Capturing Performance Assumptions using Stochastic Performance Logic

Lubomír Bulej[1,2] Tomáš Bureš[1,2] Jaroslav Keznikl[1,2]
Alena Koubková[1] Andrej Podzimek[1,2] Petr Tůma[1]

[1]Charles University in Prague
Faculty of Mathematics and Physics
Malostranské náměstí 25
118 00 Prague 1, Czech Republic

[2]Academy of Sciences of the Czech Republic
Institute of Computer Science
Pod Vodárenskou věží 2
182 07 Prague 8, Czech Republic

{bulej,bures,keznikl,koubkova,podzimek,tuma}@d3s.mff.cuni.cz

ABSTRACT

Compared to functional unit testing, automated performance testing is difficult, partially because correctness criteria are more difficult to express for performance than for functionality. Where existing approaches rely on absolute bounds on the execution time, we aim to express assertions on code performance in relative, hardware-independent terms. To this end, we introduce Stochastic Performance Logic (SPL), which allows making statements about relative method performance. Since SPL interpretation is based on statistical tests applied to performance measurements, it allows (for a special class of formulas) calculating the minimum probability at which a particular SPL formula holds. We prove basic properties of the logic and present an algorithm for SAT-solver-guided evaluation of SPL formulas, which allows optimizing the number of performance measurements that need to be made. Finally, we propose integration of SPL formulas with Java code using higher-level performance annotations, for performance testing and documentation purposes.

Categories and Subject Descriptors

D.2.4 [**Software/Program Verification**]: Assertion checkers; F.3.1 [**Specifying and Verifying and Reasoning about Programs**]: Assertions

General Terms

Algorithms, Theory

Keywords

performance testing, regression benchmarking

ICPE'12, April 22-25, 2012, Boston, Massachusetts, USA

1. INTRODUCTION

Closing the gap between code and documentation is an important trend that can be found in modern software engineering approaches, such as test driven development [1] or design by contract [2]. In both cases, the project code is imbued with additional information, capturing developer assumptions or intended usage. Such information usually takes the form of assertions, unit tests, or preconditions and postconditions associated with individual methods. Besides enhancing the documentation, this information lends itself to automatic testing or formal verification, which can be easily incorporated into the development process. By using tools such as JUnit [3] or Google Test [4] for testing, and Java Modeling Language [5] or Microsoft Verifier for Concurrent C [6] for verification, the developers gain more freedom in exploring design choices and evolve existing design to meet new requirements. Should they make a mistake, an automatic safety net will promply inform them of assertion or contract violations, or newly introduced bugs.

However, the tools available today are mostly geared towards functional testing. We believe there is little doubt that similar support for performance testing – that is, being able to express performance-related developer assumptions or intended usage in code and test or verify them automatically – would be beneficial. Yet it is, unfortunately, more difficult to do – for multiple reasons:

- Except in special application domains, such as real-time systems, the boundary between sufficient and insufficient performance is not sharp. It is therefore more difficult to specify conditions that should be tested.
- Performance is typically platform-dependent, but the conditions that should be met need to be more general, lest their utility is severely limited.[1]
- Performance testing can be more difficult to configure and execute than functional testing.

In past work [7, 8, 9], we focused on the execution and evaluation aspects of performance testing. Here, we contribute to the ability to specify conditions that should be tested.

[1]Actually, test results are also platform-dependent and therefore of potentially limited utility. On the other hand, the fact that a test is only testing a particular execution on a particular platform is generally accepted, this issue is therefore not unique to performance.

To avoid expressing conditions in a platform-dependent manner, we have decided to rely on relative terms – we develop a special many-sorted first-order logic that allows us to express statements about relative performance of functions or methods in code. The logic, here called Stochastic Performance Logic (SPL), is interpreted using statistical hypothesis testing, which allows calculating the probability at which a particular SPL statement holds.

Besides introducing the logic and its interpretation, we also prove basic properties of the logic and present an algorithm for SAT-solver-guided evaluation of SPL formulas, which allows optimizing the number of performance measurements that need to be made. We also propose to integrate SPL formulas with Java code using annotations.

Among potential applications of our work, we see the possibility to document developer assumptions related to performance – for example, when the developer implements a user interface method with the assumption that caching a bitmap representation of a picture is faster than decoding the picture on each draw, that assumption can be represented in code and tested automatically. The assumptions can also be interpreted as a contract between code and external components and tested during integration, making the integration process more reliable. Another useful application is an aid in debugging, similar to traditional assertions.

The paper is structured as follows. We define the Stochastic Performance Logic (SPL) in Section 2, to provide a formal ground for statements about performance. To illustrate the semantics of SPL, we introduce a natural interpretation of the logic in Section 3, while for use with real-world performance data, we define sample-based SPL interpretation in Section 4. In Section 5 we discuss the issues related to statistical errors when evaluating SPL formulas and show that for a special group of practically relevant formulas, the probability of error can be bounded. The fitness of SPL for performance comparisons is discussed and evaluated in Section 6. For efficient evaluation of SPL formulas in performance unit testing, we introduce a SAT-solver-guided algorithm in Section 7, and outline the potential integration of SPL into Java programs in Section 8. In closing, we discuss related approaches in Section 9, and conclude the paper in Section 10.

2. STOCHASTIC PERFORMANCE LOGIC

To avoid platform dependency found in statements expressing that a method completes its operation in certain time bounds, we need to compare the performance of one method to performance of some other method. Thus 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 attention.

To achieve this, we formally define the performance of a method as a random variable representing the time it takes to execute the method with random input parameters. The nature of the random input is formally represented by *workload class* and *method workload*. The workload is parametrized by *workload parameters*, which capture the dimensions along which the workload can be varied, e.g. array size, matrix sparsity, numberof vertices in a graph, etc.

Definition 1. Workload class is a function $\mathfrak{L} : P^n \rightarrow (\Omega \rightarrow I)$, where for a given $\mathfrak{L}$, P is a set of *workload parameter* values, n is the number of parameters, Ω is a sample space, and I is a set of objects (method input arguments) in a chosen programming language.

For later definitions we also require that there is a total ordering over P.

Definition 2. Method workload is a random variable $L^{p_1,\ldots,p_n}$ such that $L^{p_1,\ldots,p_n} = \mathfrak{L}(p_1,\ldots,p_n)$ for a given workload class $\mathfrak{L}$ and parameters $p_1,\ldots,p_n$.

Unlike conventional random variables that map observations to a real number, method workload is a random variable that maps observations to object instances, which serve as random input parameters for the method under test. If necessary, the developer may adjust the underlying stochastic process to obtain random input parameters representing domain-specific workloads, e.g., partially sorted arrays.

To demonstrate the above concepts, let us assume we want to measure the performance of a method S, which sorts an array of integers. The input parameters for the sort method S are characterized by workload class $\mathfrak{L}_S : \mathbb{N}^+ \rightarrow (\Omega_S \rightarrow I_S)$. Let us assume that the workload class $\mathfrak{L}_S$ represents an array of random integers, with a single parameter determining the size of the array. The method workload returned by the workload class is a random variable, whose observations are instances of random arrays of given size. For example, method inputs in form of random arrays of size 1000 will be obtained from observations of random variable $L_S^{1000} : \Omega_S \rightarrow I_S = \mathfrak{L}_S(1000)$.

Note that without loss of generality, we assume in the formalization that there is exactly one $\mathfrak{L}_M$ for a particular method M and that M has just one input argument.

With the formal representation of a workload in place, we now proceed to define the method performance.

Definition 3. Let $M(in)$ be a method in a chosen programming language and $in \in I$ its input argument. Then *method performance* $P_M : P^n \rightarrow (\Omega \rightarrow \mathbb{R})$ is a function that for given workload parameters $p_1,\ldots,p_n$ returns a random variable, whose observations correspond to execution duration of method M with input parameters obtained from observations of $L_M^{p_1,\ldots,p_n} = \mathfrak{L}_M(p_1,\ldots,p_n)$, where $\mathfrak{L}_M$ is the workload class for method M.

We can now define the Stochastic Performance Logic (SPL) that will let us make comparative statements about method performance under a particular method workload. To facilitate comparison of method performance, SPL is based on regular arithmetics, in particular on axioms of equality and inequality adapted for the method performance domain.

Definition 4. SPL is a many-sorted first-order logic defined as follows:

- There is a set $FunPe$ of function symbols for method performances with arities $P^n \rightarrow (\Omega \rightarrow \mathbb{R})$ for $n \in \mathbb{N}^+$.
- There is a set $FunT$ of function symbols for performance observation transformation functions with arity $\mathbb{R} \rightarrow \mathbb{R}$.
- The logic has equality and inequality relations $=, \leq$ for arity $P \times P$.
- The logic has equality and inequality relations $\leq_{p(tl,tr)}$, $=_{p(tl,tr)}$ with arity $(\Omega \rightarrow \mathbb{R}) \times (\Omega \rightarrow \mathbb{R})$, where $tl, tr \in FunT$.

- Quantifiers (both universal and existential) are allowed only over finite subsets of P.
- For $x, y, z \in P$ and $P_M, P_N \in FunPe$, the logic has the following axioms:

$$x \leq x \tag{1}$$
$$(x \leq y \wedge y \leq x) \leftrightarrow x = y \tag{2}$$
$$(x \leq y \wedge y \leq z) \rightarrow x \leq z \tag{3}$$

$$\text{For each pair } tl, tr \in FunT \text{ such that}$$
$$\forall o \in \mathbb{R} : tl(o) \leq tr(o), \text{there is an axiom} \tag{4}$$
$$P_M(x_1, \ldots, x_m) \leq_{p(tl,tr)} P_M(x_1, \ldots, x_m)$$

$$(P_M(x_1, \ldots, x_m) \leq_{p(tm,tn)} P_N(y_1, \ldots, y_n) \wedge$$
$$P_N(y_1, \ldots, y_n) \leq_{p(tn,tm)} P_M(x_1, \ldots, x_m)) \leftrightarrow \tag{5}$$
$$P_M(x_1, \ldots, x_m) =_{p(tm,tn)} P_N(y_1, \ldots, y_n)$$

Axioms (1)–(3) come from arithmetics, since workload parameters (P) are essentially real or integer numbers. In analogy to (1)–(2), axiom (4) may be regarded as generalised reflexivity, and axiom (5) shows the correspondence between $=_p$ and $\leq_p$. An analogy of (3), i.e. transitivity, cannot be introduced for $=_p$ and $\leq_p$, because it does not hold for all interpretations of SPL (see Section 4).

Note that even though we currently do not make use of the axioms in our approach, they make the properties of the logic more obvious (in particular the performance relations $=_p$ and $\leq_p$). Specifically, the lack of transitivity for performance relations ensures that SPL formulas can only express statements that are consistent with hypothesis testing approaches used in the SPL interpretation.

Using the logic defined above, we would like to express assumptions about method performance in the spirit of the following examples:

Example 1. "On arrays of 100, 500, 1000, 5000, and 10000 elements, the sorting algorithm A is at most 5% faster and at most 5% slower than sorting algorithm B."

$$\forall n \in \{100, 500, 1000, 5000, 10000\} :$$
$$P_A(n) \geq_{p(id,\lambda x.0.95x)} P_B(n) \wedge P_A(n) \leq_{p(id,\lambda x.1.05x)} P_B(n)$$

Example 2. "On buffers of 256, 1024, 4096, 16384, and 65536 bytes, the Rijndael encryption algorithm is at least 10% faster than the Blowfish encryption algorithm and at most 200 times slower than array copy."

$$\forall n \in \{256, 1024, 4096, 16384, 65536\} :$$
$$P_{Rijndael}(n) \leq_{p(id,\lambda x.0.9x)} P_{Blowfish}(n) \wedge$$
$$P_{Rijndael}(n) \leq_{p(id,\lambda x.200x)} P_{ArrayCopy}(n)$$

For compact in-place representation of performance observation transformation functions, we use the lambda notation [10], with id as a shortcut for identity, $id = \lambda x.x$.

To ensure correspondence between SPL formulas in Examples 1 and 2 and their textual description, we need to define SPL semantics that provides the intended interpretation.

3. SPL INTERPRETATION

A natural way to compare random variables is to compare their expected values. Since method performance is a random variable, it is only natural to base SPL interpretation, and particularly the interpretation of equality and inequality relations, on the expected value of method performance. Other (valid) interpretations are possible, but for simplicity, we first define the *expected-value-based interpretation* and prove its consistency with the SPL axioms.

Each function symbol $f_{Pe} \in FunPe$ is interpreted as a method performance, i.e. an n-ary function that for input parameter $p_1, \ldots, p_n$ returns a random variable $\Omega \rightarrow \mathbb{R}$, the observation of which corresponds to performance observation as defined in Definition 3.

Each function symbol $f_T \in FunT$ is interpreted as a performance observation transformation function, which is a function $\mathbb{R} \rightarrow \mathbb{R}$. In the context of equality and inequality relations between method performances, f_T represents transformation (e.g. scaling) of the observed performance – e.g., statement "*M is 2 times slower than N*" is expressed as $P_M =_{p(id,\lambda x.2x)} P_N$, where $f_{T_1} = id$ and $f_{T_2} = \lambda x.2x$.

The relational operators $\leq$ and $=$ for arity $P \times P$ are interpreted in the classic way, based on total ordering of P.

The interpretation of the relational operators $=_p$ and $\leq_p$ is defined as follows:

Definition 5. Let $tm, tn : \mathbb{R} \rightarrow \mathbb{R}$ be performance observation transformation functions, P_M and P_N be method performances, and $x_1, \ldots, x_m, y_1, \ldots, x_n$ be workload parameters. Then the relations $\leq_{p(tm,tn)}$, $=_{p(tm,tn)}$: $(\Omega \rightarrow \mathbb{R}) \times (\Omega \rightarrow \mathbb{R})$ are interpreted as follows:

$$P_M(x_1, \ldots, x_m) \leq_{p(tm,tn)} P_N(y_1, \ldots, y_n) \quad \text{iff}$$
$$E(tm(P_M(x_1, \ldots, x_m))) \leq E(tn(P_N(y_1, \ldots, y_n)));$$

$$P_M(x_1, \ldots, x_m) =_{p(tm,tn)} P_N(y_1, \ldots, y_n) \quad \text{iff}$$
$$E(tm(P_M(x_1, \ldots, x_m))) = E(tn(P_N(y_1, \ldots, y_n))),$$

where $E(X)$ denotes the expected value of the random variable X, and $tm(X)$ denotes a random variable derived from X by applying function tm on each observation of X.[2]

At this point, it is clear that the expected-value-based interpretation of SPL has the required semantics. However, we have yet to show that this interpretation is consistent with the SPL axioms.

The following lemma and theorem show that the interpretation of $=_p$ and $\leq_p$, as defined by Definition 5, is consistent with axioms (4) and (5). The consistency with other axioms trivially results from the assumption of total ordering on P.

LEMMA 1. *Let $X, Y : \Omega \rightarrow \mathbb{R}$ be random variables, and $tl, tr, tx, ty : \mathbb{R} \rightarrow \mathbb{R}$ be performance observation transformation functions. Then the following holds:*

$$(\forall o \in \mathbb{R} : tl(o) \leq tr(o)) \rightarrow E(tl(X)) \leq E(tr(X))$$

$$(E(tx(X)) \leq E(ty(Y)) \wedge E(ty(Y)) \leq E(tx(X))) \leftrightarrow$$
$$E(tx(X)) = E(ty(Y))$$

[2]Note that the effect of the performance observation transformation function on distribution parameters is potentially complex. We assume that in practical applications, the performance observation transformation functions will be limited to linear shift and scale.

PROOF. The validity of the first formula follows from the definition of the expected value. Let $f(x)$ be the probability density function of random variable X. Since $f(x) \geq 0$, it holds that

$$E(tl(X)) = \int_{-\infty}^{\infty} tl(x)f(x)dx \leq \int_{-\infty}^{\infty} tr(x)f(x)dx = E(tr(X))$$

The validity of the second formula follows naturally from the properties of total ordering on real numbers. □

Note that we assumed M to be a continuous random variable. The proof would be the same for a discrete random variable, except with a sum in place of the integral.

THEOREM 1. *The interpretation of performance relations $\leq_p$ and $=_p$, as given by Definition 5, is consistent with axioms (4) and (5).*

PROOF. The proof of the theorem naturally follows from Lemma 1 by substituting $P_M(x_1, \ldots, x_m)$ for X and $P_N(y_1, \ldots, y_n)$ for Y. □

While the above interpretation illustrates the idea behind SPL, it assumes that the expected value $E(tl(X))$ can be computed. Unfortunately, this assumption hardly ever holds, because the distribution function of X is typically unknown, and so is the expected value. While it is possible to measure durations of method invocations for the purpose of method performance comparison, the type of the distribution and its parameters remain unknown.

4. SAMPLE-BASED INTERPRETATION

To avoid the problem with unknown distribution function and the expected value of a random variable, we turn to sample based methods that work with parameter estimates derived from measurements. This leads us to a *sample-based interpretation* of SPL that relies solely on the observations of random variables. Due to limited applicability of the expected-value-based interpretation, we will only deal with the sample-based interpretation in the rest of the paper.

The basic idea is to replace the comparison of expected values in the interpretation of $\leq_p$ and $=_p$ by a statistical test. Given a set of observations of method performances (i.e. random variables), the test will allow us to determine whether the mean values of the observed method performances are in a particular relation (i.e., $\leq_p$ or $=_p$).

However, to formulate the sample-based interpretation, we first need to fix the set of observations for which the relations will be interpreted. We therefore define an *experiment*, denoted $\mathcal{E}$, as a finite set of observations of method performances under a particular method workload.

Definition 6. Experiment $\mathcal{E}$ is a collection of $\mathcal{O}_{P_M(p_1,\ldots,p_m)}$, where $\mathcal{O}_{P_M(p_1,\ldots,p_m)} = \{P_M^1(p_1, \ldots, p_m), \ldots, P_M^V(p_1, \ldots, p_m)\}$ is a set of V observations of method performance P_M subjected to workload $L_M^{p_1,\ldots,p_m}$, and where $P_M^i(p_1, \ldots, p_m)$ denotes i-th observation of performance of method M.

Having established the concept of an experiment, we can now define the sample-based interpretation of SPL (note that it depends on a particular experiment).

The interpretation is the same as given in Section 3, only Definition 7 is used to assign semantics to method performance relations.

Definition 7. Let $tm, tn : \mathbb{R} \to \mathbb{R}$ be performance observation transformation functions, P_M and P_N be method performances, $x_1, \ldots, x_m, y_1, \ldots, y_n$ be workload parameters, and $\alpha \in \langle 0, 0.5\rangle$ be a fixed significance level.

For a given experiment $\mathcal{E}$, the relations $\leq_{p(tm,tn)}$ and $=_{p(tm,tn)}$ are interpreted as follows:

- $P_M(x_1, \ldots, x_m) \leq_{p(tm,tn)} P_N(y_1, \ldots, y_n)$ iff the null hypothesis

  $\mathrm{H}_0 : E(tm(P_M^i(x_1, \ldots, x_m))) \leq E(tn(P_N^j(y_1, \ldots, y_n)))$

  cannot be rejected by one-sided Welch's t-test [11] at significance level α based on the observations gathered in the experiment $\mathcal{E}$;

- $P_M(x_1, \ldots, x_m) =_{p(tm,tn)} P_N(y_1, \ldots, y_n)$ iff the null hypothesis

  $\mathrm{H}_0 : E(tm(P_M^i(x_1, \ldots, x_m))) = E(tn(P_N^j(y_1, \ldots, y_n)))$

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

where $E(tm(P_M^i(\ldots)))$ and $E(tn(P_N^j(\ldots)))$ denote the mean value of performance observations transformed by function tm or tn, respectively.

Briefly, the Welch's t-test rejects with significance level α the null hypothesis $\overline{X} = \overline{Y}$ against the alternative hypothesis $\overline{X} \neq \overline{Y}$ if

$$\left| \frac{\overline{X} - \overline{Y}}{\sqrt{\frac{S_X^2}{V_X} + \frac{S_Y^2}{V_Y}}} \right| > t_{\nu,\alpha/2}$$

and rejects with significance level α the null hypothesis $\overline{X} \leq \overline{Y}$ against the alternative hypothesis $\overline{X} > \overline{Y}$ if

$$\frac{\overline{X} - \overline{Y}}{\sqrt{\frac{S_X^2}{V_X} + \frac{S_Y^2}{V_Y}}} > t_{\nu,\alpha}$$

where V_i is the sample size, S_i^2 is the sample variance, $t_{\nu,\alpha}$ is the $(1-\alpha)$-quantile of the Student's distribution with ν levels of freedom, with ν computed as follows:

$$\nu = \frac{\left(\frac{S_X^2}{V_X} + \frac{S_Y^2}{V_Y}\right)^2}{\frac{S_X^4}{V_X^2(V_X-1)} + \frac{S_Y^4}{V_Y^2(V_Y-1)}}$$

Although Welch's t-test formally requires normal distribution of X and Y, it is robust to violations of normality due to the Central Limit Theorem.

As in Section 3, we need to show that the sample-based interpretation of SPL is consistent with axioms (4) and (5).

THEOREM 2. *The interpretation of relations $\leq_p$, and $=_p$, as given by Definition 7, is consistent with axiom (4) for a given fixed experiment $\mathcal{E}$.*

PROOF. For sake of brevity, we will denote the sample mean $E(tl(P_M^i(x_1, \ldots, x_m)))$ as $\overline{X}_{tl}$ and the sample variance $\mathrm{Var}(tl(P_M^i(x_1, \ldots, x_m)))$ as S_{tl}^2; $\overline{X}_{tr}$ and S_{tr}^2 are defined in a similar way.

Assuming $\forall o \in \mathbb{R} : tl(o) \leq tr(o)$, we have to prove that the null-hypothesis $H_0 : \overline{X}_{tl} \leq \overline{X}_{tr}$ cannot be rejected by the Welch's t-test.

Based on the formulation of the t-test, it means that the null-hypothesis can be rejected if

$$\frac{\overline{X}_{tl} - \overline{X}_{tr}}{\sqrt{\frac{S^2_{tl}}{V} + \frac{S^2_{tr}}{V}}} > t_{\nu,\alpha}$$

where V is the number of samples $P^i_M(x_1, \ldots, x_m)$ in the experiment $\mathcal{E}$.

Since the denominator is a positive number, the whole fraction is non-positive. However, the right hand side $t_{\nu,\alpha}$ is a non-negative number since we assumed that $\alpha \leq 0.5$. This means that the inequality never holds and thus the null-hypothesis cannot be rejected. □

THEOREM 3. *The interpretation of relations $\leq_p$, and $=_p$, as given by Definition 7, is consistent with axiom (5) for a given fixed experiment $\mathcal{E}$.*

PROOF. For sake of brevity, we will denote the sample mean $E(tx(P^i_X(x_1, \ldots, x_m)))$ as $\overline{X}$ and the sample variance $\mathrm{Var}(tx(P^i_X(x_1, \ldots, x_m)))$ as S^2_X; $\overline{Y}$ and S^2_Y are defined in a similar way.

By interpreting axiom (5) according to Definition 7, we get the following statements:

$$P_M(x_1, \ldots, x_m) \leq_{p(tm,tn)} P_N(y_1, \ldots, y_n) \longleftrightarrow \frac{\overline{X} - \overline{Y}}{\sqrt{\frac{S^2_X}{V_X} + \frac{S^2_Y}{V_Y}}} \leq t_{\nu_{X,Y},\alpha}$$

$$P_N(y_1, \ldots, y_n) \leq_{p(tn,tm)} P_M(x_1, \ldots, x_m) \longleftrightarrow \frac{\overline{Y} - \overline{X}}{\sqrt{\frac{S^2_Y}{V_Y} + \frac{S^2_X}{V_Y}}} \leq t_{\nu_{Y,X},\alpha}$$

$$P_M(x_1, \ldots, x_m) =_{p(tm,tn)} P_N(y_1, \ldots, y_n) \longleftrightarrow -t_{\nu_{X,Y},\alpha} \leq \frac{\overline{X} - \overline{Y}}{\sqrt{\frac{S^2_X}{V_X} + \frac{S^2_Y}{V_Y}}} \leq t_{\nu_{X,Y},\alpha}$$

Thus, we need to show that

$$\frac{\overline{X} - \overline{Y}}{\sqrt{\frac{S^2_X}{V_X} + \frac{S^2_Y}{V_Y}}} \leq t_{\nu_{X,Y},\alpha} \wedge \frac{\overline{Y} - \overline{X}}{\sqrt{\frac{S^2_Y}{V_Y} + \frac{S^2_X}{V_Y}}} \leq t_{\nu_{Y,X},\alpha} \longleftrightarrow -t_{\nu_{X,Y},\alpha} \leq \frac{\overline{X} - \overline{Y}}{\sqrt{\frac{S^2_X}{V_X} + \frac{S^2_Y}{V_Y}}} \leq t_{\nu_{X,Y},\alpha}$$

This holds, because $\nu_{X,Y} = \nu_{Y,X}$ and thus

$$\frac{\overline{Y} - \overline{X}}{\sqrt{\frac{S^2_Y}{V_Y} + \frac{S^2_X}{V_Y}}} \leq t_{\nu_{Y,X},\alpha} \longleftrightarrow -t_{\nu_{X,Y},\alpha} \leq \frac{\overline{X} - \overline{Y}}{\sqrt{\frac{S^2_X}{V_X} + \frac{S^2_Y}{V_Y}}}$$

□

Note that, as indicated in Section 2, the transitivity (i.e. $(P_X(\ldots) \leq_{p(tx,ty)} P_Y(\ldots) \wedge P_Y(\ldots) \leq_{p(ty,tz)} P_Z(\ldots)) \rightarrow P_X(\ldots) \leq_{p(tx,tz)} P_Z(\ldots)$) does not hold for the sample-based interpretation. This can be shown by considering the following observations and performing single-sided tests at significance level $\alpha = 0.05$: $\mathcal{O}_{P_X} = \{2, 4\}$, $\mathcal{O}_{P_Y} = \{-1, 1\}$, $\mathcal{O}_{P_Z} = \{-4, -2\}$.

5. CORRECTNESS OF EVALUATION

The valuation of the relations $=_p$ and $\leq_p$ in the sample-based interpretation of an SPL formula depends on the results of statistical tests applied to the samples of method performance collected during an experiment. Each statistical test performed to determine the valuation of a particular relation in the SPL formula may introduce either a Type I (true null hypothesis rejected) or Type II (false null hypothesis not rejected) error. As a consequence, the valuation of a formula in SPL with sample-based interpretation is correct only with some probability.

This probability could be estimated from the probabilities of introducing a Type I or Type II error in each test. For Type I error, the probability equals the test significance level and is 2α for $=_p$ and α for $\leq_p$. For Type II error, the probability is unknown and cannot be determined. It is therefore impossible, in general, to calculate the probability that the valuation of a formula is incorrect.

In some cases, for example when the valuation of a formula only depends on a rejection of a particular test, an estimate of Type I error based on the significance level can be made. However, when interested in the error potentially introduced by the evaluation of the whole formula, the cumulative nature of errors introduced by individual tests must be accounted for. For example, when the formula

$$P_M(10) \leq_{p(id,id)} P_N(10) \wedge P_M(50) \leq_{p(id,id)} P_N(50)$$

evaluates to "false", it is possible that the first, or the second, or both tests introduced a Type I error into the evaluation. Conversely, when the formula evaluates to "true", some of the tests may have introduced a Type II error. These two types of errors may also occur at the same time, for example when a formula contains terms with and without negation.

To bound the probability of incorrect evaluation, we define the set of relations needed for the formula evaluation:

Definition 8. A set of performance relations $\mathcal{R}$ is a set of all tuples $\langle \diamond, tm, tn, P_M(\dot{x}_1, \ldots, \dot{x}_m), P_N(\dot{y}_1, \ldots, \dot{y}_n) \rangle$, where $\diamond$ is either $=_p$ or $\leq_p$ and $\dot{x}_i$, $\dot{y}_j$ are particular fixed workload parameter values.

Definition 9. For a given formula F, we define the *evaluation skeleton* as a partial function $\mathcal{S}_F : \mathcal{R} \rightarrow \{True, False\}$, such that F can be evaluated using just the valuations given by $\mathcal{S}_F$. The skeleton is minimal in the sense that no relation can be removed from $\mathcal{S}_F$ without breaking the skeleton property of being sufficient to evaluate formula F.

Definition 10. For a given formula F, we define $\mathcal{S}^{True}_F$ as a set of all evaluation skeletons under which the formula evaluates to "true". Similarly $\mathcal{S}^{False}_F$ is a set of all evaluation skeletons under which the formula evaluates to "false".

The probability that a formula has been evaluated incorrectly can be then estimated using the idea behind Bonferroni correction [12]. For example, when a formula evaluates to "true", we can bound the probability P(F is "true" | F does not hold) by summing up α (i.e. Type I error probability) or β (i.e. Type II error probability) for all tests that may be needed for the evaluation.

Since only α is known, we can effectively bound the probability only for formulas that are evaluated solely as a result of test rejections. For example, if the formula $P_M(10) \leq_{p(id,id)}$

$P_N(10) \wedge P_M(50) \leq_{p(id,id)} P_N(50)$ is "false", we know that this valuation was based only on test rejections.

This is formalized by the following theorem.

THEOREM 4. *Let F be an SPL-formula. If every evaluation of formula F to "true" is based only on rejections (i.e. $\forall \mathcal{S}_F \in \mathcal{S}_F^{True} : range(\mathcal{S}_F) = \{False\}$), then*

$$P(F \text{ is "true"} \mid F \text{ does not hold}) \leq \sum_{\forall R \in \mathcal{R}_F^{True}} P_S(R)$$

where $\mathcal{R}_F^{True} = \bigcup_{\mathcal{S}_F \in \mathcal{S}_F^{True}} domain(\mathcal{S}_F)$ is the set of all relations that may be used for evaluating the formula to "true", and $P_S(R)$ is the significance level for a test, i.e. 2α for $=_p$, and α for $\leq_p$.

Similarly, if every evaluation of formula F to "false" is based only on rejections,

$$P(F \text{ is "false"} \mid F \text{ holds}) \leq \sum_{\forall R \in \mathcal{R}_F^{False}} P_S(R)$$

where $\mathcal{R}_F^{False} = \bigcup_{\mathcal{S}_F \in \mathcal{S}_F^{False}} domain(\mathcal{S}_F)$.

PROOF. The probability P(F is "true" | F does not hold), may be bounded by the probability that at least one of the tests from $\mathcal{R}_F^{True}$ gave an incorrect Type I result (the test rejected a true null hypothesis). This probability can bound using Boole's inequality as:

$$P\left(\bigcup_{R \in \mathcal{R}_F^{True}} Err(R)\right) \leq \sum_{R \in \mathcal{R}_F^{True}} P(Err(R))$$

where $Err(R)$ denotes an event "Type I error occurred in test for R". This inequality holds even for tests which are not statistically independent.

The bound for the probability P(F is "false" | F holds) can be proved in the same way. □

Although Theorem 4 is only applicable to a special class of formulas, it appears sufficient for practical use. Expressing simple assumptions (along the lines presented in the examples) using SPL will result in formulas simple enough to allow calculating the probability of incorrect evaluation.

Considering Examples 1 and 2, when the formula evaluates to "false", we can calculate the probability of correct evaluation based solely on the knowledge of α. In fact, the probability is $(1 - 10\alpha)$ for both cases.

It is also possible to transform the formulas to similar dual formulas for which the probability of correctness can be determined when they evaluate to "true". This can be done by using strict inequalities instead of non-strict ones, which will reverse the null hypothesis, so the underlying test will need to reject the null hypothesis to be considered successful. Since the strict inequality is used as a "syntactic sugar" for the negation of the opposite non-strict inequality,

$$P_M <_{p(tm,tn)} P_N \leftrightarrow \neg(P_M \geq_{p(tm,tn)} P_N)$$
$$P_M >_{p(tm,tn)} P_N \leftrightarrow \neg(P_M \leq_{p(tm,tn)} P_N),$$

the formula with strict inequalities can be simply converted to the corresponding SPL formula which only uses the non-strict inequalities.

Examples 1 and 2 can be thus rewritten to Examples 3 and 4 (we only show the first step of the conversion) as follows:

Example 3. "On arrays of 100, 500, 1000, 5000, and 10000 elements, the sorting algorithm A less than 5% faster and less than 5% slower compared to sorting algorithm B"

$$\forall n \in \{100, 500, 1000, 5000, 10000\} :$$
$$P_A(n) >_{p(id,\lambda x.0.95x)} P_B(n) \wedge P_A(n) <_{p(id,\lambda x.1.05x)} P_B(n)$$

The formula above therefore corresponds to the following SPL formula:

$$\forall n \in \{100, 500, 1000, 5000, 10000\} :$$
$$\neg(P_A(n) \leq_{p(id,\lambda x.0.95x)} P_B(n) \vee P_A(n) \geq_{p(id,\lambda x.1.05x)} P_B(n))$$

Example 4. "On buffers of 256, 1024, 4096, 16384, and 65536 bytes, the Rijndael encryption algorithm is more than 10% faster than the Blowfish encryption algorithm and less than 200 times slower than array copy."

$$\forall n \in \{256, 1024, 4096, 16384, 65536\} :$$
$$P_{\text{Rijndael}}(n) <_{p(id,\lambda x.0.9x)} P_{\text{Blowfish}}(n) \wedge$$
$$P_{\text{Rijndael}}(n) <_{p(id,\lambda x.200x)} P_{\text{ArrayCopy}}(n)$$

6. FITNESS FOR PURPOSE

Before moving on to discuss the automated evaluation of SPL formulas, we need to answer an obvious question: for which kind of program methods is the SPL approach suitable? The answer clearly depends on the choice of SPL interpretation. So far, we have introduced interpretations that are based on comparing the mean value of method performance, i.e. the location estimator of the underlying distribution. Therefore, the SPL approach should be well-suited for methods whose performance can be reasonably described by a mean value, i.e. the underlying distribution is unimodal, without heavy tails. We expect such methods to be relatively small, often representing the computational kernel of an application, handling (bulk) data transformations and processing. We believe that for such methods, most developers will be able to intuitively understand the concept of method performance, identify factors influencing it, and possibly express performance assumptions by comparing it (in relative terms) to performance of other (similar) methods.

Although the mean value can be calculated even in the case of multi-modal and heavy-tailed distributions, it does not represent the essential characteristics of the distribution very well. Such performance data are difficult to interpret, and we are not confident that a developer will be able to intuitively understand the performance of a method with such complex behavior — let alone express performance assumptions by comparison with other methods.

To evaluate the fitness of SPL for performance comparisons, we have therefore conducted experiments with two sets of simple methods that fall into the (loose) category of computational kernels. One set implements two sort algorithms, and the other implements various encryption algorithms, including null encryption (memory copying). The experiments correspond to the examples introduced in Sections 2 and 5.

All experiments were run on a 64-bit platform[3], executing on a single core within Oracle JVM 7 on top of Fedora Linux[4], with all non-essential system services disabled. We

[3]Dell OptiPlex 780, Intel Core 2 Quad Q9550 CPU at 2.83 GHz, 4 GiB DDR3 RAM at 1066 MHz

[4]Fedora 15, Linux Kernel 2.6.40.4-5.fc15, GLIBC 2.14, JRE 1.7.0-b147

collected the durations of individual method invocations, with new random input generated before each invocation. The data from the first 15000 invocations were discarded, intended only to warm up the system and let the JIT compiler optimize the code. The next 100000 invocations provided samples of method performance for each method under test.

As per the sample-based SPL interpretation, all performance relations are evaluated using Welch's one-sided t-test. Results presented below show the evaluation of SPL formulas on experimental data with the underlying tests performed at significance level $\alpha = 0.01$.

6.1 "Similar performance" assumptions

The first set of experiments covers Examples 1 and 3, both assuming that the performance of two methods is similar, i.e. the performance difference is bounded.

We evaluate the SPL formula from Example 1 in two scenarios. In the first one, we compare (the performance of) two different methods with an identical implementation of the Insertion Sort algorithm, and expect the formula to evaluate to *true*. In the second case, we compare methods implementing the Insertion Sort and Dual Pivot Quick Sort algorithms, and expect the formula to evaluate to $false$.

$n = 100$	$n = 500$	$n = 1000$	$n = 5000$	$n = 10^4$	$\alpha = 0.01$
$P_{\text{Insertion}}(n) \geq_{p(id,\lambda x.0.95x)} P_{\text{Insertion}}(n)$					$true \wedge true$
0.969	1	1	1	1	
$P_{\text{Insertion}}(n) \leq_{p(id,\lambda x.1.05x)} P_{\text{Insertion}}(n)$					
0.999	1	1	1	1	
$P_{\text{Insertion}}(n) \geq_{p(id,\lambda x.0.95x)} P_{\text{DualPivot}}(n)$					$true \wedge false$
1	1	1	1	1	
$P_{\text{Insertion}}(n) \leq_{p(id,\lambda x.1.05x)} P_{\text{DualPivot}}(n)$					
0	0	0	0	0	

Table 1: Test results and p-values for Example 1

The results of the experiment are shown in Table 1. For each relation, the table presents the p-value of the t-test applied to observations of method performance under a particular workload. A performance relation will evaluate to *true* if none of the tests rejects the null hypothesis. Conversely, if any of the tests rejects the null hypothesis (p-value $< \alpha$), the relation evaluates to $false$. The final column combines the evaluation of individual relations into the final result.

In both cases, the results correspond to the expectation, but there is a problem with the *true* evaluation in the first case. When an expression written in standard logic holds, we are not used to question the result. In SPL with sample-based interpretation (i.e. statistical testing) and this particular formulation of assumptions, we have no indication as to how "strong" the *true* evaluation is — while the tests did not reject the null hypotheses, they did not confirm them.

As a remedy, in Section 5 the formula from Example 1 was rewritten to only evaluate to *true* if all the tests reject the null hypothesis. Formulated as in Example 3, it not only provides "stronger" answers, but also enables estimating the probability that the answer is wrong (Type I error).

The results of evaluating the modified SPL formula on the same data are shown in Table 2. Unlike in the previous case, a relation evaluates to *true* only if the null hypothesis is rejected for all workloads. While the result of comparison between Selection Sort and Dual Pivot Quick Sort did not change, the formula relating the performance of two identical implementations of Selection Sort now evaluates to $false$.

$n = 100$	$n = 500$	$n = 1000$	$n = 5000$	$n = 10^4$	$\alpha = 0.01$
$P_{\text{Insertion}}(n) > e_{p(id,\lambda x.0.95x)} P_{\text{Insertion}}(n)$					$false \wedge true$
0.031	0	0	0	0	
$P_{\text{Insertion}}(n) <_{p(id,\lambda x.1.05x)} P_{\text{Insertion}}(n)$					
5.555e-05	0	0	0	0	
$P_{\text{Insertion}}(n) >_{p(id,\lambda x.0.95x)} P_{\text{DualPivot}}(n)$					$true \wedge false$
0	0	0	0	0	
$P_{\text{Insertion}}(n) <_{p(id,\lambda x.1.05x)} P_{\text{DualPivot}}(n)$					
1	1	1	1	1	

Table 2: Test results and p-values for Example 3

The p-value of the failing test indicates that a true null hypothesis could be rejected with probability 0.031, which is too much for $\alpha = 0.01$, but would be sufficient for $\alpha = 0.05$. The test failed to reject the null hypothesis due to outliers in the data. Unfortunately, even though they are present in all measurements, they have bigger impact on shorter durations measured with smaller workloads.

At this point, even though the performance assumption had failed, the developer has obtained a more informative result and has several options. By analysing the cause of the failure, the developer can conclude that there is too much interference during measurement and either relax the significance level α, avoid measurements on small workloads, improve measurement accuracy [13], or filter the outliers from the measurement.

In this particular case, if the developer chose to avoid workloads with array of size 100, the formula would evaluate to *true* at significance level $\alpha = 0.01$ even if the interval for "considered similar" difference in performance was reduced to 1% from the current (liberal) 10%.

6.2 "Different performance" assumptions

The second set of experiments covers Examples 2 and 4, both making statements about relative performance of three methods.

As in the previous experiment, we first evaluate the SPL formula from Example 2, which assumes the Rijndael encryption algorithm to be at most 200× slower than array copy, but still at least 10% faster than the Blowfish encryption algorithm.[5]

$n = 2^8$	$n = 2^{10}$	$n = 2^{12}$	$n = 2^{14}$	$n = 2^{16}$	$\alpha = 0.01$
$P_{\text{Rijndael}}(n) \leq_{p(id,\lambda x.0.9x)} P_{\text{Blowfish}}(n)$					$true \wedge false$
1	1	1	1	1	
$P_{\text{Rijndael}}(n) \leq_{p(id,\lambda x.200x)} P_{\text{ArrayCopy}}(n)$					
1	1.444e-73	0	0	1	
$P_{\text{Rijndael}}(n) \leq_{p(id,\lambda x.275x)} P_{\text{ArrayCopy}}(n)$					$true$
1	1	1	1	1	

Table 3: Test results and p-values for Example 2

The results of the experiment are shown in Table 3. As in the experimental evaluation of Example 1, a relation evaluates to *true* if non of the tests rejects the null hypothesis.

The first part of the formula regarding the relative performance of the Rijndael and the Blowfish algorithms evaluates to *true*. The second part, regarding the relative performance of the Rijndael algorithm vs. array copy (null encryption), evaluates to $false$, because the underlying test rejects the null hypothesis in 3 out of 5 cases. To a developer, this

[5]Both algorithms operated in CBC mode, using 128-bit keys, and the implementation used was provided by the default (Oracle) provider of Java Cryptography Extensions.

would mean that he either overestimated the performance of the Rijndael encryption algorithm during formulation of the assumption, or that the formula does not hold on this particular platform.

Since we know that we have only guessed at the relative performance of Rijndael vs. array copy, we can use a more conservative estimate and assume that the Rindael algorithm is, say, at most 275× slower. Under this assumption, the second part of the formula will evaluate to *true* (double row at the bottom of Table 3), as will the whole formula.

However, like in the previous experiment, we have no indication as to "how true" is the *true* the formula evaluates to. Again, as a remedy, we rewrite the formula as per Example 4, to obtain a variant that will only evaluate to *true* if all the tests reject the null hypothesis.

$n = 2^8$	$n = 2^{10}$	$n = 2^{12}$	$n = 2^{14}$	$n = 2^{16}$	$\alpha = 0.01$
$P_{\text{Rijndael}}(n) <_{p(id,\lambda x.0.9x)} P_{\text{Blowfish}}(n)$					*true* ∧ *true*
0	0	0	0	0	
$P_{\text{Rijndael}}(n) <_{p(id,\lambda x.275x)} P_{\text{ArrayCopy}}(n)$					
0	0	1.988e-52	0	0	

Table 4: Test results and p-values for Example 4

The results of evaluating the modified SPL formula on the same data are shown in Table 4. Again, like in the experimental evaluation of Example 3, a relation evaluates to *true* only if the null hypothesis is rejected for all workloads.

Since we have only tested the more conservative assumption regarding the relative performance of the Rijndael algorithm and array copy, the entire formula evaluates to *true*. Unlike in the previous case though, we know that the resulting *true* is "fairly strong", because only one of the tests had non-zero p-value, and even that was practically zero.

7. EFFICIENT FORMULA EVALUATION

While writing SPL formulas is relatively easy, evaluating them requires collecting significant amount of performance data. In many cases, evaluating a single performance relation or a single test may decide the value of the whole formula, rendering other tests and relations irrelevant, and time put into collecting performance data wasted.

To enable efficient evaluation of SPL formulas in automated performance testing, we have applied the idea of SMT-solving [14, 15, 16] to solving SPL formulas. The basic idea is to solve the propositional part of a formula using a regular SAT solver, and delegate the non-propositional predicates to a specialized decision procedure. In case of SPL, the decision procedure is responsible for collecting performance data and applying the statistical tests to evaluate performance predicates. This approach allows to only collect performance data demanded by the solving algorithm and avoid measurements for predicates that do not influence the value of a formula.

Before describing the actual SPL-evaluation algorithm, we first need to define the concepts of a *propositional skeleton* and *satisfiability-irrelevant variable set*:

Definition 11. For a quantifier-free SPL formula F we define its *propositional skeleton* as a propositional-logic formula F_S, where each occurrence of a performance-relation predicate (i.e., $P_M(\dot{x}_1, \ldots, \dot{x}_m) \leq_{p(tn,tm)} P_N(\dot{y}_1, \ldots, \dot{y}_n)$ or $P_M(\dot{x}_1, \ldots, \dot{x}_m) =_{p(tn,tm)} P_N(\dot{y}_1, \ldots, \dot{y}_n)$) is replaced by a variable $W_{P_M(\dot{x}_1,\ldots,\dot{x}_m) \leq_{p(tn,tm)} P_N(\dot{y}_1,\ldots,\dot{y}_n)}$, $W_{P_M(\dot{x}_1,\ldots,\dot{x}_m) =_{p(tn,tm)} P_N(\dot{y}_1,\ldots,\dot{y}_n)}$ respectively.

Note that since SPL only allows quantifiers over finite subsets of P, any SPL formula can be transformed to a quantifier-free SPL formula by expanding all quantifiers into finite conjuction (universal) or finite disjunction (existential).

The unsatisfiability of a propositional skeleton implies unsatisfiability of the associated SPL formula (the opposite does not hold). Additionally, the satisfiability of an SPL formula implies satisfiability of its propositional skeleton.

Definition 12. Having a propositional formula F and its satisfying partial valuation[6] V_P, then a *satisfiability-irrelevant* set R_{SI} is a subset of all propositional variables of F such that all the possible valuations of R_{SI} combined with V_P yield a satisfying valuation of F.

SPL-solving algorithm. For a given formula, the algorithm uses a SAT solver to obtain a valuation of its propositional skeleton and checks the feasibility of the skeleton valuation by evaluating the associated performance predicates. If the skeleton valuation is infeasible (i.e., the valuation of a performance predicate given by the decision procedure differs from the valuation of the associated skeleton variable), another valuation is obtained from the SAT solver. The results of the decision procedure are stored and taken into account in subsequent runs of the SAT solver, thus eliminating the infeasible valuations and "locking" the matching valuations. This is repeated either until the valuation of all performance predicates is validated, or until the skeleton, in combination with the stored results, becomes unsatisfiable.

In contrast to SMT-solving, the aim of SPL-solving is to (preferably) only evaluate performance predicates necessary for deciding the satisfiability of a formula (recall evaluation skeleton from Definition 9). Therefore, while checking the feasibility of a skeleton valuation, we identify the satisfiability-irrelevant set with respect to this valuation and consider only the relevant variables. This allows us to skip evaluation of a (potentially large) number of performance predicates. The satisfiability-irrelevant set is constructed incrementally. Before running the decision procedure for a particular skeleton variable, the variable is tested for inclusion in the current version of the satisfiability-irrelevant set. However, since the decision procedure can reject the current skeleton valuation, it is necessary to rebuild this set accordingly.

An outline of the SPL-solving algorithm is shown in Figure 1. Before going into detail, we first describe the notation. For a given SPL formula F, the $MakeSkeleton$ function returns its propositional skeleton F_S and the set of all propositional variables substituted for performance predicates R. R_U is the set of all variables from R that have not yet been evaluated by the decision procedure and their valuations are thus undecided. R_{SI} is a satisfiability-irrelevant subset of variables from F_S. V_P is a partial valuation of F_S enforcing the results of the previous decision-procedure runs. The $SolveSAT$ function provides a temporary valuation V_{temp} of F_S, based on the partial valuation V_P. The tuple (var, val) denotes a variable from R and its valuation

[6] A *satisfying partial valuation* is a partial valuation that can be extended to a complete satisfying valuation.

in V_{temp}. For a variable var, the $IsSatIr$ function decides whether $R_{SI} \cup \{var\}$ is satisfiability-irrelevant for formula F_S and partial valuation V_{SIP}. $MeasureAndTest$ is the decision procedure for performance predicates (as described in Section 4), and m is the result of the procedure (i.e., true or false).

```
 1: F_S, R ← MakeSkeleton(F)
 2: R_U ← R, R_SI ← ∅, V_P ← ∅
 3: V_temp ← SolveSAT(F_S, V_P)
 4: if V_temp = false then
 5:    return false
 6: end if
 7: for all (var, val) ∈ V_temp ∩ R_U do
 8:    V_SIP ← V_temp \ ({var} ∪ R_SI)
 9:    if IsSatIr(var, F_S, V_SIP, R_SI) then
10:       R_U ← R_U \ {var}
11:       R_SI ← R_SI ∪ {var}
12:    else
13:       m ← MeasureAndTest(var)
14:       V_P ← V_P ∪ {(var, m)}
15:       R_U ← R_U \ {var}
16:       if val ≠ m then
17:          R_U ← R_U ∪ R_SI, R_SI ← ∅
18:          goto line 3
19:       end if
20:    end if
21: end for
22: return true
```

Figure 1: SPL-solving algorithm

After the propositional skeleton F_S is created and the involved sets R_U, R_{SI}, as well as the partial valuation V_P are initialized (lines 1-2), a satisfying valuation of F_S combined with V_P is obtained from the SAT solver (line 3). If the SAT solver indicates that F_S combined with V_P is unsatisfiable, the algorithm returns "false" (lines 4-6), because it implies that the original SPL formula is unsatisfiable with respect to measurements dictating V_P. Otherwise, the algorithm sequentially processes valuations of all variables which were not yet checked by the decision procedure (line 7). Each variable is first tested for membership in the current version of R_{SI} with respect to the current skeleton valuation V_{temp} (lines 8-9). If the variable can be added into R_{SI}, then it is added (lines 10-11). Otherwise, it is necessary to call the decision procedure $MeasureAndTest$ (line 13). The result of the decision procedure is added to V_P to be enforced in the subsequent SAT solver runs (line 14). If the stored result conforms to the current skeleton valuation V_{temp}, the next variable is processed. Otherwise V_{temp} is infeasible with respect to the measurements and a new skeleton valuation has to be obtained from the SAT solver (lines 16-19). The new valuation also invalidates the current R_{SI} (line 17).

Correctness of the SPL-solving algorithm. The correctness of the algorithm results from the fact that the algorithm returns "false" only if the propositional skeleton itself is unsatisfiable or if enforcing the measurement-based valuations makes it unsatisfiable. Moreover, the algorithm returns "true" only in cases where the partial measurement-based valuation is satisfying and all the other variables form a satisfiability-irrelevant set (including the case of empty partial valuation when the formula is a tautology).

Identification of a satisfiability-irrelevant set. During SPL-solving, the variables are sequentially tested for membership in the (incrementally constructed) satisfiability-irrelevant set. For this, we provide a simple algorithm transforming the problem of deciding whether a given set is satisfiability-irrelevant to a formula-satisfiability problem, subsequently solved by a SAT solver. In fact, the resulting SAT formula mimics the Definition 12.

For a formula F, its partial valuation V_P, and the tested (potentially satisfiability-irrelevant) set R_{SI} the transformation yields the following auxiliary formula (let V_T and V_F represent the positively and negatively valuated variables of V_P, respectively, and $r_1 \dots r_n$ be the elements of R_{SI}):

$$\begin{gathered}(\wedge_{\forall x \in V_T} x) \ \wedge \ (\wedge_{\forall y \in V_F} \neg y) \ \wedge \\ F[r_1 \mapsto false, r_2 \mapsto false, \dots r_n \mapsto false] \ \wedge \\ F[r_1 \mapsto true, r_2 \mapsto false, \dots r_n \mapsto false] \ \wedge \\ F[r_1 \mapsto false, r_2 \mapsto true, \dots r_n \mapsto false] \ \wedge \\ \vdots \\ F[r_1 \mapsto true, r_2 \mapsto true, \dots r_n \mapsto true]\end{gathered}$$

where $F[r_1 \mapsto val_1, \dots, r_n \mapsto val_n]$ denotes a formula derived from F by substituting all occurrences of $r_1, \dots, r_n$ by the associated boolean constants $val_1, \dots, val_n$.

The first line of the auxiliary formula enforces the given partial valuation V_P, while the remaining lines capture all possible valuations of R_{SI}.

Decision procedure for performance predicates. Under the sample-based SPL interpretation (Section 4), the decision procedure evaluates performance predicates via statistical testing. For this it first effects collection of performance data from experiments in which the methods under test (two sides of a performance relation) are subjected to workload according to given performance parameters. The statistical test is applied to the performance data and the result of the test is returned as the result of the decision procedure.

7.1 Algorithm discussion

The above SPL-solving algorithm heuristically optimizes the number of evaluated performance predicates (and thus the number of performance measurements). This is important, since SPL formulas may have non-trivial Boolean structure, containing arbitrary combination of conjunctions and disjunctions. For example, if a method M uses two implementations A and B of a library function (selected according to environment settings), we may want to express that performance of M depends on performance of either A or B. This could be expressed by a formula similar to the following disjunction:

$$\begin{gathered}\big(P_M(\dot{n}) \geq_{p(id,\lambda x.c_{A1}x)} P_A(\dot{n}) \wedge P_M(\dot{n}) \leq_{p(id,\lambda x.c_{A2}x)} P_A(\dot{n})\big) \\ \vee \\ \big(P_M(\dot{n}) \geq_{p(id,\lambda x.c_{B1}x)} P_B(\dot{n}) \wedge P_M(\dot{n}) \leq_{p(id,\lambda x.c_{B2}x)} P_B(\dot{n})\big)\end{gathered}$$

where the coefficients c_{A1}, c_{A2}, c_{B1}, and c_{B2} capture the level of dependency. Disjunctions will be also introduced when using implication or equivalence.

The results of the heuristic depend on the skeleton valuation given by the SAT solver and the order in which the predicates are evaluated. However, the heuristic can be further improved, especially by exploiting the fact that the cost of

evaluation of performance predicates can differ depending on the methods under test and workload parameters — mainly because collection of performance data will take different time. For example, evaluating a predicate comparing sorting algorithms will be significantly faster for arrays of size 100 than arrays of size 10000. This leads to a slight modification of the initial problem — each performance predicate in an SPL formula F can be assigned a cost of its evaluation. While solving the propositional skeleton of F, identification of a satisfiability-irrelevant set containing expensive predicates may greatly reduce the run-time of the SPL-solving algorithm, since the most expensive performance predicates might not need to be evaluated.

This could be addressed by employing a SAT solver that provides the satisfiability-irrelevant set as a part of the satisfying valuation, and is able to return an optimal valuation with respect to a given valuation-cost function. For such a SAT solver, a cost function assigning to each positively or negatively-valuated skeleton variable the evaluation cost of the associated performance predicate, while assigning 0 to all the variables in the satisfiability-irrelevant set, would yield the desired valuation. Although optimizing SAT solvers do exist (for example MiniSAT [17]), the problem of returning a satisfiability-irrelevant set as a part of the result has not yet been satisfactorily addressed. Nevertheless, even the usage of a minimizing SAT solver does no guarantee the least total cost of the evaluated performance predicates, which provides room for futher investigation.

8. INTEGRATION WITH JAVA

We aim to use SPL in a setting similar to unit testing with JUnit or similar framework. A developer wishing to capture and periodically test performance assumptions should be able to do so by performing steps similar to writing unit tests. Specifically, we target the following use-cases. (UC1) The author of a method makes an assumption about its performance and wants to capture this assumption along with the method definition (i.e., provided nonfunctional property). (UC2) In code using another method, the author of the code makes an assumption about the performance of the used method and wants to capture this assumption along with the code (i.e., required nonfunctional property). (UC3) A third-party developer/tester has additional performance assumptions to those captured in the code; it is therefore necessary to capture and test these assumptions separately.

For the scenario denoted as UC1, we propose encoding SPL formulas using Java annotations. Based on these annotation, automated performance testing software can carry out measurements needed by the algorithm presented in Section 7 to evaluate the SPL formula. The `@SPL` annotation can be used to specify a set of SPL formulas for each method.

A simple application of the `@SPL` annotation is shown in Figure 2. In this case, the SPL formula requires that the execution time of the annotated method (referenced by the `SELF` keyword) be lower or equal to the execution time of method `g()` in class `Y` for two different types of input. The variable `n` in the SPL formula is used as a parameter for an implicit *input generator*[7] related to the annotated method. In straightforward cases like this one, input generators can be located using a naming conventions.

```
@SPL(formula = "for n {1,2} SELF(n) <= Y.g(n)")
int method(int[] parameter) { /* ... */ }
```

Figure 2: A simple application of `@SPL`.

Figure 3 illustrates how a custom input generator can be used (and reused). When an input generator is specified explicitly, the SPL evaluation framework will use the custom generator instead of the default one and use the supplied parameters to configure it. Each method can have multiple input generators.

```
@SPL(
    generators = "org.gen.Generator(parameter)",
    formula = "for n {1,2} SELF(n) <= Y.g(n)"
)
```

Figure 3: Defining a shared custom generator.

An advanced scenario with multiple generators is shown in Figure 4. Three different generators (with identifiers `gen[1-3]`) are defined and can be referenced within `formula`. The various forms of generator names provide the testing framework with information on creating generator instances.

```
@SPL(
    generators = {
        "gen1:gen.Gen1(parameter)",
        "gen2:gen.Gen2#factory(arg)",
        "gen3:gen.Gen3(parameter)#fact(arg)"
    },
    methods = {
        "methodX:pkg.AClass(parameter)#methodX",
        "methodY:pkg.JustStatic#methodY"
    },
    formula =
        "for j {1,2} k {1,2}" +
        "SELF[gen1](j) >= methodX[gen2](j) &" +
        "methodY[gen3](j,k) <=(x.10x,id)" +
            "SELF[gen3](j,k)"
)
```

Figure 4: A complex `@SPL` annotation example.

The example in Figure 4 also includes method references. When referencing non-static methods in other classes, these classes have to be instantiated first, which is the main purpose of the `method` parameter. Aliases defined in `methods` can be used instead of the fully qualified method names in `formula`.

As far as UC1 is concerned, performance annotations can be evaluated by a standalone SPL evaluation tool, which automatically evaluates all SPL-annotated methods in a given class or set of classes. UC2 can be addressed by taking an approach similar to UC1, i.e., by introducing SPL-annotated wrappers of the respective methods.

[7]Input generators are instances of `Iterable<Object[]>` that provide sequences of method inputs for performance measurements.

To address UC3, as well as complex cases of UC1 and UC2 (beyond the expressive power of annotations), we envision using an API inspired by JUnit, as shown in Figure 5.

```
void assertSPLOfMethod(
    Method method,
    Map<String, Entry<Object, Method>> methods,
    Map<String, InputGenerator> generators
);

void assertSPLFormula(
    String formula,
    Method method
);

void assertSPLFormula(
    String formula,
    Map<String, Entry<Object, Method>> methods,
    Map<String, InputGenerator> generators
);
```

Figure 5: SPL API based on JUnit.

The `assertSPLOfMethod()` method addresses the complex cases of UC1 and UC2 by evaluating the `@SPL` annotation of `method` in combination with the `methods` and `generators` parameters, which allows overriding the original method and generator definitions, and using locally initialized generators and method references. The `assertSPLFormula()` method family covers UC3, where the first version of `assertSPLFormula()` evaluates the supplied SPL `formula` using method and generator references obtained from the `@SPL` annotation of `method` and the second version uses explicitly provided `methods` and `generators` for the evaluation.

9. RELATED WORK

Current performance specification methods focus on analysis of previously-measured or run-time performance data (with the goal to verify given performance assertions). In contrast, SPL is targeting repetitive performance testing by first stating a performance assertion and then using the SPL-solving algorithm to obtain the measurements needed to decide satisfiability of the assertions. The following related performance-specification methods share this property.

PSpec [18] is a language for expressing performance assertions which targets similar goals – regression testing and code documentation. It uses absolute performance metrics (e.g., execution time) to capture the expected performance, which makes the formulas either non-portable, or too liberal. The performance data is collected from application logs.

PIP [19] is a similar approach exploiting declarative performance expectations with the goal of debugging behavior and performance of distributed systems. In contrast to SPL, PIP includes description of system behavior and the expected performance is declared with respect to such behavioral specification. Similar to PSpec, PIP uses application logs to obtain performance measurements and uses absolute performance metrics (such as CPU time, message latency) in performance expectations.

Performance assertions based on the PA language are introduced in [20]. Similar to our approach, the assertions are part of the source code. The assertions are checked at runtime and support local behavior adaptations based on the results. However, the method is not suitable for automated performance testing. The PA language provides access to various performance metrics (both absolute and relative) as well as key features of the architecture and user parameters.

Complementary to our method, in [21] the system-level performance expectations are descibed imperatively, using programmatic tests of globally measured performance data. The measurements are performed at run-time via injected probes and the data is analyzed continuously.

Similarly, [22] employs the A language to express validation programs concerning both business logic and performance characteristics (balanced CPU load) of Internet services. The method focuses mainly on run-time validation of operator actions and static configuration.

JUnitPerf [8] is an extension to the JUnit [3] framework, based on ideas from [18], It provides accurate time measurements and tests in the scope of a unit test. Similar to SPL, JUnitPerf targets automated performance testing while stressing simple usage scenarios. However, the performance assertions are not portable.

While we do not address the creation of SPL assertions in this work, there are also methods concerning specification of performance requirements at design-time. A process for constructing a performance-annotation model for UML is described in [23], with annotations based on the UML Profile for Schedulability, Performance, and Time specification. Such UML models can be then transformed to performance-analysis models based for example on Petri Nets [24].

An evolutionary methodology for performance-requirements specification is presented in [25]. It is based on refinement of the performance requirements during development. Further approaches for performance assessment and modeling in context of software architecture are compared in [26].

An important part of this work is the SPL-solving algorithm based on the idea of SMT-solving. Similar to classic SMT-solving techniques [14, 15], employed for example in Z3 [16], our approach uses a SAT solver for solving the propositional skeleton of the input formula. However, since the SPL-solving algorithm determines the measurements needed for deciding a given performance assertion, the number of evaluated performance predicates can be optimized by identifying a satisfiability-irrelevant set of propositional variables. This is not supported by the state-of-the-art SMT solvers, since there is no need to optimize the number of invocations of the underlying predicate-logic decision procedure.

10. CONCLUSION AND FUTURE WORK

Performance assertions and their analysis is an extensively-studied topic, yet so far in most cases, the assertions can be only expressed in absolute terms. Our goal is to enable the developer to express performance assumptions in simple, intuitive terms. To this end, we have introduced a novel method for describing performance assertions using Stochastic Performance Logic (SPL), which allows making statements about relative method performance in a platform independent way. Developer input is required in matters such as choosing the workload sizes relevant for the assumption, but providing such input appears to be more intuitive than having to guess the duration of method execution.

[8] http://clarkware.com/software/JUnitPerf.html

Our approach relies on statistical testing, and unless different interpretation is defined, it is well suited for a specific class of methods (computational and data transformation kernels) with simple behavior (performance-wise). The performance of such methods can be represented by an unimodal distribution of execution times for a given workload, and for which comparisons between location parameters such as mean value make sense. To facilitate efficient evaluation of SPL formulas in automated testing, we have presented an SPL solving algorithm that will drive the execution of experiments to collect performance data required to evaluate an SPL formula, possibly avoiding unnecessary measurements. To stress the practical impact of our approach, we have presented a set of annotations for the Java language that enable initial integration of SPL formulas with code.

We believe that these three components (the logic, the solving algorithm, and the annotation) provide a solid foundation for automated performance testing. Even though the foundations are solid, there is still a lot of room for improvement. In case of SPL, new theorems and axioms may be introduced, not only to show more complex properties of the logic, but also to guide the SAT solver in its search for satisfying valuations of SPL formulas. The SPL solving algorithm itself can be improved, provided a suitable optimizing SAT-solver can be found or developed. Finally, integration with code can be simplified by introducing more high-level annotations that will cover typical situations without the need to resorting to low-level SPL formulas.

11. ACKNOWLEDGMENTS

This work was partially supported by the Grant Agency of the Czech Republic project GACR P202/10/J042 and by the EU project ASCENS 257414.

12. REFERENCES

[1] K. Beck, *Test Driven Development: By Example*. 2002.

[2] B. Meyer, "Applying "Design by Contract"," *IEEE Computer*, vol. 25, 1992.

[3] P. Tahchiev, F. Leme, V. Massol, and G. Gregory, *JUnit in Action, 2nd Edition*. 2010.

[4] Google, "Googletest." `http://code.google.com/p/googletest/`.

[5] G. T. Leavens, C. Ruby, R. Leino, E. Poll, and B. Jacobs, "JML: Notations and Tools Supporting Detailed Design in Java," in *OOPSLA'00*, 2000.

[6] E. Cohen, W. Schulte, and S. Tobies, "A Practical Verification Methodology for Concurrent Programs," 2009.

[7] T. Kalibera and P. Tuma, "Precise Regression Benchmarking with Random Effects: Improving Mono Benchmark Results," in *Formal Methods and Stochastic Models for Performance Evaluation*, vol. 4054 of *LNCS*, Springer, 2006.

[8] T. Kalibera, J. Lehotsky, D. Majda, B. Repcek, M. Tomcanyi, A. Tomecek, P. Tuma, and J. Urban, "Automated Benchmarking and Analysis Tool," in *VALUETOOLS'06*, ACM, 2006.

[9] L. Bulej, T. Kalibera, and P. Tuma, "Repeated Results Analysis for Middleware Regression Benchmarking," *Perf. Evaluation*, vol. 60, 2005.

[10] H. P. Barendregt, *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, Amsterdam, 1984.

[11] B. L. Welch, "The Generalization of 'Student's' Problem when Several Different Population Variances are Involved," *Biometrika*, vol. 34, 1947.

[12] L. Wasserman, *All of Statistics: A Concise Course in Statistical Inference*. Springer Texts in Statistics, Springer, 2003.

[13] T. Kalibera and P. Tuma, "Precise Regression Benchmarking with Random Effects: Improving Mono Benchmark Results," in *Formal Methods and Stochastic Models for Performance Evaluation*, vol. 4054 of *LNCS*, Springer, 2006.

[14] C. Barrett, D. Dill, and A. Stump, "Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT," in *Computer Aided Verification*, vol. 2404 of *LNCS*, Springer, 2002.

[15] L. de Moura and N. Bjørner, "Satisfiability Modulo Theories: An Appetizer," in *Formal Methods: Foundations and Applications*, vol. 5902 of *LNCS*, Springer, 2009.

[16] L. de Moura and N. Bjorner, "Z3: An Efficient SMT Solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, vol. 4963 of *LNCS*, Springer, 2008.

[17] N. E'en and N. Sorensson, "Translating Pseudo-boolean Constraints into SAT," *J. on Satisfiability, Boolean Modeling and Computation*, 2006.

[18] S. E. Perl and W. E. Weihl, "Performance assertion checking," *SIGOPS Oper. Syst. Rev.*, vol. 27, 1993.

[19] P. Reynolds, C. Killian, J. L. Wiener, J. C. Mogul, M. A. Shah, and A. Vahdat, "Pip: Detecting the Unexpected in Distributed Systems," in *NSDI'06*, USENIX, 2006.

[20] J. S. Vetter and P. H. Worley, "Asserting Performance Expectations," in *Proc. 2002 ACM/IEEE Conf. on Supercomputing*, Supercomputing '02, IEEE CS, 2002.

[21] X. Liu, Z. Guo, X. Wang, F. Chen, X. Lian, J. Tang, M. Wu, M. F. Kaashoek, and Z. Zhang, "D3S: Debugging Deployed Distributed Systems," in *NSDI'08*, USENIX, 2008.

[22] A. Tjang, F. Oliveira, R. Bianchini, R. Martin, and T. Nguyen, "Model-Based Validation for Internet Services," in *Proc. 28th IEEE Intl. Symp. on Reliable Distributed Systems*, 2009.

[23] H. Du, R. Gan, K. Liu, Z. Zhang, and D. Booy, "Method for Constructing Performance Annotation Model Based on Architecture Design of Information Systems," in *Research and Practical Issues of Enterprise Information Systems II*, vol. 255 of *IFIP*, Springer, 2008.

[24] S. Distefano, D. Paci, A. Puliafito, and M. Scarpa, "UML Design and Software Performance Modeling," in *Computer and Information Sciences - ISCIS 2004*, vol. 3280 of *LNCS*, Springer, 2004.

[25] C.-W. Ho and L. Williams, "Developing Software Performance with the Performance Refinement and Evolution Model," in *WOSP'07*, ACM, 2007.

[26] M. A. Isa and D. N. A. Jawawi, "Comparative Evaluation of Performance Assessment and Modeling Method for Software Architecture," in *Software Engineering and Computer Systems*, vol. 181 of *CCIS*, Springer, 2011.