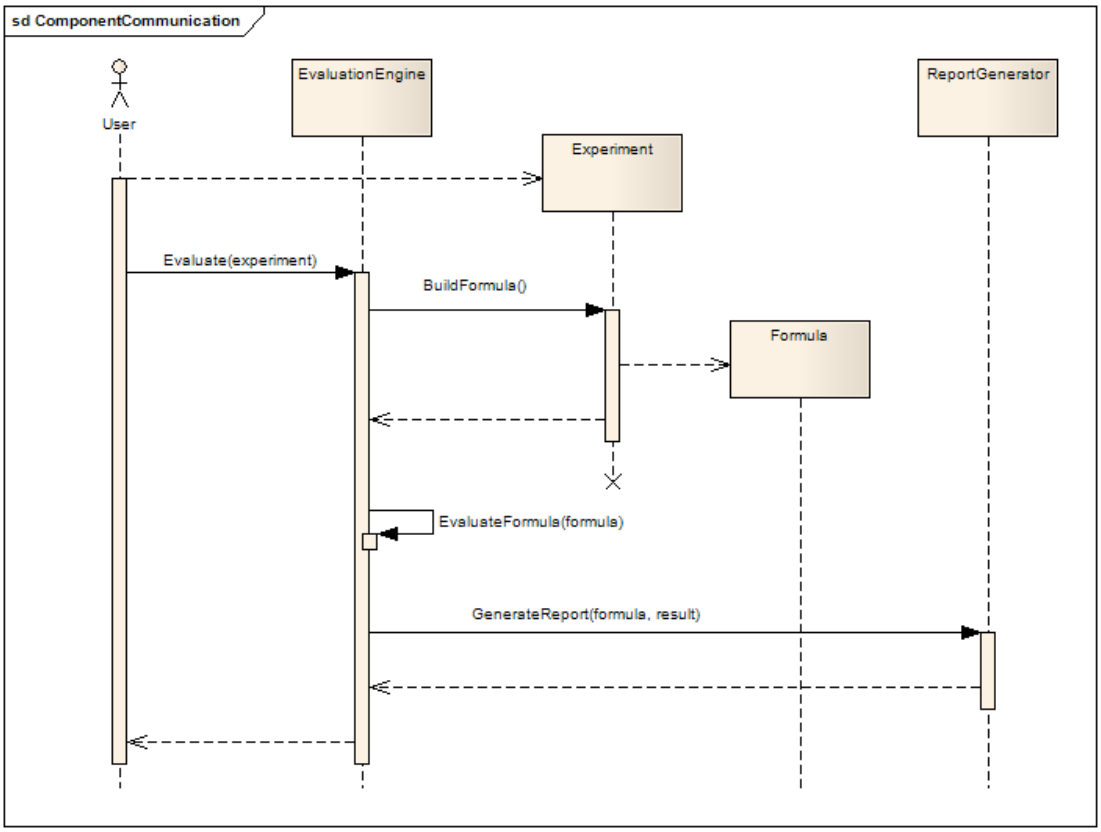


Figure 3.3: Communication between UnitRacer components

```
public interface IFunctor {
    object Execute(object[] arguments);
}
```

Figure 3.4: The *IFunctor* interface

```
IFunctor functor = FunctorFactory.Create(md5, m => m.GetHash(Arg.Of<string>()));
```

Before performance of a functor may be measured, it is first necessary to obtain some input data that could be passed to the functor as its input arguments. This is responsibility represented by the `IArgumentGenerator` interface, figure 3.5. It is completely up to implementors of this interface whether the process of argument generation will have some random aspect to it and whether the domain of generated arguments will be somehow restricted. Whatever the implementation is, it is often important that all compared functors receive arguments from the same (or at least similar) argument generator. This is a very natural requirement, because the size of input data usually affects performance. For example, the Quick Sort algorithm will be probably slower than the Bubble Sort algorithm, when given an array of million elements while the later algorithm will sort an array with only ten elements. The choice of argument generator implementation together with the decision to use one or multiple different generators for compared functors is completely up to the programmer, however.

```
public interface IArgumentGenerator {
    object[] Next();
}
```

Figure 3.5: The *IArgumentGenerator* interface

3.3.2 Performance Sampling

UnitRacer introduces three interfaces that are its main instruments for performance sampling – `IPerformanceSampler`, `ISamplingStrategy` and `ISampleSource`. They all cooperate together in order to get an array of transformed execution times of some functor. Their responsibilities and relations are described individually.

The first interface is UnitRacer's take on the *performance sampler* concept from the CSPL theory (chapter 1). For a reminder, the function $\Psi : \Omega \to \mathbb{R}^+$ is an abstraction of a method execution time sampling process. Typical implementation of `IPerformanceSampler` is expected to be injected with a functor, argument generator and a sampling strategy. The sampler uses the argument generator to create input for the functor and then measures the execution time. This is repeated until the sampling strategy that observes this process instructs the sampler to stop. The sampler then returns an array with the measured execution times.

Class `StandardPerformanceSampler` implements this logic with a simple loop that is executed on a single thread in order to prevent interference between different invocations of the sampled functor. This is on one hand time consuming, but on the other hand a very reliable approach. Code snippet is presented in figure 3.6. This is the only implementation provided by the library by default. Should some more complex logic be required in some scenario, the development team needs to write a custom code for it.

```
// measure
ulong value;
do
{
    // prepare arguments
    object[] arguments = Generator.Next();

    // measure next execution time
    _stopwatch.Restart();
    Functor.Execute(arguments);
    _stopwatch.Stop();
    value = _stopwatch.GetElapsedNanoseconds();
} while (Strategy.Put(value));

// collect
IEnumerable<ulong> result = Strategy.Collect();
```

Figure 3.6: Snippet from the *StandardPerformanceSampler* class.

The sampling strategy mentioned above is an interface that controls the sampling process. It tells the sampler what measurements are good, what measurements are to be discarded and when to stop the process. This is very useful, since it lets the programmer specify, for example, how many samples should be skipped at the beginning and how many samples should be then collected. For fast methods these numbers can be high, for slow methods it is practical to lower them so that the test does not take too long to evaluate.

Sampling strategies may be also extended by registration of so called sample filters. This functionality is used in the library to filter outliers and thus prepare better data for statistical tests, but any custom filters may be registered as well. The specific implementation of the `ISampleFilter` interface used in the library is further described in the section about the *Evaluation Engine* component because it closely relates to statistical tests used there.

The last introduced interface is `ISampleSource` that wraps around `IPerformanceSampler` and decorates the measured execution times by a linear transformation. The transformation corresponds to the *performance observation transformation function* concept from the CSPL theory. In UnitRacer, it is represented by a simple delegate that is first checked if it has one of the two permitted forms (ax or $x + b$).

3.3.3 Formula

Representation of a CSPL formula is handled in UnitRacer by `IFormula` interface. It puts together all concepts that have been introduced in this section so far. Not only does it represent the syntactic tree, it aggregates all the functors, performance samplers and similar objects that are necessary for evaluation. This means that only a single `IFormula` object may be passed around through the evaluation pipeline.

UnitRacer provides classes for all nodes of the syntactic tree, whose structure is best described by a diagram in figure 3.7.

Each concrete implementation of the `IFormula` interface must expose a property called `Value` which is a ternary boolean with value range $\{true, indeterminate, false\}$. The truth tables for operators !, $\wedge$ and $\vee$ are prescribed by the standard *Kleene's logic* [16]. The ternary boolean, represented by `TriBool` structure in the library, is necessary since after composing the formula syntactic tree some nodes do not have their value assigned and therefore are initialized to *indeterminate*. This is the case of `Hypothesis` nodes because only the *Evaluation Engine* component may later decide their value.

The `CompositeFormula` allows to combine multiple sub-formulas together using the operators $\wedge$ or $\vee$. The `NegatedFormula` negates value of a formula it wraps. Finally, implementors of `IAtom` are directly evaluated to either *true* or *false* which allows to recursively compute the value of formula's root node, where the value of the root is naturally the value of the entire formula. While the `ConstantAtom` has its value known from the beginning, the `Hypothesis` needs to have its value assigned by the *Evaluation Engine* component through measurement of execution times and statistical tests. The goal of the `Hypothesis` class is to capture a statistical hypothesis, therefore it can never happen that both its

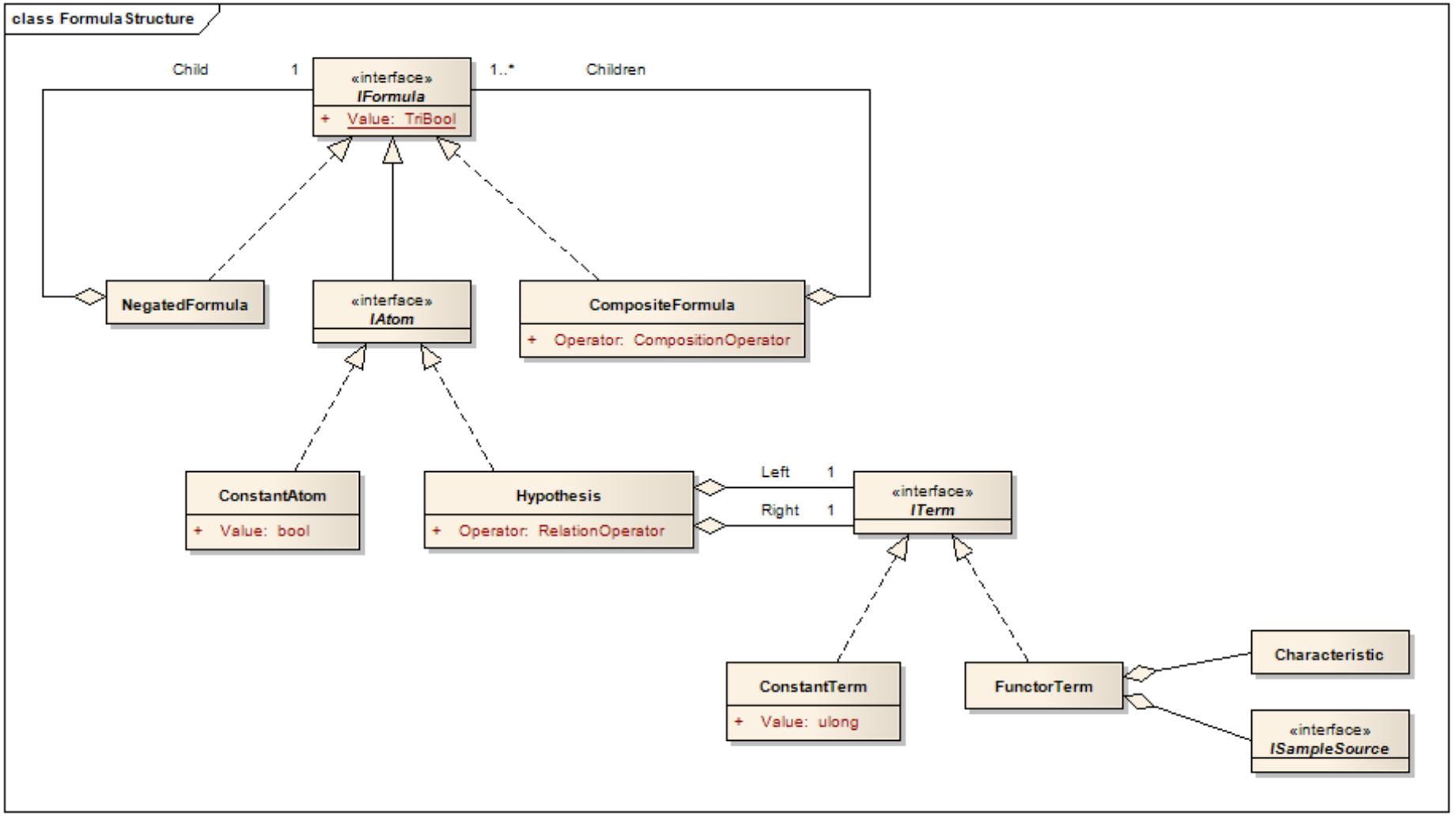


Figure 3.7: Class diagram of the *Grammar* component

terms would be `ConstantTerm` instances. For a one-sample test, either the left or the right term is `FunctorTerm` and the other is `ConstantTerm`. Two-sample test hypotheses have both terms in form of a `FunctorTerm`.

The purpose of the `ConstantTerm` is to express an absolute time bound. UnitRacer works in nanosecond units, therefore the `Value` property should be set appropriately. The `FunctorTerm` represents a random value and its tested characteristic. The random value is the `ISampleSource` that has been introduced previously, the tested characteristic is a hint for the *Evaluation Engine* component telling it what statistical test should be applied. UnitRacer ships with support for only one characteristic – *mean*, but when the library is extended with new tests, new characteristics may be added as well. This way it could be possible, for example, to test variance of a tested unit or its distribution.

3.4 Experiment

The *Experiment* component is very important to the user, because it is through its interface that he is able to specify a CSPL formula that may be later evaluated. The user could of course start composing the syntactic tree of the formula manually by himself, but that would make tests hard to read and maintain. Therefore, the *Experiment* component allows the test specification to be written in any arbitrary format and then takes the responsibility to turn it into an `IFormula` instance.

In order to give the component as much freedom as possible, it is represented by only a single interface `IExperiment`, figure 3.8. It is apparent that anything with the ability to build an `IFormula` instance can be an implementation of the interface.

```
public interface IExperiment {
    IFormula BuildFormula();
}
```

Figure 3.8: The *IExperiment* interface

Inspired by concepts like *Fluent Interface* [17] and *Convention over Configuration* [18] UnitRacer provides simple implementation of the `IExperiment` interface that lets users specify a CSPL formula by a lambda expression. This has the advantage that the notation is very short and syntax errors are discovered during compilation because the compiler checks the syntax of such code.

To illustrate a typical test specification based on a lambda expression, it is best to start with an example.

Example 9. *Let* `quickFunctor` *be an* `IFunctor` *instance that represent the sorting method on some instance of the Quick Sort algorithm class. Similarly, let* `mergeFunctor` *be a functor for the Merge Sort algorithm. In order to capture the assumption that the mean execution time of the Quick Sort algorithm is at least* 10% *lower than the mean execution time of the Merge Sort algorithm on arrays provided by generator* `generator`*, it is necessary to write the following lines:*

```
// functors
IFunctor quickFunctor = ... ;
IFunctor mergeFunctor = ... ;

// generator
IArgumentGenerator generator = ... ;

// experiment
Experiment experiment = new Experiment((quick, merge, mean) => mean(0.9 * ↩
    quick) < mean(merge));
experiment.Methods["quick"] = new MethodOperand(quickFunctor, generator);
experiment.Methods["merge"] = new MethodOperand(mergeFunctor, generator);
experiment.SamplingStrategy = new ExplicitSamplingStrategy(50, 400);
```

The `Experiment` class is the `IExperiment` implementation that accepts lambda expressions. It has been called this way to emphasize that it is the preferred way to specify formulas. The lambda expression starts with parameter declaration. It is necessary to create a token, such as `quick` and `merge`, for each functor and a token, such as `mean`, for each tested characteristic that will be used in the formula. Then it is possible to write the body of the lambda expression in a very natural manner. Performance observation transformation functions are applied directly to functors and characteristics have a form of a function that takes the transformed functor as an argument. From these elements even complex formulas with many logical operators may be built.

After expressing the formula with a lambda expression, it is still necessary to connect tokens from that expression with instances like functors, argument generators, characteristics and sampling strategies.

Functor and input argument generator form together a `MethodOperand` that is assigned to functor tokens from the expression. Similarly, a characteristic may be assigned to a characteristic token. In the example 9 this was not necessary since the token named `mean` is automatically interpreted as a mean characteristic of a random variable by convention. The last thing to be specified is an optional sampling strategy. In the example, a strategy has been used that skips the first 50 measurements and then collects the next 400. But the exact behavior may be changed by using a different `ISamplingStrategy` implementation.

Once the user has an `IExperiment` instance ready, all that is left for him to do is to pass this instance to the *Evaluation Engine* component. Therefore, the phase of capturing the formula accounts for most of the work necessary on a performance unit test with the UnitRacer library.

Before proceeding to a section that analyzes the *Evaluation Engine* component, it is still left to demonstrate what are the possibilities of different implementations of the `IExperiment` interface. First of all, the ability to write lambda expressions is not present in most of today's mainstream programming languages. Therefore, when porting UnitRacer to a different environment, another implementation of the *Experiment* component is necessary. The most straight-forward idea is to capture formulas by a string. The string can be parsed in a similar way like the lambda expression, only the compile-time validation is missing. Another way could be to let users write test specification with XML files or some custom scripting language that could be used on continuous integration servers in a

similar way that the *NAnt* [19] tool is used in *CruiseControl.NET* [20]. Finally, the article [7] proposes that attributes could be used to annotate data types in the Java programming language and thus the performance unit testing would be brought closer to the code it is aimed to test. In UnitRacer similar approach could be simulated by an attribute-parsing component, which would need to implement the `IExperiment` interface.

3.5 Evaluation Engine

The component described in this section is the essence of the library. Although user's interaction with it is usually expressed with just one line of code, it is the center of his interest. The *Evaluation Engine* component is focused on evaluating `IFormula` instances, as provided by `IExperiment`. The component manages a set of statistical tests, looks up unevaluated nodes in a syntactic tree of a formula and ensures that the reporting component receives correct input.

While the biggest benefit of the *Experiment* component is that there is an unlimited amount of ways in which formulas may be specified by a user, the *Evaluation Engine* component is not expected to be replaced by some different implementation. Although it is again encapsulated by an interface `IEngine`, its task is so well defined and structured that there is only little room for variations.

The `StandardEngine` implementation of the interface starts with an `IExperiment` instance provided by a user. The input-dependent instance is immediately transformed into an `IFormula` by using the `IExperiment.BuildFormula()` method. This leaves the engine with a syntactic tree that has some arbitrary number of `Hypothesis` nodes whose values are set to *indeterminate*. In order to compute the value of the root node, a certain subset of the hypothesis nodes needs to have their values assigned. Figure 3.9 shows that sometimes it is necessary to evaluate only a proper subset of formula's hypothesis elements in order to determine the value of the root node. However, this approach would leave some hypothesis nodes undecided and therefore the user would not be able to inspect their evaluation data with the ReportInspector tool. Additionally, it is expected for practical CSPL formulas to have only a small number of nodes, since the best practices for (performance) unit tests advise, that only a single fact should be tested at a time [21]. For these reasons, the `StandardEngine` evaluates the truth value of each single hypothesis node in the formula.

3.5.1 Tests

Each `IEngine` implementation has its own collection of statistical tests that are used to evaluate individual hypothesis nodes. The collection is accessible to the consumer of the component and may be altered by registering new tests or removing old ones. The collection is ordered so that the tests may be prioritized. When the engine encounters a hypothesis node whose value is not decided yet, it looks into this collection of tests and tries tests in order until one that understands the particular hypothesis is found. When no such test exists, the library throws an exception so that the user can rewrite the formula or register some missing test.

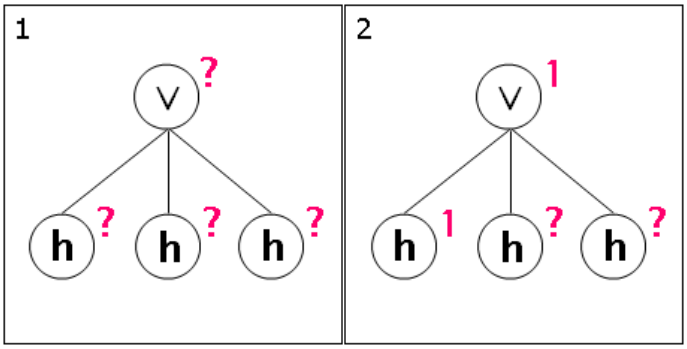


Figure 3.9: Formula that has logical or ($\vee$) as its root node and three hypothesis nodes as its children. Evaluating a single child to true is sufficient to decide the value of the formula.

A statistical test is represented in UnitRacer by the following interface, figure 3.10.

```
public interface ITest {
    double SignificanceLevel { get; set; }
    bool CanDecide(Hypothesis hypothesis);
    bool NotRejects(Hypothesis hypothesis);
}
```

Figure 3.10: The *ITest* interface (without events).

Since the goal of the library is to demonstrate the theory from the first two chapters, the default implementation of `IEvaluationEngine` comes with two ready-to-use tests. These are the Welch's t-test and the Student's t-test, both with support for one-tailed as well as two-tailed hypotheses. Their implementation of `bool CanDecide(Hypothesis)` method only checks the form of the hypothesis. This means that for the Student's t-test it has to be a hypothesis where one term is a `FunctorTerm` with `Mean` characteristic, while the other term is a `ConstantTerm` and all relation operators $\leq$, $\geq$ and $=$ are allowed. For the Welch's t-test both terms have to be `FunctorTerm` instances with `Mean` characteristics, relation operators are again unconstrained.

3.5.2 Outlier Filtering

Before handing over a sample of measured method execution times to a statistical test, it is beneficial to analyze the sample first and identify the so called *outliers*. An outlier is, according to [22], "…an observation which deviates so much from the other observations as to arouse suspicions that it was generated by a different mechanism…". When working with real-world data, measurements often suffer from corruption that may affect how the data is interpreted by statistical tests. The corruption can be caused by several factors, like imperfect measuring tools, their malfunction or human error. While the human error is not relevant for the UnitRacer library since it is a computer program, some percentage of wrong data

is still to be expected ([23], [24], [25]). These anomalies may be caused by a number of reasons. For illustration, one of them is the fact that the measured method does not run in isolation, but is a part of a complex computing environment that is composed from an operating system and other running applications that all together share a single hardware.

There are several techniques of outlier detection. A concise overview may be found in, e.g., [26]. In general, they may be divided into *parametric* and *non-parametric* groups. Techniques from the first group assume a known distribution of observations or, alternatively, they use statistical estimates of distribution parameters that are otherwise unknown. An outlier is flagged by such techniques, when it deviates significantly from the assumed model. The techniques from the second group are on the other hand useful in situations where there is no prior knowledge of the distribution. Particular algorithms choose different approaches but a popular one is so called *distance-based* since an outlier is flagged if its distance from the rest of the sample is bigger than some measure.

It depends on a particular scenario whether an outlier detection technique is better than some other one. Because essential part of UnitRacer is statistical testing, the library comes with one such algorithm pre-implemented. It is a distance-based outlier detection algorithm, described by authors Knorr and Ng in [27]. It fits normally distributed random variables, but is very robust and may be easily adapted to other distributions as well. Because both the Welch's t-test and the Student's t-test are designated for normal random variables, their implementations in the library use this algorithm in order to clean their data.

The Knorr-Ng algorithm defines outliers in [27] as follows:

> An object O in a dataset T is a UO(p,d)-outlier if at least fraction p of the objects in T are $\geq$ distance D from O.

For random variables with normal distribution, the notion of outliers is given a more formal definition. In [28] an outlier for such distribution is established as a point that lies three or more standard deviations from the mean. The [27] continues and provides the following formulas:

> Let T be a set of values that is truly normally distributed with mean μ and standard deviation σ. Define Def_{Normal} as follows: $t \in T$ is an outlier iff $\frac{t-\mu}{\sigma} \geq 3$ or $\frac{t-\mu}{\sigma} \leq -3$.
>
> $UO(p, D)$ unifies Def_{Normal} with $p_0 = 0.9988$, $D_0 = 0.13\sigma$, i.e., t is an outlier according to Def_{Normal} iff t is a $UO(0.9988, 0.13\sigma)$-outlier.

UnitRacer provides implementation of this algorithm in `KnorrNgOutlierFilter` class. The values of parameters p and D are freely configurable. When applied to a sample of method execution times, the algorithm result is satisfying in respect to filtered outliers. Figures 3.11 and 3.12 show a sample before and after outlier filtering. The sample has been taken from evaluation report of an ordinary performance unit test written with UnitRacer. Details about the test are left out as its purpose here is solely to graphically demonstrate the effect of the Knorr-Ng algorithm.

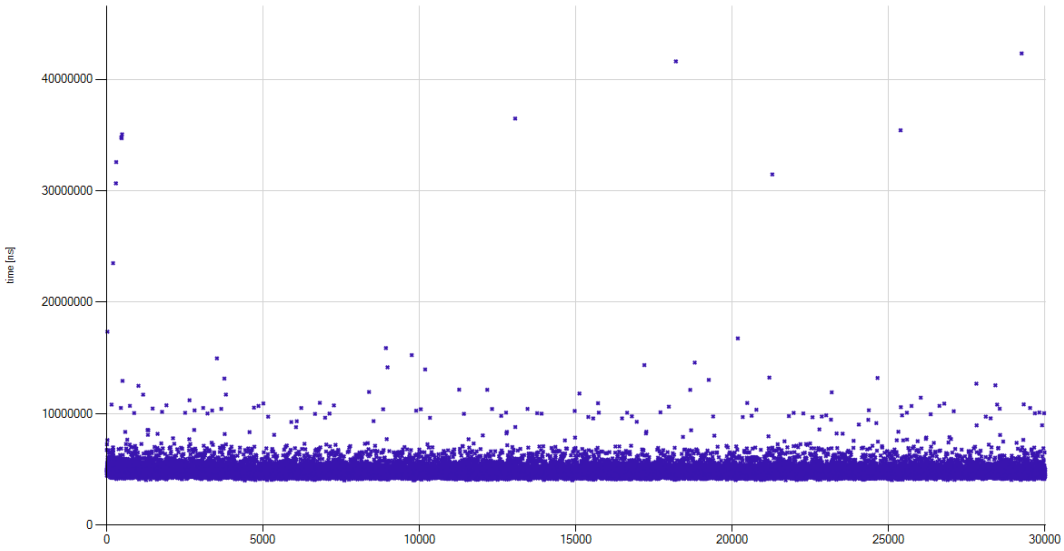


Figure 3.11: Measured method execution times *before* outlier filtering.

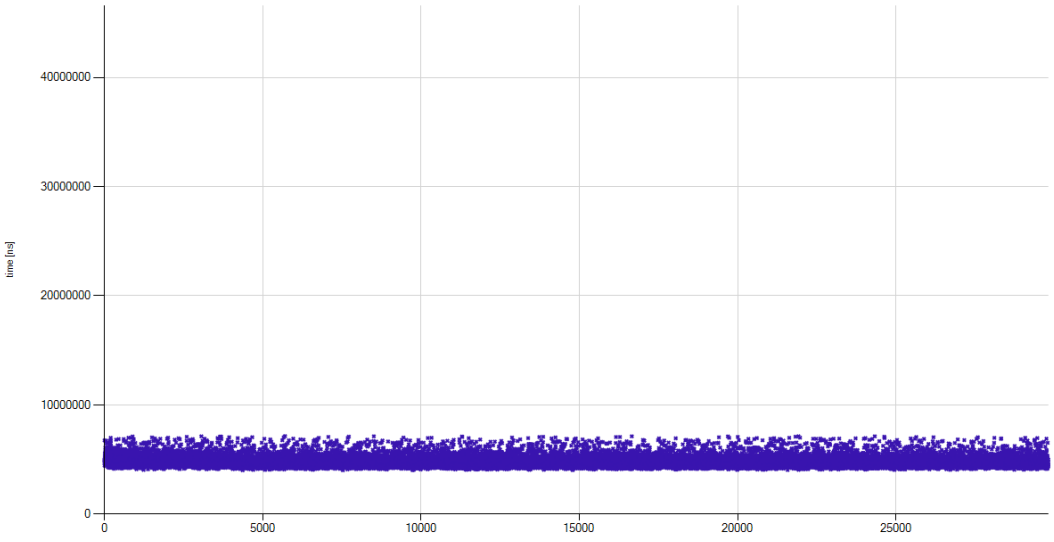


Figure 3.12: Measured method execution times *after* outlier filtering.

3.6 Reporting

Up to this point, the thesis has introduced and described three essential components that are together sufficient to perform performance unit testing. The process starts with creating a test specification based on information provided by the *Experiment* component. The next step is the input-independent formula representation and the *Grammar* component. Finally, the formula may be passed to *Evaluation Engine*, which returns *true* or *false* and thus finishes the test. Therefore, it is obvious that the fourth component called *Reporting* is somehow additional. And from the architectural perspective, it really is. The component connects to the evaluation process and observes its progress. Based on collected data a report is created. It allows the user to inspect it in detail and, using the ReportInspector tool, see its graphical representation. This is a way for the user, how to receive some better feedback about the test and possibly to do some rewriting and fine-tuning as a reaction on the report.

The component connects to the rest of the library with two interfaces – `IReportGenerator` and `IReportBuilder`. The first one is designed to be plugged into the *Evaluation Engine* component and decides the form reports will have. Whenever a formula is to be evaluated by the engine, `IReportGenerator` instantiates a new `IReportBuilder`, which subsequently takes it upon itself to create a report for the evaluation. Thus there is always a single report builder designated to service one formula evaluation. This helps to keep the design simple since no complex life cycle management is necessary as `IReportBuilder` is discarded as soon as the evaluation of its associated formula is finished.

UnitRacer ships with `XmlReportGenerator` that creates XML report files. These may be opened by the ReportInspector tool and have all the advantages of the XML format. The number of possible `IReportGenerator` implementations is not limited and so `IEngine` instances will accept any arbitrary one. For reporting over a REST service, JSON format could be used. Other ideas for future development could be to create a report generator that would log reports into a relational database.

When an evaluation engine is to evaluate a formula, it asks its report generator for a new instance of the `IReportBuilder` interface. This builder is then given the formula so that it can perform some initial configuration. Usually, the builder will register itself to events exposed by formula nodes. The `Hypothesis` class has an event that is fired each time its value gets changed and the `ISampleSource` has an event that is fired when a new sample with method execution times is collected. These data, together with the structure of the formula, is everything that the `IReportBuilder` needs to know in order to create a report. In addition, because the process of collecting data is event-driven, the report builders can output reports on the fly without having to store large data structures in memory.

This section has concluded description of the main UnitRacer components. The thesis will now focus on the ReportInspector application and on a worked example of the library in the rest of this chapter.

3.7 ReportInspector

The ReportInspector is a .NET executable that builds on top of a graphical framework in order to deliver a rich user interface for report inspection. Sometimes, it is easy to write performance unit tests with CSPL formulas, because they are either given by specification or they are general enough so that they may be written without a thorough analysis. This first case is illustrated by example 3, where the specification exactly prescribes scheduling times for a process planning algorithm. The second case may fit a situation where it is simply required that some approach A is faster than some other approach B – programmer just writes this simple assumption without knowledge of the precise relation between the two approaches.

On the other hand, sometimes it is difficult to capture performance assumptions, because a test designer has no prior knowledge of the tested units. He just wants to protect the code from breaking changes that would damage performance of the application. In that case, he may either compute the expected relation between the units, or he may use an empirical approach of learning the relation by tests. Computing the relation is usually suitable for comparison of algorithms since their asymptotic notation – if known – may be used as a basis. In contrast, relation for units whose performance is bound to hardware facilities is often impossible to determine in advance. An example of such scenario may be comparison of two bitmap rendering strategies, as proposed by [7, p. 2]. In this case one could simply compare the units using a straight-forward approach and write the formula as, for example, $mean(A) < mean(B)$. This captures the relationship only very vaguely, however. In order to make the formula more specific, it is necessary to observe the units first. And that is exactly where ReportInspector provides its advantages.

The application is targeted, similarly as the core UnitRacer library, at .NET Framework 4 and higher. It is fully portable to the Mono platform. Its user interface is based on the WinForms framework and the *ZedGraph* library, which is used for chart rendering. ReportInspector needs no special installation, it is distributed together with other UnitRacer assemblies and may be launched directly from the distribution folder.

When the ReportInspector is started, it displays no report by default. To display one, a XML report has to be opened from the *File* → *Open* menu or dragged onto the main form. The report is expected to be in the exact format as produced by `XmlReportGenerator` class described in the previous section. It is checked by the inspector first for its well-formedness and then also for its validity against the XML schema provided by the generator. If it succeeds, the report is finally displayed to the user.

There are three main views offered and each of those analyzes the evaluation from a different perspective. The first one is called *Formula view* and shows the syntactical structure of the analyzed formula with all its important properties. The second view is *Histogram view* and for `FunctorTerm` nodes of the formula, it displays histograms of their measured method execution times. The last view offered is *Timeline view* and shows the evolution of measured method execution

times during the sampling process. These views are described further in the text in detail.

3.7.1 Formula View

The formula view displays the syntactical structure of the formula whose evaluation report is currently open. For illustration purposes a sample screenshot is shown in figure 3.13.

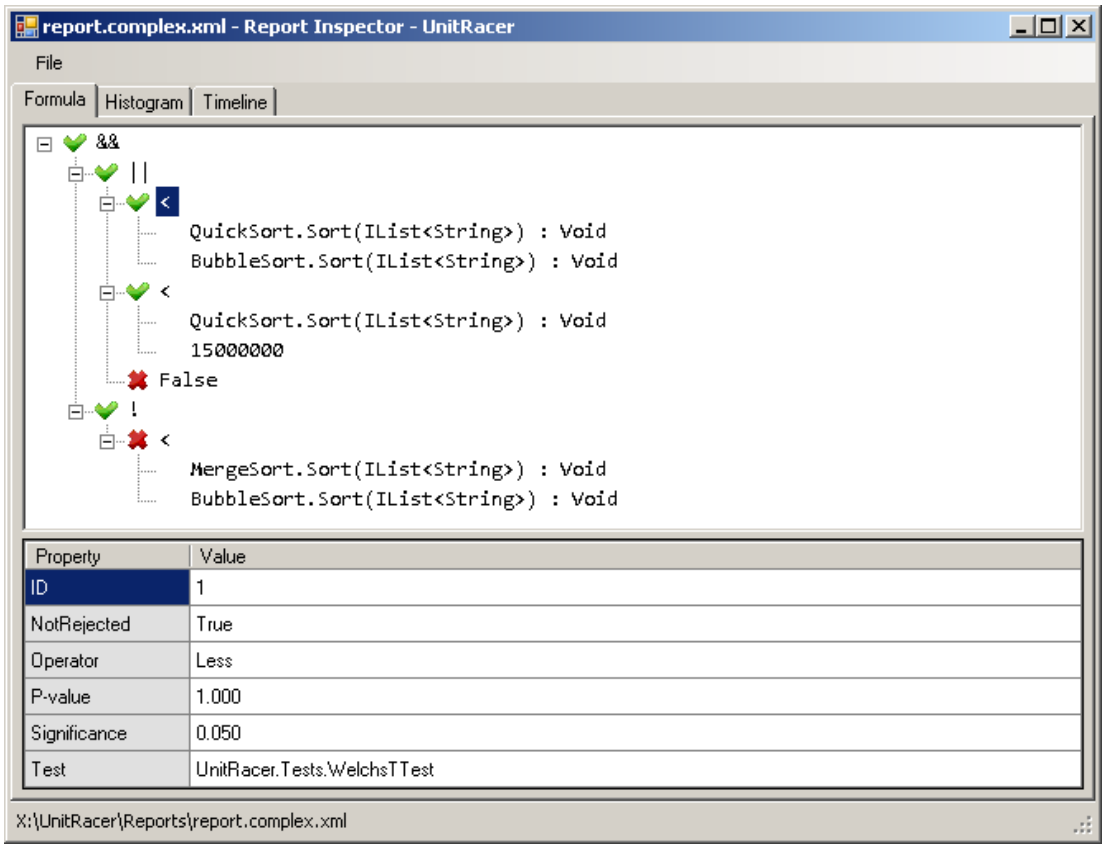


Figure 3.13: Formula view.

The upper part of the view is the syntactical tree structure, while the bottom part displays properties of the currently selected node. Nodes from the *Grammar* component that implement the `IFormula` interface have all property `Value` that is visible in the property panel as well as an icon next to the node in the tree structure. This helps the user to quickly identify, which nodes have failed during the evaluation and therefore caused the whole formula to be *false*. The `CompositeFormula` nodes are displayed with text either "&&" or "||" based on the value of their `Operator` property. The `NegatedFormula` is displayed simply as an exclamation mark and the `ConstantAtom` as "True" or "False".

On the other hand, the `Hypothesis` nodes are more complex. Their icon marks, whether they have been rejected by their statistical tests or not. Text of the node is the relation operator applied between their two `ITerm` child nodes. The property panel displays the name of the `ITest` that has been resolved by UnitRacer to evaluate the hypothesis. Other properties are the p-value based on

which the test has either rejected or not rejected the hypothesis and the ID under which the XML report identifies the node.

The `ITerm` nodes are either for `ConstantTerm` instances that represent an absolute time bound in nanoseconds, or `FunctorTerm` instances that serve as an abstraction of a tested unit. The `FunctorTerm` nodes are named with a signature of the method they represent and the property panel displays what transformation and what argument generator has been used for their sampling.

3.7.2 Histogram View

The second view renders histograms based on the data from the opened XML report file. Figure 3.14 shows a sample screenshot.

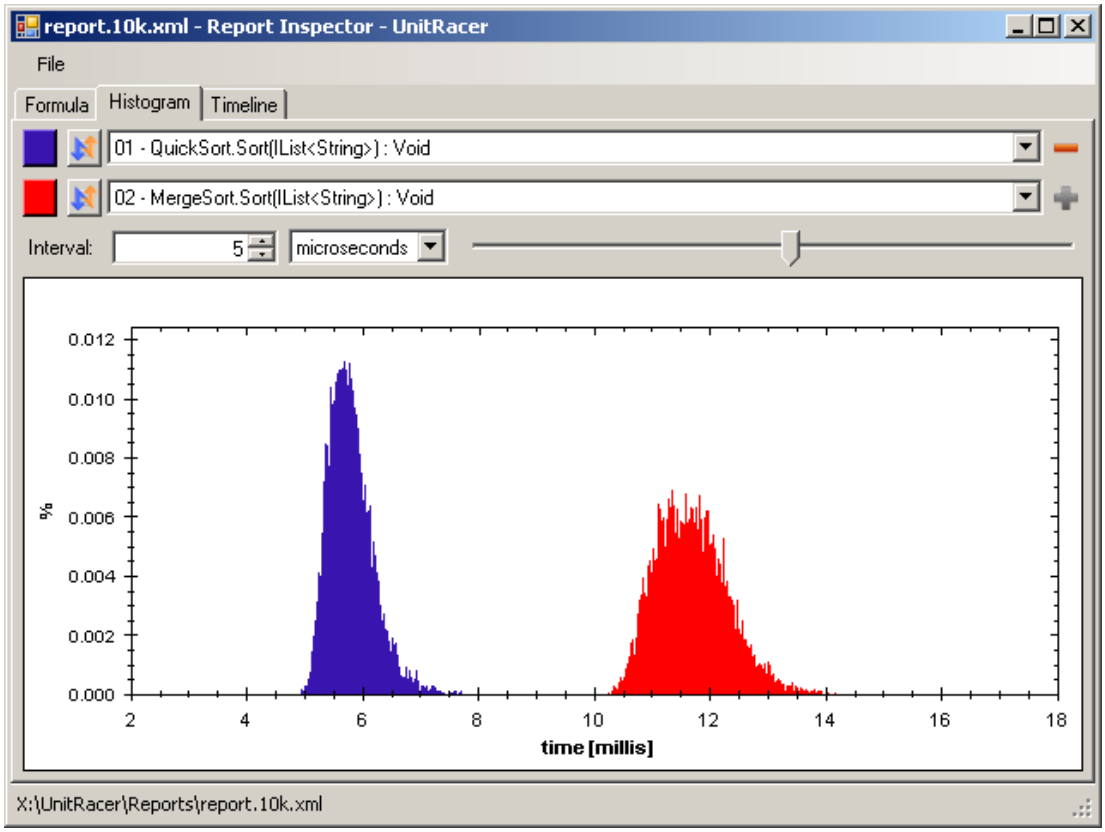


Figure 3.14: Histogram view.

In the upper area, the user can select for which `FunctorTerm` nodes the histograms should be rendered. The bottom chart area then displays his selection. There may be several histograms displayed at once and so it is possible to change their color. User can also choose whether the chart should display histograms based on raw method execution times, or whether the transformation function should be applied. This is useful when the histogram does not look as expected and the user wants to verify, whether this deviation has been caused only by the transformation or not.

An important parameter of each histogram chart is the interval over which the data frequencies should be counted. In ReportInspector, this parameter is

configurable either by using a numeric field and a combo box with time order (nanoseconds, milliseconds, …) or by using a logarithmic slider that spans over the entire meaningful data range.

The histogram view is useful when inspecting distributions of method functors. Some statistical tests are non-parametric and thus do not depend on any particular distribution of the sampled data. On the other hand, tests like the Welch's t-test and Student's t-test fit only data that comes from a normally distributed random variable. It is programmer's risk if he decides to ignore this requirement and use the tests for functors whose execution times do not have a normal distribution.

Sometimes, when a performance unit test does not yield the expected result, the histogram view may be useful, because it shows whether it has been caused by some external factor. When the histogram shows, e.g., a peak that is far away from the majority of data there might has been some measuring error caused by interference with other running applications. The same hint may be also obtained from the timeline view, where the interference factor is made yet more obvious.

3.7.3 Timeline View

The last view displays measured execution time values in the exact order in which they have been collected. This allows the user to see how the performance evolved over the time. Sample screenshot is in figure 3.15.

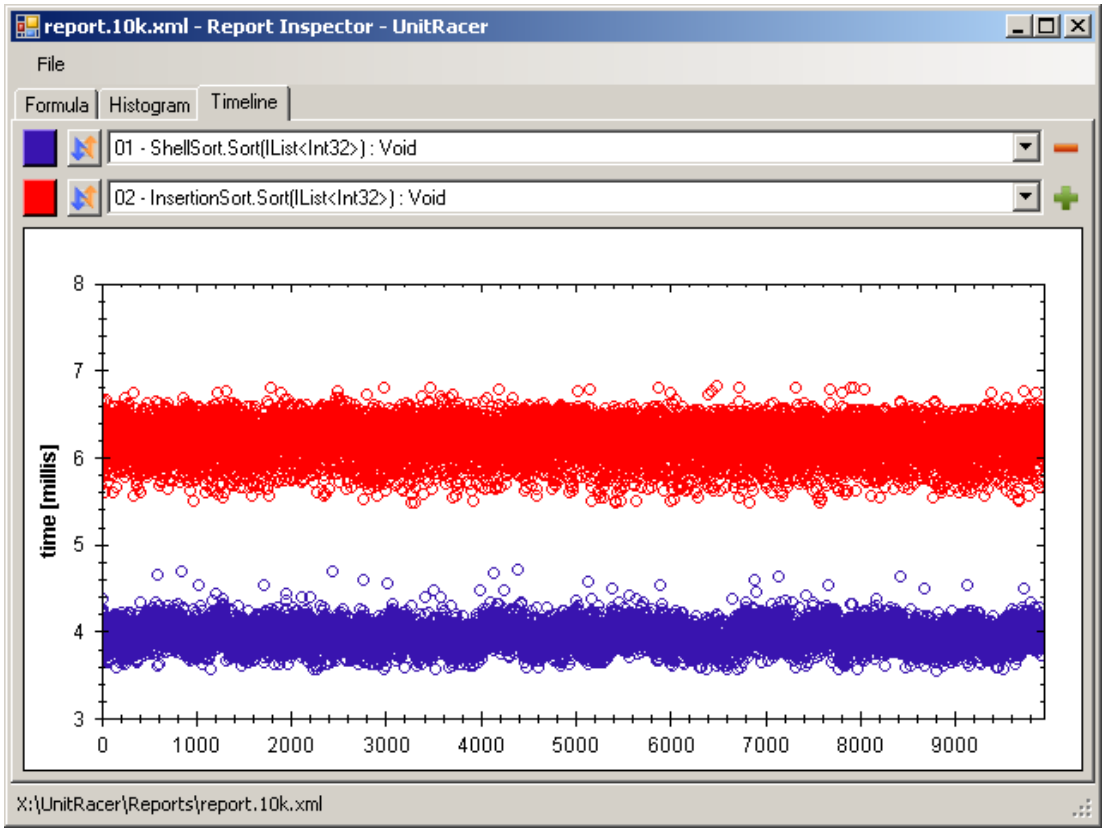


Figure 3.15: Timeline view.

The main benefit of this view is that it clearly shows any short-term measuring errors caused by interference with other running processes. When UnitRacer is measuring execution times for tested functors and some CPU demanding application starts computing in the middle of this process, the execution times of the tested functor jump immediately up.

3.8 Worked Example

Before moving to the last chapter that describes several ways of how UnitRacer can be used in order to improve quality of software projects, this last section shows a worked example that puts all the previous theory together.

The example is inspired by section 6 of the article [7] and will compare two sorting algorithms. The first sorting algorithm will be the Insertion Sort and the other one will be the Bubble Sort algorithm. These units are suitable for performance unit testing with UnitRacer, because the performance of both may be reasonably described through their mean execution time.

The performance assumption captured by the test should verify that the Insertion Sort is faster than the Bubble Sort when sorting integer arrays with one thousand elements.

Figure 3.16 shows the code that implements the test. It can be found on the attached DVD as a part of a Visual Studio project that is made available so that the example may be reproduced. The code also serves as a template that may be used in real projects to create custom performance unit tests with UnitRacer. The structure of the test is further analyzed.

The first phase of each unit test is the *arrangement* phase. It is necessary to obtain functors for both units, lines 7 and 8. Next, an input argument generator has to be prepared for the test. Because the tested algorithms expect a single argument – an integer array – the generator is configured as shown on lines 12 - 17. The `ArrayComposer` class creates arrays using a specified delegate for generation of individual elements. Based on specification of the test, the `ArayComposer` used in the code sample has been ordered to generate arrays of one thousand random integers. The last class, `CompositeGenerator`, is used to put multiple input arguments together for a tested unit. Since both the tested functors expect only a single argument, the composite generator only wraps the array composer without adding any additional arguments. Finally, the experiment is specified using a lambda expression that resembles CSPL formula, line 20. Parameters of the lambda expression are bound to functors and argument generators, lines 21 and 22. Line 23 orders that the first 500 measured values should be discarded to only warm up the system and then the next 10000 values should be collected for evaluation purposes. This finishes the arrangement phase.

The next phase is *action*. In the context of UnitRacer, this means simply to evaluate the test, line 27. Because here in the sample only the most basic functionalities are used, the static `Engine` class is used. It offers `XmlReportGenerator` which is configured to output the evaluation report that will be further analyzed, line 26.

Final phase is *asserion*, with Visual Studio Unit Testing Framework this results in the code on line 30.

```
[TestMethod]
public void InsertionSort_FasterThan_BubbleSort()
{
    // arrangement

    // .. functors
    IFunctor insertionSort = FunctorFactory.Create(new InsertionSort(), ↩
        insertion => insertion.Sort(Arg.Of<IList<int>>()));
    IFunctor bubbleSort = FunctorFactory.Create(new BubbleSort(), bubble => ↩
        bubble.Sort(Arg.Of<IList<int>>()));


    // .. generators
    Random rand = new Random();
    ArrayComposer<int> arrayComposer = new ArrayComposer<int>(1000, () => rand↩
        .Next());
    IArgumentGenerator generator = new CompositeGenerator()
    {
        () => arrayComposer.GetNext()
    };

    // .. experiment
    Experiment experiment = new Experiment((insertion, bubble, mean) => mean(↩
        insertion) < mean(bubble));
    experiment.Methods["insertion"] = new MethodOperand(insertionSort, ↩
        generator);
    experiment.Methods["bubble"] = new MethodOperand(bubbleSort, generator);
    experiment.SamplingStrategy = new ExplicitSamplingStrategy(5000, 10000);

    // action
    Engine.ReportGenerator.OutputPath = "report.xml";
    bool result = Engine.Evaluate(experiment, true);

    // assertion
    Assert.IsTrue(result);
}
```

Figure 3.16: Test method representing the example performance unit test.

Whether the test fails or succeeds depends mainly on the measured execution times. To put the UnitRacer library into a bigger perspective, the test from figure 3.16 has been evaluated in two different environments – $[\mathcal{A}]$ and $[\mathcal{B}]$.[4] The test has succeeded on both configurations with p-value being 1.000. Figure 3.17 offers a comparison of execution time histograms. Histograms are labeled with marks in format $[X - Y]$, where X stands for environment and Y is either I for Insertion Sort or B for Bubble Sort.

The presented example has shown that changing platforms does not break the performance unit test. This upholds the goal of UnitRacer to provide reliable testing framework that does not bind the tests to a particular hardware or software configuration.

[4]The hardware and software attributes of the environments are described in detail in appendix Test Machines.

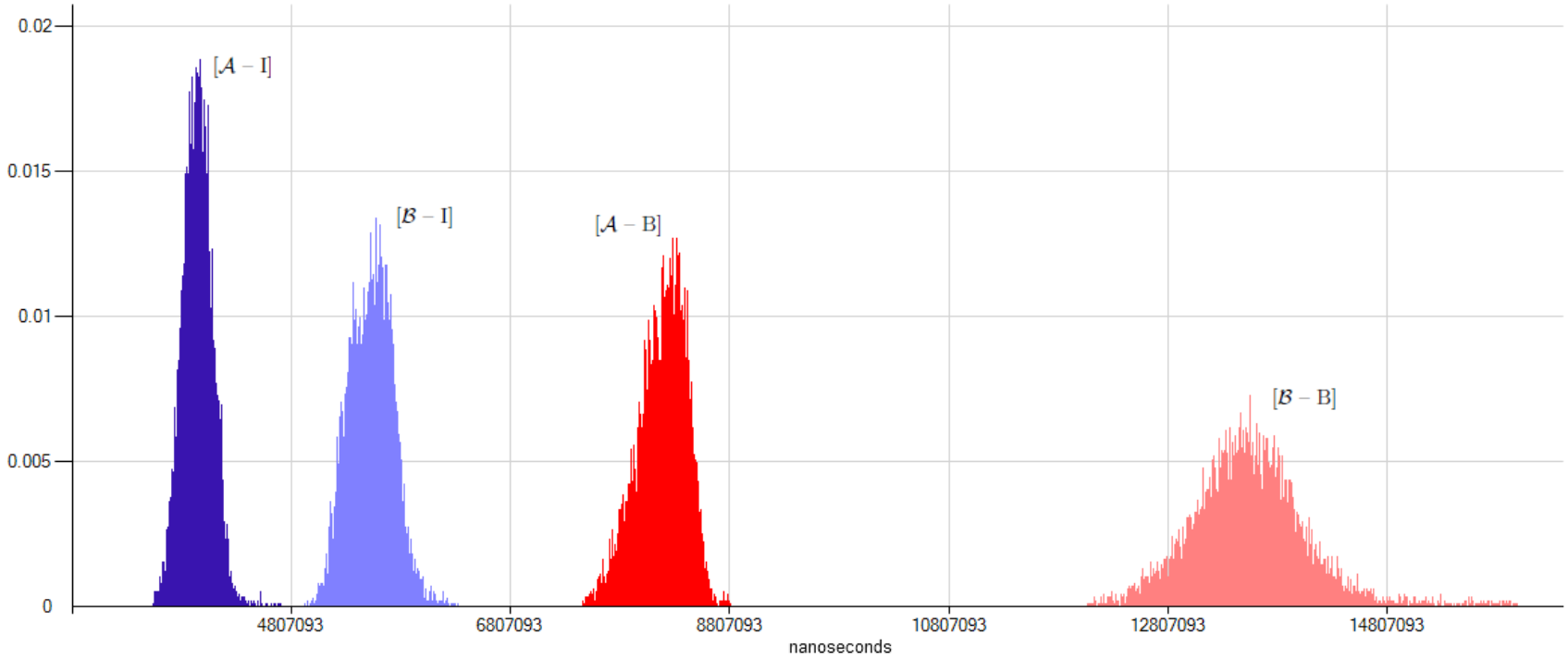


Figure 3.17: Histograms of executions times for the worked example.

Chapter 4

Use Cases

Having the working library in place, it is now time to discuss its practical use and benefits it offers to everyday software projects. This chapter has one section for each possible UnitRacer application that has been identified and supported with a sample use case. Of course, the UnitRacer library could be replaced with some other relative performance testing framework and the range of possible applications would remain the same. The use cases presented here have been, however, carried out with UnitRacer to support the practical usability of this thesis in particular.

In the introduction of the thesis performance unit testing has been compared to classical functional unit testing. Functional testing has many positive attributes that make it popular today. Specification of individual tests comes directly from the expected behavior of some unit. Either it is clearly stated in the specification of the program or it emerges as programmer's natural expectation. Either way, the functional unit test verifies that when given some particular input the tested unit will change its state and produce some output exactly as expected by the specification. The functional unit tests are usually quick to evaluate and highly reliable because of their strict determinism.

It has been also shown that performance unit testing has to deal with many problems that functional unit testing does not. It is hard to define exactly what is fast and what is slow. Tests may also fail when moved to a different hardware configuration. And even if the hardware configuration stays the same the tested units may have different performance each time they are tested because of the impact that the overall environment imposes on computing resources available for the unit. These disadvantages are believed to be reason why performance unit testing is often completely skipped in common software projects.

The relative performance unit testing based on a special logic, as proposed by [7] and implemented in this thesis by the UnitRacer library, aimed to solve these problems and offer an easy way for developers to integrate this kind of testing to their projects. Because of the comprehensible formulas it should be viable for programmers to express their performance expectations in a relative way. When talking about performance it is believed that instead of guessing absolute time bounds, it is more natural to compare the tested execution time to some familiar unit that is known to be fast or slow. The relative comparison is also the key for platform independence of unit tests written with CSPL formulas.

Finally, UnitRacer makes the tests more reliable as it uses statistics to evaluate the formulas.

In the following sections, several practical scenarios are introduced that benefit greatly from employment of performance unit tests. The first one, section 4.1, discusses how UnitRacer may be used to protect important performance invariants throughout the whole development. Next, section 4.2 shows UnitRacer as a regression prevention mechanism and also how it can be used to increase quality of distributed team projects. Section 4.3 uses the library to increase usability of projects by keeping them close to their competition or to their already existing variants.

4.1 Guarding Invariants

In order to achieve the best possible performance an application has to rely on a certain performance ordering among the units that it has decided to use or to reject. The ordering is a natural way to tell which units should be used in the application. If two units are otherwise equivalent, the faster one will be usually preferred over the slower one.

Let's start at the very beginning of the development process of some project. The first thing that has to be decided before actually writing any code is the architecture. Architectural decisions are very important, because they decide application attributes at the highest level. When a software architect makes a bad decision, it is usually very hard to change it later during the development as the cost of all associated rework is increasingly high (as described in [30]). When an application aims for a good performance, this aspect should be carefully thought through during the architecture phase and every decision to use some approach A over some approach B should be supported with research and documented. This clarifies the decision for developers that join the team later. Performance comparison of certain approaches may be already known and reported by some authorities, but some testing may be necessary even in spite of this fact.

Choices between architectural approaches are usually not interchangeable one with another. The decision may be to use the Java programming language instead of the `C#` language. There is no easy way to switch between them once some code has been already written. On the other hand, some architectural decisions may be between two approaches that are interchangeable but not equivalent, i.e., in constraints they place on the resulting application. An example of this decision may be whether to use in a project for the .NET platform the functionality to emit compiled code at runtime or whether to use reflection. Because emission is not available in the Compact edition of the .NET platform (aimed at hand-held devices) the use of the application could be restricted more than the specification allows. When the support for the Compact edition is not an issue, the reflection based and emission based approaches may be compared together and the faster one selected. The best way to document this is perhaps to write about the decision into the architecture documentation and make the performance test that compares the two approaches part of the project.

With architecture in place, the developer still faces many decisions, where he has to choose between two implementations of the same functionality. In well-

designed application these would be typically hidden behind a common interface. The interface ensures that the units are interchangeable and the decision which one to use is therefore merely a matter of configuration of an *IoC* container ([31]). As with any such decision, it is best documented by a performance unit test that is attached to the project and executed periodically.

The article [7] states "…even when the performance of both methods changes with the underlying platform, the relation between the two should hold and if it does not, it is certainly worth developer's attention…". This is cerainly true for tests described in this section so far. When a test guarding an architectural decision gets broken on some new platform it is an alert for the development team. If it is possible to change the decision and use the faster approach on that particular platform, the build process can be customized and the application can be composed from different components for different platforms. This approach is used for example by many .NET projects to use code emission on the full .NET platform and ship with reflection based approaches for the Compact edition – popular open source projects like *Ninject* and *Json.NET* (both used in this chapter as reference examples) are aware of this fact and offer customized builds for each framework edition.

This leads to another important question – how should the testing of important performance decisions be tested. Unit tests are usually executed on both the developer's machine and the continuous integration server. For performance tests described so far, this may not be the case as the code of units they test should not be modified by any programmer during the development. They are established at the beginning as invariants on which the whole project is based. However, such tests may be broken with a platform change or software update. The publication [2] argues this when describing conditions under which one should reconsider his optimization approaches. Therefore, the collection of tests that guard performance invariants should be executed for each intended platform on which the application should be available and also again for each important update or new version of the software on which the application relies.

The section will now proceed to a practical use case that demonstrates how such important architectural decision may be tested and documented with UnitRacer performance unit tests.

4.1.1 Ninject

To show real-world applicability of the UnitRacer library, a popular open-source project has been chosen to show, how it could benefit from relative performance unit testing. The project is called *Ninject* [32] and it is a dependency injector library that helps to deliver inversion of control to .NET projects. It is fairly successful within the .NET community as it has, at the time of writing the thesis, more than $350,000$ downloads from a popular distribution channel [33] and it is even used in high print run literature for official Microsoft products, such as [34].

The concept of Ninject may be simplified to following – it uses a configuration of interface-to-implementation bindings to return the correct implementation to calling code based on an interface request. The problematic part is that Ninject needs to return an instance of the requested interface and therefore it has to be

somehow able to create it. But when the constructor of the class that provides the requested implementation defines some parameters, their values need to be resolved by Ninject as well. In addition, the library has to set any additional properties on the created objects that are marked with a special attribute `Inject`. Therefore, Ninject has to use the reflection facility of the .NET framework to be able to dynamically instantiate types and set values of their properties.

Because inversion of control containers are typically used throughout the entire application into which they are integrated, they may influence the performance of such application by an order of magnitude. The reflection facility is known to be slow, described for example in [35], hence any performance boost would be most welcome by Ninject and similar projects. Starting from .NET 2.0 there is an alternative way how to create a type's instance or dynamically assign a value to a property, it is called *Dynamic Method*. It could be an important performance-affecting decision whether to use this approach or not. Certainly one of those that have been described at the beginning of this section.

Because the source code for the Ninject project is freely available, it can be seen that the Dynamic Method approach is really used in all builds except those targeted at the Compact edition. This may be a reasonable decision as the performance gain of Dynamic Method over common reflection has been documented on official Microsoft web pages ([36]), but since then a yet another approach has emerged, called *Compiled Lambda*, and the performance relation could have also changed since release of a new .NET Framework version. Article [37] shows how the performance of reflection has shifted between .NET 1.1 and .NET 2.0.

For this reasons, it will be further demonstrated how UnitRacer can be used to fix this deficiency and to verify that the popular library has made the right choices.

4.1.2 Testing Reflection Approaches with UnitRacer

The functionality from the Ninject library that is the subject of test here is encapsulated by an interface. This is very convenient for testing purposes as all the units that will be compared together expect the same input and are freely interchangeable from the functional point of view. The interface is called `IInjectorFactory` and its excerpt relevant for this text is shown in figure 4.1. The `ConstructorInjector` class returned by the method is nothing but a simple delegate that represents a constructor call – it accepts arguments and returns the newly created instance. Apart from the method that is shown in the figure, the interface has two more methods that provide similar functionality, only for invoking methods and settings values to object properties. Those two methods will not be tested here as the one from the figure is sufficient for demonstration purposes.

```
public interface IInjectorFactory {
    ConstructorInjector Create(ConstructorInfo constructor);
}
```

Figure 4.1: Excerpt from Ninject's *IInjectorFactory* interface.

The three approaches available in the .NET Framework to create an instance of `ConstructorInjector` presented here are:

- **Reflection** – Uses the standard reflection facility to invoke a dynamic constructor and return the created instance. Available in all .NET Framework versions and editions.
- **Dynamic Method** – Based on emitting IL code during runtime. This dynamic code is then compiled using JIT into a static global method that no longer needs to rely on reflection. Available since .NET Framework 2.0, except for Compact and Micro editions.
- **Compiled Lambda** – More general approach than Dynamic Method. Programmer builds syntactic tree of the code and then lets the framework compile it. Available since .NET Framework 3.5, but features required for the `ConstructorInjector` appeared later in .NET Framework 4. Again, neither the Compact or Micro edition supports this feature.

The Ninject library has two implementations of the `IInjectorFactory` interface. The `ReflectionInjectorFactory` class and `DynamicMethodInjectorFactory`. Implementation based on the Compiled Lambda approach is missing, however it has been implemented as a part of this thesis for testing purposes so that all three approaches may be compared together using a UnitRacer test.

In order to capture performance assumptions about implementations of the `IInjectorFactory` interface properly, it is necessary to differentiate between three target environments for which the Ninject library may be shipped.

The first environment is the one that has only the Compact edition of the .NET Framework installed. In that case there is no room for the other two implementations and the `ReflectionInjectorFactory` has to be used.

For environments with .NET Framework 2.0, 3.0 or 3.5 two approaches are eligible – the reflection based and the dynamic method based. UnitRacer test depicted by the formula 4.1 could be used to assure that the faster one is shipped with Ninject builds for those frameworks.

$$mean(DynamicMethod) < mean(Reflection) \tag{4.1}$$

Finally, for .NET Framework 4.0 and higher all three implementations are available. This is also the case that will be demonstrated in further detail here. The formulas 4.1 and 4.2 capture the essence of the two tests that verify that Ninject ships with the correct implementation of the `IInjectorFactory` interface.

$$mean(DynamicMethod) < mean(CompiledLambda) \tag{4.2}$$

The tests have been implemented and are available on the DVD that accompanies the thesis. Their evaluation results from machine $[\mathcal{A}]$ together with p-values of the Welch's t-test that has been applied on the hypotheses are shown in table 4.1. Measured mean of execution times is compared graphically by a bar chart in figure 4.2.

The results show that Ninject is right not to ship with the reflection based injector factory for all frameworks that support also the dynamic method approach. On the other hand, the new compiled lambda achieves the best mean

Test	Success	p-value
mean(DynamicMethod) < mean(Reflection)	true	1.000
mean(DynamicMethod) < mean(CompiledLambda)	false	7.306e-040

Table 4.1: Evaluation results from machine [$\mathcal{A}$].

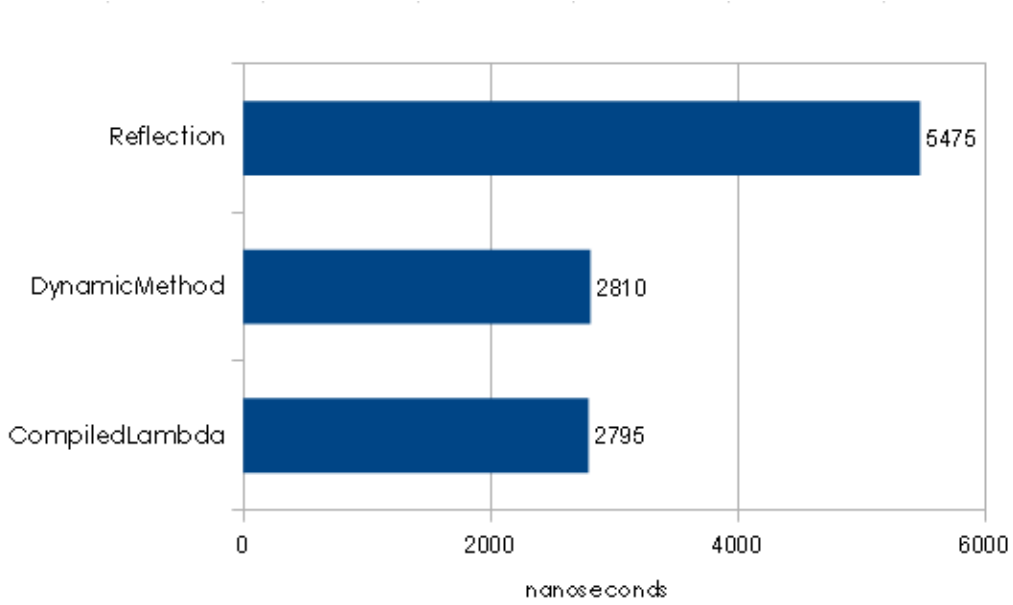


Figure 4.2: Comparison of mean execution times for different implementations of `IInjectorFactory.Create(ConstructorInfo)` as measured on machine [$\mathcal{A}$].

execution times of all the three tested implementations. The measured difference is only about 15 ns and so switching implementations would probably not have a big impact in production. But the fact itself is a hint, that creators of the library were not likely to do performance testing when .NET Framework 4.0 came out.

With these tests in place, the build process of the Ninject library could be altered to output three artifacts for three different platforms. For Compact editions, the library would be shipped with the reflection based approach. For .NET 2.0, 3.0 and 3.5 with the Dynamic Method implementations. And finally, for .NET 4.0 with Compiled Lambda.

This sample demonstration of advantages of relative performance testing has focused on a single unit from the entire Ninject library. There is of course a lot more units that could be tested this way either in Ninject or other software projects. The ideal situation is when there are several functionally equivalent units encapsulated by a common interface and switching between them imposes no restrisctions on the resulting application. Then, using a few tests, it can be quickly decided which one performs best and this fact can be taken into account when configuring the build process.

4.2 Regression Prevention

The previous section has demonstrated how a software project may benefit from using UnitRacer to guard important performance relations between architectural components. These components are usually not part of project's code base, they are more likely to be third-party components distributed as binary artifacts or competing features of the programming language in which the project is written. This is the reason why the performance relation between such components changes only with platform or with software update. The components that create building blocks of the entire application should be tested in advance and it should only happen rarely that some relative performance unit test that tests them gets broken.

Nevertheless, the biggest share of time spent on a software project is in the development phase ([38]), i.e., writing the code that is later compiled into a working application. During this time, there is no need to retest the architectural invariants, but a whole new scale of performance testing opportunities arises. This thesis proposes for performance unit tests to be written in the same manner as the standard functional unit tests.

The whole methodology is based on the TDD approach. In a classical scenario, when programmer is to write a unit, such as a class or a method, he should first write some appropriate amount of functional unit tests that will later verify his implementation. Next, he can do the programming and fill in body to the unit under development. The final step is running the test suite that has been created in advance. When every test is successful, the programmer may continue with his work. This is known as the Red-Green-Refactor process, described in more depth in [39].

With UnitRacer, the methodology originally designed for functional unit testing may be enhanced to include performance unit testing as well. Before implementing a unit, programmer writes a suite of functional unit tests and in addition also a suite of performance unit tests that will bind the performance of the unit to some other reference unit in a relative manner. This is relevant only in scenarios, where the performance of the tested unit is somehow important to the project. After implementation, the programmer executes both the functional and the relative performance tests to see if the tests are successful.

There are several benefits of writing performance unit tests for units that are developed as a part of the software project. It is verified that the application conforms to performance requirements established in the project's specification. Units are also protected from unintentional loss of their required performance attributes due to a change by some programmer. This is the so called regression which this section is focused on.

When a unit gets implemented, it may still be a subject of change. A bug may be found in its implementation and so its code needs to be rewritten. Or a programmer is just performing a refactoring aimed to increase the quality of the code. In either case, it can happen that the functionality and performance that has been once tested will be now broken. Because of that, it is important to keep the test suite and check the tests perhaps after each commit to version control

system. This will in return increase the quality of the project and prevent once tested unit to become broken.

4.2.1 Comparing Unit to Itself

Finding a reference unit that may be used in order to establish a relative performance bound for some target unit is sometimes an easy task. The reference unit may be given from project's specification or there is some other implementation of the unit already available that may be naturally used for such purposes. An example may be a scenario, where a programmer is asked to implement the Aho-Corasick string matching algorithm, [40]. Performance of this particular algorithm may be easily compared to a naïve algorithm that attempts to match the search string at each position in the input string. Therefore when somebody makes a change to the Aho-Corasick implementation and the upper performance bound gets broken, the team is notified about such event and the algorithm may be fixed.

However, it is expected that not all units that require performance testing have an easy-to-find reference counterpart. The thesis will now illustrate this on a detailed example. Let there be an application that uses query object pattern [41] to filter data in a domain-friendly way. The interface for such queries is shown in figure 4.3. The `Execute` method is given an interface that enumerates through all available domain entities and the result is a collection of those entities, pruned by a logic offered by a particular implementation of the `IQuery` interface. For a common task of looking up an entity by its ID, the application provides a `QueryByID` class that implements the interface in a way presented by figure 4.4.

Because the `QueryByID` is significant for the application from the performance point of view, the development team decides to write a relative performance unit test that will protect the `Execute` method from becoming any slower than it is in the presented implementation. It is therefore necessary to find a reference unit whose performance will form an upper bound. Although it is possible find or write such units, the best suitable one is the tested unit itself, because it fits best the test requirements – when the implementation is changed and the `Execute` method becomes slower than its initial implementation, the test will fail.

The routine may be inferred as follows. Whenever a unit with such requirements is created, it should be compiled into a custom assembly that will form a performance reference point. Next, a relative performance test is written with UnitRacer where the current version is compared to a version from the reference assembly. Finally, whenever even a minor change is introduced – for example when someone removes the `break` command from the code and the method then cycles through a half of the data source on average to no effect – the test will

```
public interface IQuery {
    IEnumerable<IEntity> Execute(IEnumerable<IEntity> datasource);
}
```

Figure 4.3: The *IQuery* interface.

```
public class QueryByID : IQuery {

    public int ID { get; set; }

    IEnumerable<IEntity> Execute(IEnumerable<IEntity> datasource) {
        IEntity result = null;

        foreach (IEntity entity in datasource) {
            if (entity.ID == ID) {
                result = entity;
                break;
            }
        }

        if (entity != null) {
            return new IEntity[] { entity };
        } else {
            return Enumerable.Empty<IEntity>();
        }
    }
}
```

Figure 4.4: The *QueryByID* class.

be sensitive to such deviation and report that assumed performance relation no longer holds.

An improvement of this routine could be a plugin for a continuous integrations server, because the server is already working with a source control management tools and could recover a specified base revision automatically and perform the test without the need for programmers to manually compile and store the performance reference assemblies.

4.2.2 Distributed Development

The process of software project development differs significantly based on whether it is a professional commercial project or an open-source collaborative effort.

Software companies typically invest in the development infrastructure and so each project has a server that provides it with continuous integration (CI). These servers ensure that each change of source codes committed to a version tracking system gets compiled and tested. If it cannot be compiled or some test fails, the breaker or the whole team is notified so that the issue is fixed. This mechanism prevents untested code to find a way into the code base of the application. As proposed earlier, the server providing continuous integration can not only execute functional tests, it is also a suitable place for performance unit tests. When a programmer introduces a new code that breaks a performance relation between some tested units, this fact is detected in the same way as failed functional tests are.

The situation is different with open-source collaborative projects that are developed by an incoherent, geographically distributed team. A great reference example is the currently popular website that hosts such projects – *GitHub* [42].

Anyone is free to come and start a project whose source codes are available to the public. Members of each project are divided into two groups - *project owners* and *common users*. While anyone can download the source codes, only project owners can update it. When a common user wants to make a contribution to a project, he needs to request some project owner to accept his code.

Since development of such projects is performed locally and then committed to the server that only host the code, there is no central authority that could perform the task of continuous integration, whose importance has been thoroughly described in literature (e.g. [2]). This deficiency can be circumvented by project owners. They may decide to set up a CI server on their own and let it periodically download the most up-to-date source codes, compile them and test them. This facility is, however, not available for common users as it is not a standard feature. Thus the common users are left only with the test suite that is part of the project source codes. This is not an issue for functional tests, but with performance unit tests that are based on absolute time bounds this becomes critical.

Because absolute time bounds of the classical performance tests are coupled to a specific hardware configuration, they cannot be reliably executed by users on their local machines. The execution times would be probably different from those that have been measured when project owners wrote the tests. In this scenario, users can only commit their code and hope that they did not break any such test. It is up to project owners that know how and where to execute performance tests in order to verify that the new code may be accepted. This is clearly a problem since it creates a distance between programmers and the test suite, which should be, on the contrary, as close as possible, [43].

With UnitRacer, this can be resolved by attaching relative performance tests to the test suite that is distributed together with the rest of project's source codes. Because such tests are only loosely coupled to a hardware configuration, they may be evaluated on client machines of common users that are contributing remotely to the main project. When a relative performance test fails and a user has not made any changes to the application that could cause this, he needs to notify a project owner about this fact. The project owner can then assess the situation and either loosen the CSPL formula a bit so that the test is more resistant to hardware changes or realize that the assumed relation does not in fact hold on some particular hardware because of its specificity. In that case the entire application could get reconfigured so that it would use the faster approach or at least a new build for that platform could be created.

The main benefit of UnitRacer remains that it brings the tests closer to developers and promotes good quality assurance habits.

4.3 Continuous Comparison

So far, this chapter has been using UnitRacer to test units and architectural components. This section will focus on a whole new area of interest – relative performance testing of entire applications. The motivation behind this idea is that many applications use faster performance as an advantage over their competition. Because UnitRacer is designed for relative performance testing, it may be used also in this scenario. Of course, there are some restrictions regarding the

type of applications that are suitable for such testing. For units, the suitability criteria for relative performance testing is argued in [7] as "…we expect such [units] to be relatively small, often representing the computational kernel of an application, handling (bulk) data transformations and processing…". In case of entire applications the criteria is very similar. It can be even generalized that the applications should resemble units described by [7], i.e., libraries that other applications use for various computational tasks. UnitRacer, on the other hand, is not expected to be of any use for applications with graphical user interface.

In order to set up a relative performance testing on application level, it is first necessary to establish tested scenarios. A typical application will offer many functionalities to its clients and so only a subset of them should be selected so that they may be included in a test suite with reasonable size. The chosen functionalities should perhaps be those that represent the application as a whole and whose performance is the most important for its users. For example, for an application that performs compression and decompression from the ZIP file format the selected functionalities may be the very methods that perform the (de)compression.

Apart from scenarios, it is also necessary to choose the reference applications. In case of a ZIP library there may be a plenty of other already existing applications, so for example a few of the currently most popular ones may be selected. The reference applications are then tested against the developed application in the same manner as regular units are – the design process starts with capturing a CSPL formula that describes the desired performance relation, next a test with UnitRacer is written, then this test is added to a test suite and evaluated whenever necessary.

The application level performance testing brings the whole project several benefits. First of all, all stakeholders are kept informed on how is the application performance standing in comparison to industry standards. For commercial applications, it is important to know where the product stands among its competition. Other benefit is similar to principles of regression testing. Should some test suddenly fail after a change, the team is notified that the change would make the application slower compared to some already existing approaches. When some reference application has its source codes open to the public, this may be an impulse to learn from its design and modify the developed application so that its performance drawbacks are removed.

The way in which the application level relative performance testing is carried out can be very similar to the one for regular units and architectural components. The captured relations can get broken from two reasons – *a)* the performance of some reference application changes or *b)* the performance of the developed application itself changes. The first case may occur whenever a new version of a reference application is released. The latter case may occur generally after any commit to a version tracking system. The test suite should be also executed on any environment that the application targets.

This section continues with a practical example that demonstrates how application level testing with UnitRacer could have been used in a popular open-source project to avoid several performance related mistakes that have been made during

its development. First the project is introduced and then the sample application of UnitRacer is presented.

4.3.1 Json.NET

Continuous comparison of an application with its alternatives is best demonstrated on a project for which there exists a big number of such alternatives to choose from. On this basis, the *Json.NET* [44] project has been selected. It is a library that enables serialization and deserialization between runtime .NET objects and JSON format. The source code version tracking system of the library is freely available at a public hosting server [45]. This enabled the thesis to process the library revision by revision, as described later in the text. The library had, at the time of writing this thesis, more than 630,000 downloads from a popular distribution channel [46]. It has been also used in some very important projects for the .NET community such is the *MonoRail* project [47] that has been a popular web MVC framework until Microsoft released the official ASP.NET MVC or even in the Mono implementation of the .NET Framework.

The concept of the library is fairly simple. It works in a similar way as the XML serializers that are native part of the .NET Framework. Only instead of XML, the library is focused on *JSON* format [48] that is currently very popular in server-client communication or even as a database record format (e.g. [49] or [50]). When serializing a runtime .NET object to JSON, there are typically two scenarios – either the object has not been decorated with any serialization-guiding attributes or it uses such attributes to prescribe the serialization process. In the simple case, the serializer traverses all public properties of the object and converts them, together with their values, to JSON object fields. When the object is decorated with special attributes, the serialization may be taken over by some custom logic written specifically for that particular object so that the calling code has complete control over the process. The deserialization process is analogous.

However, there are some generic challenges that every JSON serialization library has to deal with. The one that is perhaps the most important from the performance point of view is the way reflection is used to get information about a runtime object. Common reflection approaches tend to be slower by an order of magnitude compared to new ones, similar to those that were discussed in section 4.1 when introducing the Ninject library. Other challenge is to provide library users with sufficiently rich customization options so that objects can be serialized to a desired JSON schema. There is also a lot of smaller, but no less important, issues like serialization of dates with time zones and other compatibility tasks.

Because JSON serialization is typically used in performance demanding applications, a well considered design is necessary for success of such library. The Json.NET library promotes itself by stating in [44] that it is "…high performance, faster than .NET's built-in JSON serializers…". But there are many other freely available libraries that compete with Json.NET. This thesis has focused on the following:

- DataContractJsonSerializer [51] – Official Microsfot JSON serializer intended for use in the WCF framework.

- FastJson [52] – Lightweight open-source implementation with minimum customizability.
- JavaScriptSerializer [53] – Other official Microsoft JSON serializer. Because of its less strict requirements on serialized objects, this one is intended for use in more heterogenous environments, like AJAX-enabled web applications.
- JayRock [54] – Open-source library focused on JSON RPC that features also a standalone serializer.
- JsonFx [55] – Open-source serializer that, apart from JSON, understands other formats like BSON and XML.
- ServiceStack [56] – Another generic open-source serializer that targets multiple output formats, JSON being one of them.

4.3.2 Monitoring Reference Applications with UnitRacer

To verify benefits of application level testing, this thesis has carried out an experiment. First a test suite that tested relative performance of the Json.NET library with other JSON serializer libraries has been created. Then development of the Json.NET library has been simulated using the freely available source codes from its version tracking system. The test suite has been evaluated each time after updating to a next revision. Because the assumed relations were truly broken by certain revisions, the thesis analyzes how these facts could have been used by the library to deliver a better performance.

It is necessary to say that the experiment is purely theoretical since it only uses fixed versions of competing libraries. The particular versions used are listed in table 4.2. During real development the competing libraries would probably release new versions simultaneously. However, the experiment is simplified and the Json.NET library is compared only against those listed releases.

Name	**Version**
DataContractJsonSerializer	4.0.0.0
FastJson	1.9.6
JavaScriptSerializer	4.0.0.0
JayRock	0.9.12915
JsonFx	2.0.1106.2610
ServiceStack	3.93

Table 4.2: Versions of competing serializers used in the experiment.

In order to simulate evolution of the tested library, each revision from its version tracking system [45] has been downloaded and compiled into an assembly.

At the time of writing this thesis there has been a total of 392 revisions available, for which it was possible to compile their source code without an error. The experiment is focused on a single scenario - standard serialization of an object to JSON without using any attributes that control the serialization process. The unit responsible for this functionality has been, however, renamed by programmers of the library in course of the development. Therefore, each revision had to be injected with a special additional class that wraps the responsible unit so that the name stays the same during all revisions.

Finally, the tests have been executed for each available revision to see how the library could have benefited from employing UnitRacer during its development. The figure 4.5 tracks evolution of mean execution time measured in each revision in comparison to the reference competing libraries.

The figure marks six important revisions that affected performance of the library. The first one and the second one are related, because the revision marked by number *1* brought the ability to control JSON serialization with special attributes. These attributes have been however checked by reflection each time a type was about to be serialized. The revision marked by number *2* started to cache these attributes in order for them to be checked exactly once. It can be seen from the chart that any redundant reflection operation means a big performance loss. It is hard to say whether the author of the library fixed this because he compared the performance to some competing library or not. However, the idea to cache reflected information is crucial also in the other libraries.

Revisions marked as *3* and *4* also appear to be related, but the truth is that the author introduced a bad design decision in *3* and fixed a different one in *4*. Nevertheless, the fix has helped to hide the performance penalty caused by the former change. The bad design decision was to start checking type's inheritance structure manually by calling reflection for each inheritance step instead of using prepared functions. The performance fix was to introduce a yet new caching mechanism that causes the mentioned query to be performed only once for each type. At the time, when the author was solving this performance issue, all other reference libraries had better performance. It is suspected that all of them have a better approach to using reflection.

The jumps *5* and *6* are both caused by manipulation of dates. The first jump started to use a more reliable approach for converting `DateTime` instances to UTC time. The new approach is officially preferred by the .NET Framework, because it is more precise and reacts to changes of system time even during execution of the code. This, however, eliminates caching and results in roughly 7 microseconds penalty on each call (measured on machine [$\mathcal{A}$]). The second jump added the ability to serialize the `Kind` property of the `DateTime` structure. Users are able to differentiate between `Local`, `UTC`, `Unspecified` and `Roundtrip`. Both revisions traded functionality for performance, which may be a defensible decision.

The thesis believes that if the project had been developed with UnitRacer, the author would recognize early in the development process that other existing libraries achieve a performance that is better by an order. This indicates several bad design decisions that really proved to be true later, when the author fixed them. It can be seen that the source codes for the Json.NET library contain also a small set of performance tests. These have the typical simple form that is

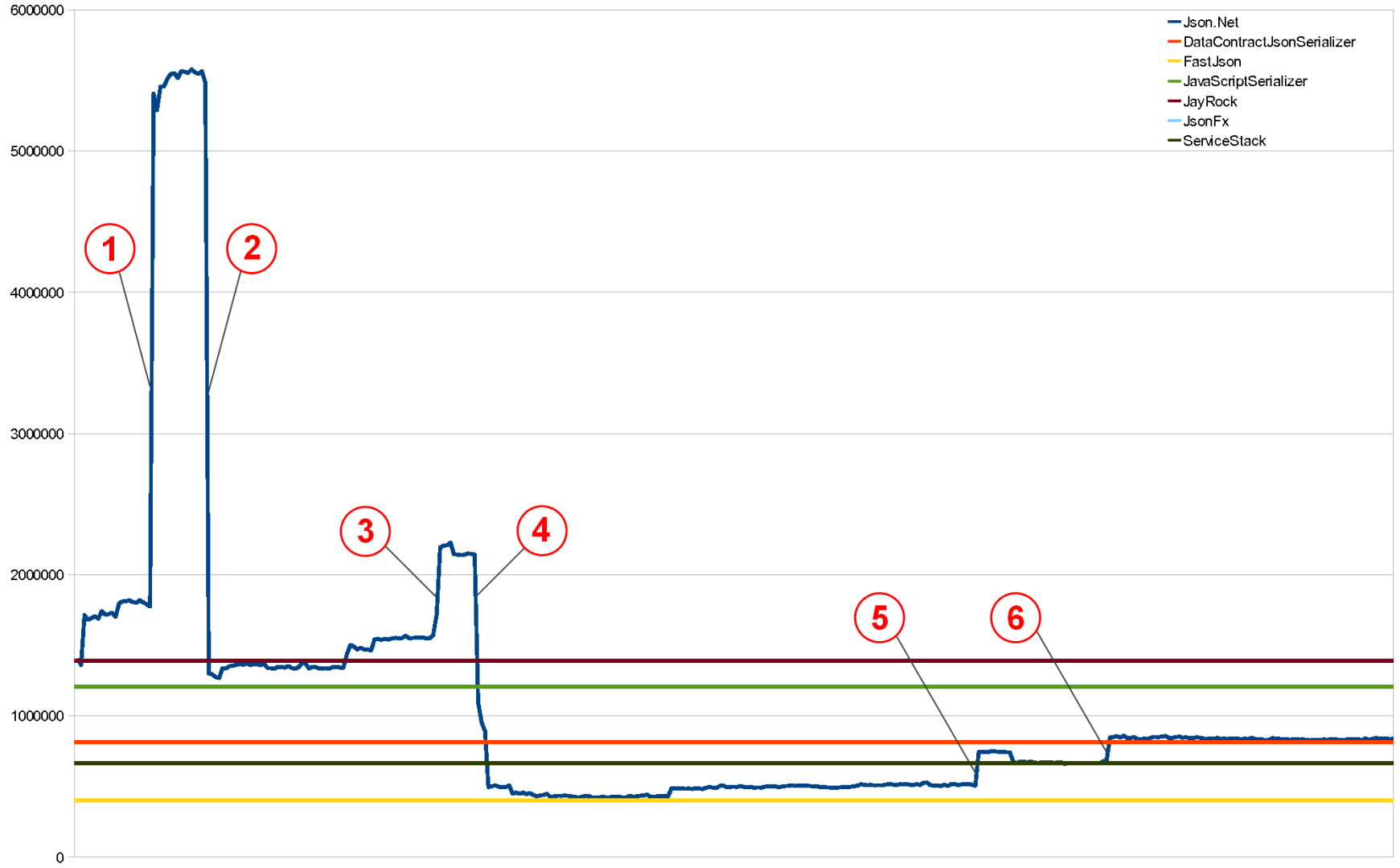


Figure 4.5: Evolution serialization unit's mean execution time.

expected from such tests when they are written without a designated framework library. Tests merely repeat execution of the tested unit one hundred times and then report the summed absolute measured time. The performance tests have been added to the project in revision marked by the number *2*, i.e., when the performance has been improved by the largest order.

Performance testing, even when started at the very beginning of a project, cannot compensate for bad design decisions. However, this thesis believes that by setting up a few tests, the development team of the project would be kept informed about the relation that their project has to other already existing applications. This could cause an impulse to analyze problematic areas and redesign the application in early stages where the cost of potential rewrites is not so big.

Conclusion

Functional testing is being employed in everyday projects as a measure of quality assurance. It is the cornerstone of popular TDD methodology and there are many tools that support such testing. Acceptance criteria for functional tests are usually easy to define and even the implementation is not expected to be difficult as the tests simply reproduce certain steps on a unit in isolation.

In contrast, performance testing is often skipped altogether in spite of its benefits for the target application. Unsatisfactory performance may be a reason why application is not accepted by clients or why it fails with users. In other cases there might be precisely defined performance requirements and because of missing testing process a developed application is not able to meet them.

There are several reasons why performance testing is not employed as often as functional testing. First of all it is difficult to capture what is slow and what is sufficiently fast. With absolute time bounds, the precise hardware configuration has to be part of the test specification but that makes the criteria too specific to be of any use in common environment with many diverse hardware configurations. It is also hard to write the tests, because their set up and evaluation with existing tools is often difficult.

To this end, the article [7] has introduced a new approach that is focused on solving the common issues. The approach brings the concept of relative performance testing, where units are compared together instead of using an absolute time bound to express acceptance criteria. The article defines Stochastic Performance Logic that formalizes the way in which performance assumptions are to be captured. It is argued that it is more natural for programmers or for any stakeholder in general to express the required performance of some unit in a relative way by comparing it to some other unit that the person is already familiar with. The article then introduces a sample-based interpretation that uses measurements of unit execution times and statistical tests to evaluate SPL formulas. It is further discussed how the approach could be put into work by arguing its fitness for purpose, presenting an algorithm for evaluation and proposing a particular integration technique for the Java programming language.

This thesis has continued and elaborated further on the topic. In chapter 1 a Customized Stochastic Performance Logic is introduced as a modification of the original SPL. Some original features are left out, but the logic adds new relation symbols so that it may be used also in scenarios where testing against an absolute time bound is necessary, thus broadening the area of its applicability.

In chapter 2 a sample-based interpretation is presented for the CSPL logic in a similar way that [7] did for SPL. Stated theorems and their proofs show the correctness of the newly introduced logic. The chapter also shows how the

introduced symbols are to be understood and assigned their semantic meaning in practical scenarios.

UnitRacer, a library that provides means of relative performance testing, is introduced in chapter 3. The library has been implemented as an important part of this thesis and demonstrates the theoretical concepts in practice. It targets the .NET Framework and uses many of its novel features to increase its usability and make the process of writing performance unit tests as simple as possible. The library is highly configurable – the method in which CSPL formulas are specified may be changed at any time as well the format of evaluation reports that the library may produce as a part of the evaluation process. UnitRacer uses a customizable set of statistical tests that are used to decide value of hypotheses in a formula. By default Student's t-test and Welch's t-test are provided, but the library is open to new ones, perhaps more suitable for a scenario not foreseen by this thesis.

The discussion of UnitRacer's applicability in real-life software projects is discussed throughout the thesis, but to sufficiently support its benefits, chapter 4 presents practical use cases, where the library has been used in order to improve quality of existing popular open-source projects. The chapter also discusses some general issues regarding relative performance testing and also some best practices that have been observed during the use cases and that are expected to help to deliver tests of better quality. First, it is demonstrated how UnitRacer may be used in the phase where application's architecture is decided. Next, testing of generic units similar to the well-known functional unit testing is analyzed together with notes on prevention of performance regression. The last use case puts the entire development period of a selected application in comparison with similar competing projects. The use case shows how keeping in touch with industry standards may positively affect the quality of the resulting product.

At this point, the research has been concluded but there are still some final words missing that will put the thesis in a better perspective.

Drawbacks

Although UnitRacer solves many problems of the classical absolute time bound based performance testing approach, there are still some drawbacks to using the library that the author of this thesis is aware of.

The first important disadvantage of the current implementation is that it only offers statistical tests for normally distributed random variables. For CSPL relation operators that compare two units the Welch's t-test is used. For CSPL relation operators that compare a unit with a time bound the Student's t-test is used. None of these tests require any additional knowledge about model parameters except that the distribution is normal. This may still be very limiting as it is hard to ensure this requirement for common units that are to be tested.

One way to formally satisfy this constraint is for the programmer to perform a sample evaluation of the test and then use data from the evaluation report to decide whether the distribution is really normal. This can be done by using a designated test like Shapiro-Wilk [57], Anderson-Darling [58] or any other suitable normality test. The problem is that even if the distribution is verified to be

normal, it may no longer be so on some other hardware configuration. This has been experienced while writing this thesis as a custom implementation of the Shell Sort algorithm has passed a normality test at a relatively strict significance level of $\alpha = 0.01$ on machine $[\mathcal{A}]$, but on machine $[\mathcal{B}]$ the histogram of measured method execution times has not been bell-shaped but rather appeared as a histogram of a multimodal distribution.

The programmer may of course ignore the normality requirement at any time. This will cause the tests evaluated by UnitRacer to stop being precise and reliable, but it is suspected that even in this case the library may be of some benefit. This decision is at programmer's own risk, however.

Next disadvantage is common to any performance testing in general – performance testing requires for the computing environment not to be shared with other applications that could interfere during the measuring process. When UnitRacer is sampling method execution times and user suddenly starts to, e.g., decompress a ZIP file, the measured execution times will naturally jump up by an order during the decompression. This is an obstacle for everyday usage of the library as this issue does not exist for functional unit tests and the programmer may continue with his work while the tests are evaluating. Letting a programmer wait for extended periods of time may cause him to stop using the performance test suite altogether, as argued in [59]. A solution to this problem is to have only the continuous integration server perform this sort of tests. This is not a big drawback and the thesis recommends this approach as a solution, although it puts the tests away from the programmer who is making the changes.

Yet another counter-argument for using the UnitRacer library may be that the tests require too much effort to configure. For common performance unit tests, the thesis does not expect this to be an issue. However, for use cases similar to the one described in 4.2.1 the required time is in fact higher than with other common test scenarios. The user has to compile the base revision into a standalone assembly and then link this assembly to a test project. This process may become too confusing in enterprise environment and so perhaps some tooling support would be crucial for this case.

On the other hand, the author of this thesis had a great opportunity to test the library on several projects and a big number of sample scenarios. The library has been very reliable in its results, even when working in an environment shared with other running applications. The common relative performance unit tests were quick to write and captured performance expectation in a natural straightforward way. This makes the author believe that there is a real potential in this testing approach.

Future Work

The goal of the thesis has been to use the theory from article [7] and verify its applicability in real-world scenarios. Based on customized logic a practical library called UnitRacer has been implemented and employed in sample use cases. The thesis has fulfilled its scope but nevertheless some topics have been observed that could be further elaborated.

The set of statistical tests available in UnitRacer could be enriched with new ones. Perhaps with some non-parametric tests that do not rely on any particular distribution of the underlying data. This would make the library more universal and eliminate some of its current constraints.

Apart from `XmlReportGenerator` the library could benefit from some real-time logger of the evaluation process. For example, a `ConsoleReportGenerator` could write information about the currently processed test to the standard console output. This would give feedback about the progress, which would be valuable for long running tests.

New tools could be also developed for regression testing use cases. When a unit is compared to a certain base revision of itself, the process of compiling the base revision into a standalone assembly requires perhaps too much effort. This may discourage the development team from using these tests. A plugin for continuous integration servers could be created that understands UnitRacer reports and appends their data to its native log that is usually sent via email messages. The plugin could also allow regression testing based solely on specification of number of the base revision and create the assembly automatically.

It is also possible that the current method of test specification in UnitRacer is too lengthy. In that case a new *Experiment* component would be necessary in order to reduce the number of lines required to set up a working test.

Bibliography

[1] D. Huizinga and A. Kolawa. *Automated Defect Prevention: Best Practices in Software Management*. Hoboken, N.J.: Wiley-IEEE Computer Society, c2007, xxvi, 426 p. ISBN 04-700-4212-5.

[2] S. McConell. *Code Complete*. 2nd ed. Washington: Microsoft Press, c2004, xxxvii, 914 p. ISBN 07-356-1967-0.

[3] K. Beck. *Test-Driven Development: By Example*. Boston: Addison-Wesley, c2003, xix, 220 p. ISBN 03-211-4653-0.

[4] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ: Addison-Wesley, c2007, lx, 883 p. ISBN 01-314-9505-4.

[5] R. Osherove. *The Art of Unit Testing: With Examples in .NET*. Greenwich, CT.: Manning, 2009, xxiii, 296 p. ISBN 19-339-8827-4.

[6] M. Fowler. Continuous Integration. In: *Martin Fowler* [online]. 01 May 2006 [cit. 2012-10-21]. Available at: http://martinfowler.com/articles/continuousIntegration.html

[7] L. Bulej, T. Bureš, J. Keznikl, A. Koubková, A. Podzimek and P. Tůma. Capturing Performance Assumptions using Stochastic Performance Logic. In: *ICPE '12 Proceedings of the third joint WOSP/SIPEW international conference on Performance Engineering*. New York: ACM, 2012, pp. 311-322.

[8] D. E. Knuth. *The art of computer programming*. 3rd ed. Upper Saddle River: Addison-Wesley, c1998, xiii, 780 p. ISBN 02-018-9685-0.

[9] H. P. Barendregt. *Lambda Calculus: Its Syntax and Semantics*. Rev. ed., paperback ed. Amsterdam: Elsevier, 2001, xv, 621 p. ISBN 04-448-6748-1.

[10] M. H. DeGroot, M. J. Schervish. *Probability and statistics*. 4th ed. Boston: Addison-Wesley, c2012, xiv, 893 p. ISBN 03-215-0046-6.

[11] B. L. Welch. The Generalization of 'Student's' Problem when Several Different Population Variances are Involved. *Biometrika*. 1947, 34 (1-2), pp. 28-35. ISSN 0006-3444. DOI: 10.1093/biomet/34.1-2.28.

[12] *Math.NET Numerics* [online]. 2012 [cit. 2012-10-24]. Available at: http://numerics.mathdotnet.com/

[13] *ZedGraph* [online]. 2012 [cit. 2012-11-29]. Available at: http://sourceforge.net/projects/zedgraph/

[14] Code Metrics Values. In: *MSDN: Microsoft Developer Network* [online]. 2012 [cit. 2012-10-24]. Available at: http://msdn.microsoft.com/en-us/library/bb385914.aspx

[15] *Moq: The simplest mocking library for .NET and Silverlight* [online]. 2012 [cit. 2012-10-25]. Available at: http://code.google.com/p/moq/

[16] S. C. Kleene. *Introduction to Metamathematics*. 6th Reprint. Groningen: Wolters-Noordhoff, 1971, 10, 550 p. Bibliotheca mathematica. ISBN 04-441-0088-1.

[17] M. Fowler. FluentInterface. In: *Martin Fowler* [online]. 20 December 2005 [cit. 2012-10-25]. Available at: http://www.martinfowler.com/bliki/FluentInterface.html

[18] N. Chen. Convention over Configuration. In: *Software Engineering Matters* [online]. November 29, 2006 [cit. 2012-10-25]. Available at: http://softwareengineering.vazexqi.com/files/pattern.html

[19] *NAnt* [online]. June 9, 2012 [cit. 2012-10-25]. Available at: http://nant.sourceforge.net/

[20] *CruiseControl.NET* [online]. Sep. 5, 2011 [cit. 2012-10-25]. Available at: http://www.cruisecontrolnet.org/

[21] R. Osherove. Try to avoid multiple asserts in a single unit test. In: *Roy Osherove: Team Leadership, Agile & Lean Development, Ruby .NET & Java- Speaking, Consulting, Training and Tools* [online]. April 14, 2005 [cit. 2012-10-25]. Available at: http://osherove.com/blog/2005/4/14/try-to-avoid-multiple-asserts-in-a-single-unit-test.html

[22] D. M. Hawkins. *Identification of Outliers*. New York: Chapman and Hall, 1980, x, 188 p. ISBN 04-122-1900-X.

[23] F. R. Hampel. Robust Estimation: A Condensed Partial Survey. In: *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete*. Berlin: Springer-Verlag, June 1973, pp. 87-104. 27: 2. DOI: 10.1007/BF00536619.

[24] D. Clark. Using Consensus Ensebles to Identify Suspect Data. In: *The Information Science Discussion Paper Series*. Berlin: Springer Berlin Heidelberg, November 2000, pp. 483-490. Number 2000: 17. ISSN 1172-6024. DOI: 10.1007/978-3-540-30133-2_63.

[25] J. I. Maletic and A. Marcus. Data Cleansing: Beyond Integrity Analysis. In: *Proceedings of The Conference on Information Quality*. Cambridge, MA: Massachusetts Institute of Technology, 2000, pp. 200-209. IQ2000.

[26] I. Ben-Gal. Outlier Detection. In: O. Z. Maimon and L. Rokach. *Data Mining and Knowledge Discovery Handbook*. New York: Springer, c2005, pp. 131-146. ISBN 0-387-24435-2. DOI: 10.1007/0-387-25465-X_7.

[27] E. M. Knorr and R. T. Ng. A Unified Notion of Outliers: Properties and Computation. In: *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining*. Menlo Park, CA: AAAI, 1997, pp. 219-222. ISBN 978-1-57735-027-9.

[28] D. Freedman, R. Pisani and R. Purves. *Statistics*. 1st ed. New York: Norton, c1978, xv, 506, 83 p. ISBN 03-930-9076-0.

[29] Microsoft. *Microsoft Chart* [online]. May, 2010 [cit. 2012-10-25]. Available at: http://archive.msdn.microsoft.com/mschart

[30] S. McConnell, Steve. *Software Estimation: Demystifying the Black Art*. Redmond: Microsoft Press, 2006, xxix, 308 p. ISBN 0735605351.

[31] M. Fowler. Inversion of Control Containers and the Dependency Injection pattern. In: *Martin Fowler* [online]. 23 January 2004 [cit. 2012-10-28]. Available at:
http://www.martinfowler.com/articles/injection.html

[32] *Ninject: Open Source Dependency Injector for .NET* [online]. 2012 [cit. 2012-10-28]. Available at:
http://www.ninject.org/

[33] Ninject. Outercurve Foundation. *NuGet Gallery* [online]. 2012 [cit. 2012-10-28]. Available at:
http://nuget.org/packages/ninject

[34] A. Freeman. *Pro ASP.NET MVC 3 Framework*. 3rd. ed. New York: Apress, c2011, xxv, 824 p. The Expert's Voice in .NET. ISBN 978-1-4302-3404-3.

[35] A. Sur. Reflection - Slow or Fast? Demonstration with Solutions. In: *DOT NET TRICKS* [online]. August 23, 2012 [cit. 2012-10-28]. Available at:
http://www.abhisheksur.com/2010/11/reflection-slow-or-faster-demonstration.html

[36] Reflection Emit Dynamic Method Scenarios. In: *MSDN Library* [online]. 2012 [cit. 2012-10-28]. Available at:
http://msdn.microsoft.com/en-us/library/sfk2s47t.aspx

[37] J. Pobar. Dodge Common Performance Pitfalls to Craft Speedy Applications. *MSDN Magazine: The Microsoft Journal for Developers* [online]. 2005, 2005-July [cit. 2012-10-28]. Available at:
http://msdn.microsoft.com/magazine/cc163759.aspx

[38] W. Agresti, F. McGarry, D. Card, J. Page, V. Church and R. Werking. NASA. *Manager's Handbook for Software Development*. Revision 1. Greenbelt, Maryland, November 1990.

[39] J. Shore. Red-Green-Refactor. In: *The Art of Agile* [online]. 30 Nov, 2005 [cit. 2012-10-29]. Available at:
http://www.jamesshore.com/Blog/Red-Green-Refactor.html

[40] A. V. Aho and M. J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. In: G. Manacher. *Communications of the ACM*. Murray Hill, NJ: Association for Computing Machinery, c1975, pp. 333-340. June 1975: Volume 18, Number 6. DOI: 10.1145/360825.360855.

[41] M. Fowler. *Patterns of Enterprise Application Architecture*. Boston: Addison-Wesley, c2003, xxiv, 533 p. ISBN 03-211-2742-0.

[42] Github Inc. *GitHub: Social Coding* [online]. 2012 [cit. 2012-10-29]. Available at:
https://github.com/

[43] R. C. Martin, *Clean code: A Handbook of Agile Software Craftsmanship*. Upper Saddle River, NJ: Prentice Hall, c2009, xxix, 431 p. ISBN 01-323-5088-2.

[44] J. Newton-King. Json.NET. *James Newton-King* [online]. September 09, 2007 [cit. 2012-10-29]. Available at:
http://james.newtonking.com/pages/json-net.aspx

[45] JamesNK / Newtonsoft.Json. In: *GitHub: Social Coding* [online]. 2012 [cit. 2012-10-29]. Available at:
https://github.com/JamesNK/Newtonsoft.Json

[46] Json.NET. Outercurve Foundation. *NuGet Gallery* [online]. 2012 [cit. 2012-10-29]. Available at:
http://nuget.org/packages/newtonsoft.json

[47] MonoRail. *Castle Project* [online]. 2012 [cit. 2012-10-29]. Available at:
http://www.castleproject.org/projects/monorail/

[48] *JSON* [online]. 2012 [cit. 2012-10-29]. Available at:
http://json.org/

[49] 10GEN, Inc. *MongoDB* [online]. 2012 [cit. 2012-10-29]. Available at:
http://www.mongodb.org/display/DOCS/Home

[50] The Dojo Foundation. *Preserve* [online]. 2012 [cit. 2012-10-29]. Available at:
http://www.persvr.org/

[51] DataContractJsonSerializer Class. In: *MSDN: Microsoft Developer Network* [online]. 2012 [cit. 2012-10-29]. Available at:
http://msdn.microsoft.com/library/system.runtime.serialization.json.datacontractjsonserializer.aspx

[52] M. Gholam. FastJSON. In: *CodeProject: For Those Who Code* [online]. 23 Oct 2012 [cit. 2012-10-29]. Available at:
http://www.codeproject.com/Articles/159450/fastJSON

[53] JavaScriptSerializer Class. In: *MSDN: Microsoft Developer Network* [online]. 2012 [cit. 2012-10-29]. Available at: http://msdn.microsoft.com/library/system.web.script.serialization.javascriptserializer.aspx

[54] JayRock: JSON and JSON-RPC for .NET. *BerliOS: The Open Source Mediator* [online]. 2012 [cit. 2012-10-29]. Available at: http://jayrock.berlios.de/

[55] S. M. McKamey. *JsonFx.NET* [online]. c2006-2009 [cit. 2012-10-29]. Available at: http://www.jsonfx.net/

[56] ServiceStack / ServiceStack.Text. In: *GitHub: Social Coding* [online]. 2012 [cit. 2012-10-29]. Available at: https://github.com/ServiceStack/ServiceStack.Text

[57] S. S. Shapiro and M. B. Wilk. An Analysis of Variance Test for Normality (Complete Samples). In: *Biometrika*. United Kingdom: Oxford University Press, December 1965, pp. 591–611. 52: 3-4. ISSN 0006-3444. DOI: 10.1093/biomet/52.3-4.591

[58] T. W. Anderson and D. A. Darling. Asymptotic Theory of Certain "goodness-of-fit" Criteria Based on Stochastic Processes. In: *Annals of Mathematical Statistics*. USA: Institute of Mathematical Statistics, 1952, pp. 193–212. 23: 2. ISSN 0003-4851. DOI: 10.1214/aoms/1177729437.

[59] T. Ottinger and J. Langr. Unit Tests Are FIRST. In: *Pragpub* [online]. January 2012 [cit. 2012-10-30]. ISSN 1948-3562. Available at: http://pragprog.com/magazines/2012-01/unit-tests-are-first

Appendix A

Test Machines

[$\mathcal{A}$]

- AMD Turion 64 MK38 @ 2.2 GHz
- 3.12 GB DDR2 @ 667 MHz
- Windows XP SP3; Command-Prompt mode
- Microsoft .NET Framework v4.0.30319

[$\mathcal{B}$]

- Intel Core i7 CPU 920 @ 2.67 GHz
- 1.88 GB DDR3 @ 1066 MHz
- Gentoo 2.1; Linux kernel 3.4.11; Non-essential services turned off
- Mono 2.10.9

Appendix B

DVD

This thesis comes with an attached DVD, whose content is following:

`Thesis`	
`thesis.pdf`	Copy of this text.
`UnitRacer`	
`binaries.zip`	Compiled ready-to-use assemblies.
`reports.zip`	Sample reports for ReportInspector.
`sources.zip`	Library source codes.
`worked_example.zip`	Worked example.
`UseCases`	
`Json.NET`	
`competition.zip`	Redistributables of competing libraries.
`revisions.7z`	Compiled Json.NET revisions.
`Ninject`	
`sources.zip`	Tests for the Ninject use case.