Product Optimization in Stepwise Design *

Don Batory[1], Jeho Oh[1], Ruben Heradio[2], and David Benavides[3]

[1] The University of Texas at Austin, Austin, TX, 78712, USA
{batory,jeho}@cs.utexas.edu
[2] Universidad Nacional de Educación a Distancia, Madrid, Spain
rheradio@issi.uned.es
[3] University of Seville, Seville, Spain benavides@us.es

Abstract. Stepwise design of programs is a divide-and-conquer strategy to control complexity in program modularization and theorems. It has been studied extensively in the last 30 years and has worked well, although it is not yet commonplace. This paper explores a new area of research, finding efficient products in colossal product spaces, that builds upon past work.

1 To My Friend Egon

Egon and I (Batory) first met at the "Logic for System Engineering" Dagstuhl Seminar on March 3-7, 1997. Egon presented his recent work on ***Abstract State Machines* (ASMs)** entitled "An Industrial Use of ASMs for System Documentation Case Study: The Production Cell Control Program". I presented my work on ***Database Management System* (DBMS)** customization via feature/layer composition. I had not yet directed my sights beyond DBMS software to software development in general.

At the heart of our presentations was the use and scaling of Dijkstra's concepts of layering and software virtual machines [15] and Wirth's notions of stepwise refinement [53]. The connection between our presentations was evident to us but likely no others. I was not yet technically mature enough to have a productive conversation with Egon then to explore our technical commonalities in-depth.

Our next encounter was at a Stanford workshop on Hoare's "Verifying Compiler Grand Challenge" in Spring 2006. Egon would make a point in workshop discussions and I would think: That is exactly what I would say! And to my delight as I learned later, Egon reacted similarly about my discussion points. At the end of the workshop, we agreed to explore interests and exchanged visits – I to Pisa and he to Austin. We wrote a joint paper [6] and presented it as a keynote at the June 2007 Abstract State Machine Workshop in June 2007. In doing so, I learned about his pioneering JBook case study [44]. Our interactions were a revelation to me as our thinking, although addressing related problems from very different perspectives, led us to similar world view.

* This work was partially funded by the EU FEDER program, the MINECO project OPHELIA (RTI2018-101204-B-C22), the TASOVA network (MCIU-AEI TIN2017-90644-REDT), and the Junta de Andalucia METAMORFOSIS project.

In this paper, I explain our source of commonality and where these ideas have been taken recently in my community, ***Software Product Lines* (SPLs)**.

2 Similarity of Thought in Scaling Stepwise Design

Central to the ***Stepwise Design* (SWD)** of large programs is the scaling of a step to an increment in program functionality. A program is a composition of such increments. To demonstrate that such technology is possible, one must necessarily focus on the SWD of a single application (as in JBook [44]) or a family of related applications (as in SPLs) where stereotypical increments in functionality can be reused in building similar programs. These increments are ***features***; think of features as the legos [49] of domain-specific software construction.

The JBook [44] presented a SWD of a suite of programs: a parser, **ASTs** (***Abstract Syntax Trees***), an interpreter, a compiler and a JVM for Java 1.0. At each step, there is a proof that the interpretation of *any* Java 1.0 program `P` and the compilation and then JVM execution of `P` produced identical results. The divide-and-conquer strategy used in JBook centered on the Java 1.0 grammar. The base language was the sublanguage of Java imperative expressions (`ExpI`). For this sublanguage, its grammar, ASTs, interpreter, compiler and JVM were defined, along with a proof of their consistency, Fig. 1a. Then imperative statements (`ΔStmI`) were added to `ExpI`, lock-step extending its grammar, ASTs, interpreter, compiler, JVM and proof of their composite consistency, Fig. 1b.[4] And then static fields and expressions (`ΔExpC`) were added, Fig. 1c, and so on until the complete syntax of Java 1.0 was formed, with its complete AST definitions, a complete interpreter, compiler and JVM for Java 1.0 too, Fig. 1d.

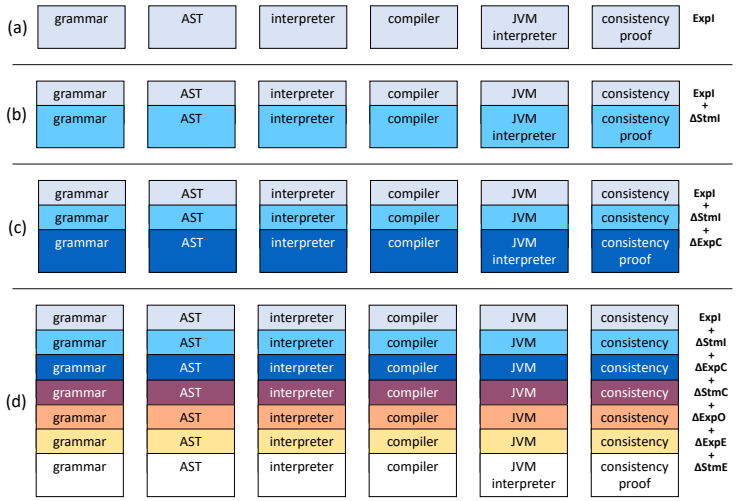

Fig. 1. SWD of JBook.

[4] All extensions were manually defined – this is normal.

The JBook was a masterful case study in SWD. It fit my SPL theory of features, where an application is defined by a set representations (programs, documents, property files, etc.). Features incrementally extend each representation so that they are consistent. Features could add new documents as well.

An SPL follows the JBook example, but with important differences. Some programs can have different numbers of features and different features can implement identical functionalities in different ways.[5] This enables a family of related programs to be built simply by composing features. Each program in an SPL is defined by a unique set of features. If there are `n` optional features, the size of the SPL's ***product space*** can be up to 2^n distinct programs/products.

It is well-known that features obey constraints: selecting one feature may demand the selection and/or exclusion of other features. And there is a preferred order in which features are composed. It was discovered that a context sensitive grammar could define the product space of an SPL whose sentences are legal sequences of features. Such a grammar is a ***feature model*** [7]. A partial feature model for JBook is below (given that each of the sublanguages in its design is useful); the first line is a context free grammar. Notation "`[T]`" denotes feature `T` is optional. Subsequent lines define propositional formulas as compositional constraints to make the grammar context sensitive:

```
JBook : Expl [ΔStml] [ΔExpC] [ΔStmC] ... ;
// constraints
ΔExpC ⇒ ΔStml;       // if ΔExpC then so too must ΔStml
ΔStmC ⇒ ΔExpC;       // if ΔStmC then so too must ΔExpC
```

These are the basics of SPLs [4]; a more advanced discussion is in [5].

3 SPL Feature Models and Product Spaces

A feature model can be translated to a propositional formula ϕ [2,3,4]. This is accomplished in two steps: (1) the context free grammar is translated to a propositional formula ϕ', and (2) composition constraints are conjoined with ϕ' to produce ϕ. For example, the lone production of the JBook context free grammar, defined above, is translated to:[6]

$$\phi' \;=\; \big(\; \texttt{JBook} \Leftrightarrow (\texttt{Expl})\;\big) \wedge \big(\; (\Delta\texttt{Stml} \;\vee\; \Delta\texttt{ExpC} \;\vee\; \Delta\texttt{StmC} \;\vee\; \ldots) \Rightarrow \texttt{JBook}\;\big)$$

where each term is a boolean variable. The complete propositional formula ϕ is:

$$\phi \;=\; \phi' \;\wedge\; (\Delta\texttt{ExpC} \Rightarrow \Delta\texttt{Stml}) \;\wedge\; (\Delta\texttt{StmC} \Rightarrow \Delta\texttt{ExpC})$$

Every solution of ϕ corresponds to a unique product (a unique set of features) in that SPL. ***Binary Decision Diagrams* (BDD)** and ***Sharp-SAT solvers* (#SAT)** can count the number of products of ϕ [12,21,32,40]. Industrial SPLs can have colossal product spaces. Consider the table below from [5,21]:

[5] Much like different data structures implement the same container abstraction [8].
[6] More involved examples and explanations are given in [2,3,4].

Model	#Variables	#SAT-Solutions	Source
axTLS 1.5.3	64	10^{12}	http://axtls.sourceforge.net/
uClibc 201 50420	298	10^{50}	https://www.uclibc.org/
Toybox 0.7.5	316	10^{81}	http://landley.net/toybox/
BusyBox 1.23.2	613	10^{146}	https://busybox.net/
EmbToolkit 1.7.0	2331	10^{334}	https://www.embtoolkit.org
LargeAutomotive	17365	10^{1441}	[26]

272 is a magic number in SPLs. If an SPL has 272 optional features, it has 2^{272} unique products. $2^{272} \approx 10^{82}$ is a ***really big number***: 10^{82} is the current estimate of the number of atoms in the universe [46]. The **LargeAutomotive** SPL in the above table has a colossal space of 10^{1441} products. That makes the largest numbers theoretically possible in Modern Cosmology look really, really small ☺ [35].[7] And there are even *larger* known SPLs (e.g., the Linux Kernel), whose size exceeds the ability of state-of-the-art tools to compute.

Beyond admiring the size of these spaces, suppose you want to know which product in a space (or a user-defined subspace) has the best performance for a given a workload. Obviously, enumerating and benchmarking each product is infeasible. The immediate question is: How does one search colossal product spaces efficiently? A brief survey of current approaches is next.

4 Searching SPL Product Spaces

To predict the performance of SPL products, a mathematical performance model is created. Historically, such models are developed manually using domain-specific knowledge [1,13]. More recently, performance prediction models are learned from performance measurements of sampled products. In either case, a performance model is given to an optimizer, which can then find near-optimal products that observe user-imposed feature constraints (e.g., product predicates that exclude feature `F` and include feature `G`).

4.1 Prediction Models

Models can estimate the performance of any valid product [17,37,43,42,54]. The goal is to use as few samples as possible to learn a model that is 'accurate'. Finding a good set of training samples to use is one challenge; another is minimizing the variance in predictions.

Let $\mathbb{C}$ be the set of all legal SPL products. 1^{st}-order performance models have the following form: let $\$P$ be the estimated performance of an SPL product $P \in \mathbb{C}$, where f_P is the set of `P`'s selected features and $\$F_i$ is the performance

[7] Still 10^{1441} does pale in comparison to 10^{284265}, the size of the space of texts a monkey can randomly type out, one text of which *is* Hamlet [36,34] or 10^{40000}, the size of the space of texts a monkey can type out, one text of which is this paper.

contribution of feature F_i:

$$\$P = \sum_{i \in f_P} \$F_i \tag{1}$$

$\$F_i$ might be as simple as a constant (c_i) or a constant-weighted expression [17]:

$$\$F_i = c_0 \tag{2}$$

or

$$= c_0 + c_1 \cdot n + c_2 \cdot n \cdot \log(n) + c_3 \cdot n^2 + ... \tag{3}$$

where n is a global variable that indicates a metric of product or application 'size'. The value for n is given; the values of constants (c_i) must be 'learned'.

1^{st}-order performance models are linear regression equations without (feature) interaction terms. Such models are inaccurate. Let $\$F_{ij}$ denote the performance contribution of the interaction of features F_i and F_j, which requires both F_i and F_j to be present in a product; $\$F_{ij}=0$ otherwise. 2^{nd}-order models take into account 2-way interactions:

$$\$P = \left(\sum_{i \in f_P} \$F_i\right) + \left(\sum_{i \in f_P} \sum_{j \in f_P} \$F_{ij}\right) \tag{4}$$

Models with n-way interactions add even more nested-summations to `(4)` [42].

Manually-developed performance models [1,9,13] are different as they:

- Identify operations [O_1..] invoked by system clients;
- Define a function $\$O_k$ to estimate the performance of each operation O_k;
- Encode system workloads by operation execution frequencies, where ν_k is the frequency of O_k; and
- Express performance $\$P$ of a program P as a weighted sum of frequency times operation cost:

$$\$P = \sum_k \nu_k \cdot \$O_k \tag{5}$$

Features complicate the cost function of each operation, where the set of features of product $P \in \mathbb{C}$ becomes an explicit parameter of each O_k:

$$\$P = \sum_k \nu_k \cdot \$O_k(f_P) \tag{6}$$

In summary, manual performance models include workload variances in their predictions, whereas current SPL performance models use a *fixed* workload. Workload variations play a significant role in SPL product performance. To include workloads in learned models requires relearning models from scratch or transfer learning which has its own set of issues [23].[8]

[8] Transfer learning is an automatic translation of one performance model to another.

4.2 Finding a Near-Optimal is NP-Hard

The simplest formulation of this problem, namely as linear regression equations, is NP-Hard [52]. Here's a reformulation of Eqn. (1) as a 0-1 Integer Programming Problem. Let $\mathbb{1}_i(P)$ be a boolean indicator variable to designate if feature F_i is present ($\mathbb{1}_i(P)$=1) or absent ($\mathbb{1}_i(P)$=0) in P. Rewrite Eqn. (1) as:

$$\$P \;=\; \sum_{i \in f_P} \$F_i \;=\; \sum_{i} \$F_i \cdot \mathbb{1}_i(P)$$

We want to find a configuration c_{near} to minimize \$P, Eqn. (7). To do so, convert Eqn. (7) into a inequality with a cost bound b, Eqn. (8). By solving Eqn. (8) a polynomial number of times (progressively reducing b) we can determine a near optimal performance $\$c_{near}$ and c_{near}'s features (the values of its indicator variables):

$$\min_{P \in \mathbb{C}} (\$P) \;=\; \min_{P \in \mathbb{C}} \Big(\sum_{i} \$F_i \cdot \mathbb{1}_i(P) \Big) \tag{7}$$

$$\min_{P \in \mathbb{C}} \;\Big(\sum_{i \in f} \$F_i \cdot \mathbb{1}_i(P) \Big) \;\leq\; b \tag{8}$$

Recall a feature model defines constraints among features, like those in the "**Prop Formula**" column of Fig. 2. There are well-known procedures to translate a propositional formula to a linear constraint, and then to $\leq$ inequalities [16,22].

Prop Formula	Linear Constraint	Linear Inequality
$x \wedge y$	$x + y = 2$	$-x - y \leq -2$
$x \vee y$	$x + y \geq 1$	$-x - y \leq -1$
$x \Rightarrow y$	$x - y \geq 0$	$x - y \leq 0$
$x \Rightarrow \neg y$	$x + y \leq 1$	$x + y \leq 1$
$x \Leftrightarrow y$	$x = y$	$(x - y \leq 0) \wedge (y - x \leq 0)$
$Choose1(x, y)$	$x + y = 1$	$(x + y \leq 1) \wedge (-x - y \leq -1)$

Fig. 2. Prop Formula to an Integer Inequality.

To optimize Eqn. (8) correctly, feature model constraints must be observed. The general structure of the optimization problem described above is:

$$\begin{array}{rll} \text{find } \mathbf{x} \text{ such that} & \mathbf{c}^T\mathbf{x} \leq b & \text{rewrite of Eqn. (8)} \\ \text{subject to} & \mathbf{A}\mathbf{x} \leq \mathbf{d} & \text{feature model constraints} \end{array}$$

where $\mathbf{x} \in \mathbb{1}^n$ ($\mathbf{x}$ is an array of n booleans), $c \in \mathbb{Z}^n$ and $b \in \mathbb{Z}$ (c is an array of n integers, b is an integer), $A \in \mathbb{Z}^{m \times n}$ (A is an $m \times n$ array of integers), and $\mathbf{d} \in \mathbb{Z}^m$ ($\mathbf{d}$ is an array of n integers). This is the definition of 0-1 Linear Programming, which is NP-Complete [52]. The NP-hard version removes bound b and minimizes $\mathbf{c}^T\mathbf{x}$.

4.3 Uniform Random Sampling

Optimizers and prediction models [17,18,19,37,38,39,54] rely on 'random sampling', but the samples used are not provably uniform. ***Uniform Random Sampling***

(URS) conceptually enumerates all η=|$\mathbb{C}$| legal products in an array $\mathbb{A}$. An integer $\mathtt{i} \in [\,\mathtt{1}..\eta\,]$ is randomly selected (giving all elements in the space an equal chance) and $\mathbb{A}[\mathtt{i}]$ is returned. This simple approach is not used because η could be astronomically large. Interestingly, URS of large SPLs was considered infeasible as late as 2019 [24,33].

An alternative is to randomly select *features*. If the set of features is valid, a product was "randomly" selected. However, this approach creates far too many invalid feature combinations to be practical [17,18,37,39,54]. Another approach uses SAT solvers to generate valid products [19,38], but this produces products with similar features due to the way solvers enumerate solutions. Although Henard et al. [19] mitigated these issues by randomly permuting the parameter settings in SAT solvers, true URS was not demonstrated.

The top path of Fig. 3 summarizes prior work: the product space is non-uniformly sampled to derive a performance model; samples are interleaved with performance model learning until a model is sufficiently 'accurate'. That model is then used by an optimizer, along with user-imposed feature constraints, to find a near-optimal performing product.

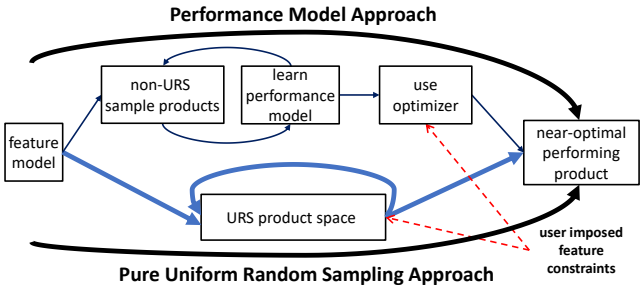

Fig. 3. Different Ways to Find Near-Optimal Products.

In contrast, a pure URS approach (the bottom path of Fig. 3) uses neither performance models nor optimizers. Near-optimal products are found by uniformly probing the product space directly, and benchmarking the performance of sampled products using the required workload. User-imposed feature constraints simply reduce the space to probe. A benefit of URS is that it is a standard way to estimate properties accurately and efficiently of colossal spaces [14]. It replaces heuristics with no guarantees with mathematics with confidence guarantees.

Note ❮ For some, it may not evident that URS could be used for optimization. In fact, ***Random Search*** (**RS**) algorithms [10,50] do exactly this – find near-optimal solutions in a configuration space. We present evidence later that URS requires many fewer samples than existing performance model approaches [30]. ❯

5 URS Without Enumeration

Let $\eta = |\phi|$ be the size of an SPL product space whose propositional formula is ϕ. Let $\mathcal{F} = [\,\mathtt{F}_1, \mathtt{F}_2, ..\mathtt{F}_\theta\,]$ be a list of optional SPL features. Randomly select an integer $\mathtt{i}\in[\mathtt{1}..\eta]$ and compute $\mathtt{s}_1 = |\phi \wedge \mathtt{F}_1|$, the number of products with feature $\mathtt{F}_1$. If $\mathtt{i}\leq\mathtt{s}_1$, then $\mathtt{F}_1$ belongs to the $\mathtt{i}^{\mathtt{th}}$ product and recurse on the subspace $\phi\wedge\mathtt{F}_1$ using feature $\mathtt{F}_2$. Otherwise $\neg\mathtt{F}_1$ belongs to the $\mathtt{i}^{\mathtt{th}}$ product and recurse on subspace $\phi \wedge \neg\mathtt{F}_1$ with $\mathtt{i}=\mathtt{i}-|\phi \wedge \mathtt{F}_1|$ using feature $\mathtt{F}_2$. Recursion continues until feature $\mathtt{F}_\theta$ is processed, at that point every feature in the $\mathtt{i}^{\mathtt{th}}$ product is known.

> **Note** ❮ Historically, Knuth first proposed this algorithm in 2012 [25]; Oh and Batory reinvented and implemented it in 2017 using classical BDDs [30]. Since then other SAT technologies were tried [11,32,40]. (A #SAT solver is a variant of a SAT solver: instead of finding a solution of ϕ efficiently, #SAT counts ϕ solutions efficiently.) The most scalable version today is by Heradio et al. and uses reduced BDDs [21], which in itself is surprising as for about a decade, SAT technologies have dominated feature model analysis. ❯

Given the ability to URS a SPL colossal product space, how can a near-optimal product for a given workload be found? That's next.

6 Performance Configuration Space (PCS) Graphs

Let $\mathbb{C}$ denote the product space of ϕ, where $\eta{=}|\phi|{=}|\mathbb{C}|$. Imagine that for every product $\mathtt{P}\in\mathbb{C}$ we predict or measure a ***performance metric*** $\$(\mathtt{P})$ for a given benchmark. By "performance metric", we mean any non-functional property of interest of $\mathtt{P}$ (response time, memory size, energy consumption, throughput, etc.). A small $ value is good (efficient) and a large $ value is bad (inefficient). An optimal product $\mathtt{P}_{\mathtt{best}}$ in $\mathbb{C}$ has the smallest $ metric:[9]

$$\exists \mathtt{P}_{\mathtt{best}}\in\mathbb{C} \;:\; \Big(\;\forall \mathtt{P}\in\mathbb{C} \;:\; \$(\mathtt{P}_{\mathtt{best}}) \leq \$(\mathtt{P})\;\Big) \tag{9}$$

For large $\mathbb{C}$, creating all $(\mathtt{P}, \$(\mathtt{P}))$ pairs is impossible... ***but imagine that we could do so.*** Further, let's normalize the range of $ values: Let $\$(\mathtt{P}_{\mathtt{best}})=0$ be the best performance metric and let $\$(\mathtt{P}_{\mathtt{worst}})=1$ be the worst. Now sort the $(\mathtt{P}, \$(\mathtt{P}))$ pairs in increasing $\$(\mathtt{P})$ order where $\$(\mathtt{P}_{\mathtt{best}})=0$ is first and $\$(\mathtt{P}_{\mathtt{worst}})=1$ is last, and plot them. The result is a ***Performance Configuration/Product Space (PCS)*** graph, Fig. 4a. This graph suggests that PCS graphs are continuous; they are not. PCS graphs are stair-stepped, discontinuous and non-differentiable [27] because consecutive products on the X-axis encode discrete decisions (features) that can make discontinuous jumps in performance, Fig. 4b.

[9] To maximize a metric, negate it.

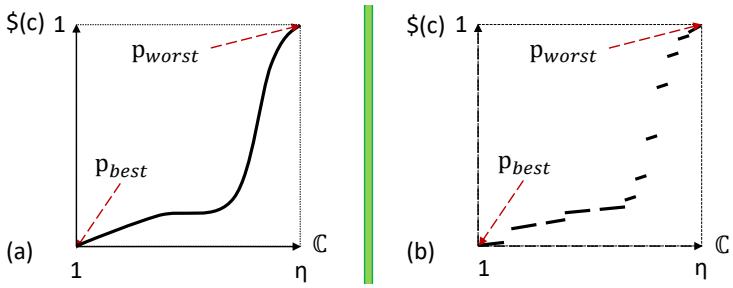

Fig. 4. Normalized PCS Graphs.

Example ❮ Suppose product P_i has feature F and F is replaced by G in P_{i+1}. If F increments performance by .01, say, and G increments performance by .20, there will be a discontinuity from $\$(P_i)$ and $\$(P_{i+1})$ in a PCS graph. ❯

Note ❮ Every PCS graph is monotonically non-decreasing. The latter means that consecutive products on the X-axis, like P_i and P_{i+1}, must satisfy $\$(P_i) \leq \$(P_{i+1})$. Many products in $\mathbb{C}$ may have indistinguishable performance values/metrics because their differing features have *no* impact on performance, leading to $\$(P_i) = \$(P_{i+1})$. ❯

Random Search ($\mathbb{RS}$) is a family of numerical optimization algorithms that can be used on functions that are discontinuous and non-differentiable [10,50]. The simplest of all $\mathbb{RS}$ algorithms is the ***Best-of-n-Samples*** below. Here we use URS for sampling:

```
1. Initialize x with a random product in the search space.
2. Until a termination criterion is met (n − 1 samples) repeat:
   2.1 Sample a new product y in the search space.
   2.2 If $(y)<$(x) set x=y.
3. Return x.
```

Listing 1.1. Best-of-n-Samples

How accurate is the returned product? An answer can be derived by exploiting a PCS graph's monotonicity, next.

7 Analysis of Best-of-n-Samples

The X-axis of a PCS graph (i.e., the product space) can be approximated by the real unit interval $\mathbb{I}=[0..1]$ when $\eta > 2000$ [30]. $\mathbb{I}$ emerges from the limit:

$$\lim_{\eta\to\infty} \frac{1}{\eta} \cdot \left[1..\eta\right] \;=\; \lim_{\eta\to\infty} \left[\frac{1}{\eta}..\frac{\eta}{\eta}\right] = [0..1] \;=\; \mathbb{I} \tag{10}$$

Randomly select a product in $\mathbb{C}$, i.e., one point in $\mathbb{I}$. URS means each point in $\mathbb{I}$ is equally likely to be chosen. It follows that *on average* the selected product $p_{1,1}$ partitions $\mathbb{I}$ in half:

$$p_{1,1} = \int_0^1 x \cdot dx = \frac{1}{2} \tag{11}$$

Now randomly select n products from $\mathbb{C}$. *On average* n points partition $\mathbb{I}$ into $n+1$ equal-length regions. The k^{th}-best product out of n, denoted $p_{k,n}$, has rank $\frac{k}{n+1}$, where the $k \cdot \binom{n}{k}$ term below is the normalization constant [5]:[10]

$$p_{k,n} = k \cdot \binom{n}{k} \cdot \int_0^1 x^k \cdot (1-x)^{n-k} \cdot dx = \frac{k}{n+1} \tag{12}$$

The left-most selected product, which is a near-optimal product $P_{nearOpt}$=$p_{1,n}$, is an average distance $\frac{1}{n+1}$ from the optimal $P_{best}=0$ by Eqn. (12):

$$p_{1,n} = \frac{1}{n+1} \tag{13}$$

Let's pause to understand this result. Look at Fig. 5. As the sample set size n increases (Fig. (a)→Fig. (c)), $P_{nearOpt}$ progressively moves closer to P_{best} at $X=0$. ***If n=99 samples are taken,*** $P_{nearOpt}$ ***on average will be 1% from*** P_{best} ***in the ranking along the X-axis.***

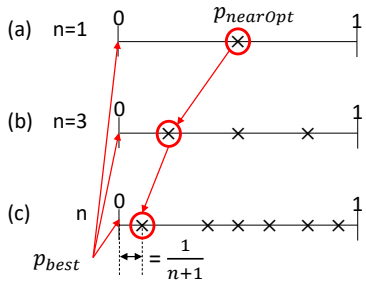

Fig. 5. Convergence to P_{best} by increasing Sample Size (n).

Note: *None* of the equations (11)-(13) reference η or $|\mathbb{C}|$; both disappeared when we took the limit in (10). This means (11)-(13) predict sampled ranks (that is, X-axis ranks) for an *infinite-sized space*. Taking $n=99$ samples on *any* colossal product space, on average $P_{nearOpt}$ will be 1% from P_{best} in the ranking. It is only for minuscule product spaces, $\eta \leq 2000$, where predictions by equations (11)-(13) will be low [30]. Such small product spaces are enumerable anyway, and not really of interest to us.

[10] Eqn. (12) is an example of the Beta function [47].

Third, how accurate is the $p_{1,n}=\frac{1}{n+1}$ estimate? Answer: the standard deviation of $p_{1,n}$, namely $\sigma_{1,n}$ [5,29], can be computed from $v_{1,n}$, the second moment of $p_{1,n}$:

$$v_{1,n} = 1 \cdot \binom{n}{1} \cdot \int_0^1 x^2 \cdot (1-x)^{n-1} \cdot dx = \frac{2}{(n+1)\cdot(n+2)}$$

$$\sigma_{1,n} = \sqrt{v_{1,n} - {p_{1,n}}^2} = \sqrt{\frac{2}{(n+1)\cdot(n+2)} - \left(\frac{1}{n+1}\right)^2} \quad (14)$$

Observe $p_{1,n}$ is almost equal to $\sigma_{1,n}$. Fig. 6 plots the percentage difference between the two:

$$\%\texttt{diff} = 100 \cdot \left(\frac{p_{1,n}}{\sigma_{1,n}} - 1\right) \quad (15)$$

For $n=80$ samples $p_{1,n}$ is 1.2% higher than $\sigma_{1,n}$ which is itself small. *For* $n \geq 100$ *there is no practical difference between* $p_{1,1}$ *and* $\sigma_{1,n}$. Stated differently, URS offers remarkable good accuracy and variance with $n \geq 100$ [30].

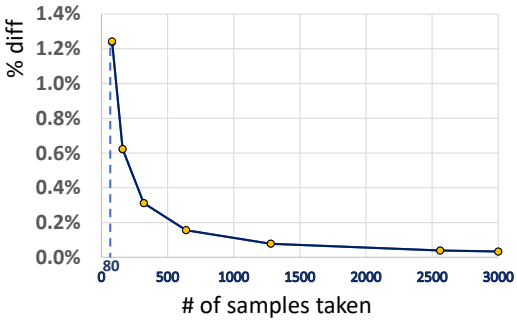

Fig. 6. %diff Plot.

Bottom line. To find a near-optimal product in colossal product space, take a uniform-random sample set of size n, predict or measure the performance of each product, and return the best performing product as it will be $\frac{100}{n+1}\%$ away, with variance approximately $\frac{100}{n+1}\%$, from the optimal product, P_{best}, in the space.

Percentiles. Readers may have noticed that our ranking is how 'close' $P_{nearOpt}$ is X-axis-based from P_{best}, where the more typical notion is Y-axis-based, i.e., the fraction $(P_{nearOpt})$ is from (P_{best}). The utility of PCS graphs is this: *optimizing* X *also optimizes* Y.

A common X-axis metric in statistics is a ***percentile***, see Fig. 7 [28]. Candidates are lined up and the percentage of candidates that are "shorter" than You (the blue person) is computed. You are in the (top) 80% percentile. In performance optimization, we want to be in the lowest percentile, ideally <1% means "in the top <1 percentile".

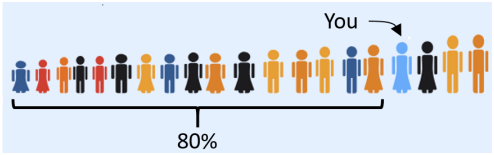

Fig. 7. Percentiles.

8 Another Benefit of URS in Product Optimization

Prior to performance models for and URS of colossal product spaces, URS was compared with early results on small SPLs that used performance models [30].

The performance model contestants were **Sankar2015** [37] and **Siegmund2012** [42], which at the time were the best models to date.

Fig. 8 shows two non-PCS graphs: the X-axis is the number `n` of samples taken to form a prediction model or a Best-of-`n`-Samples result and the Y-axis is the fraction distance of their returned $\mathtt{P}_{\mathtt{nearOpt}}$ to $\mathtt{P}_{\mathtt{best}}$.

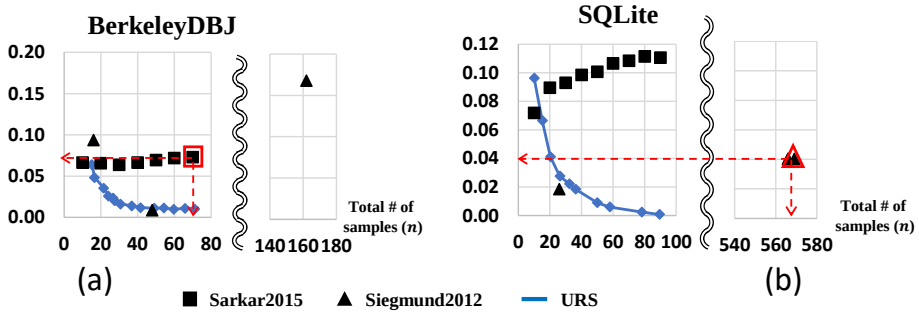

Fig. 8. Comparison of URS with Existing Performance Models in 2017.

In short:

- The accuracy of both performance models did *not* improve with increasing `N`, unlike URS which progressively improves;
- URS obtained the same accuracy with less work (fewer samples) than both **Sarkar2015** (see ▣ in Fig. 8a) and **Siegmund2012** (see ◬ in Fig. 8b); and
- URS obtained better results for the same work (see ▣ and ◬ above).

Other similar results are reported in [30]. However, these results are outdated and need to be refreshed with the latest performance model technologies and URS technologies; at best they are provisional and suggest future work.

9 Choosing a Sample Set Size

An open problem with non-URS methods is: what sample set size is needed to find $\mathtt{P}_{\mathtt{nearOpt}}$ with a given accuracy? As there is no formal analysis of non-uniform sampling, it is not known how to answer this question. However, URS has an answer. The following elegant derivation is by Heradio [20], better than [31].

Confidence assertions are of the form: with `90%` probability $\mathtt{P}_{\mathtt{nearOpt}}$ will be within the top ρ percentile. Let ρ be the desired percentile (e.g., top 1% has ρ=`.01`). In one random selection, we have probability ρ that a desired product was selected and $(1-\rho)$ that it was not. After `n` selections, we have probability $(1-\rho)^{\mathtt{n}}$ that *no* selections were desirable and $1-(1-\rho)^{\mathtt{n}}$ that at least one of them is. Let `c` denote the confidence (probability) that after `n` selections $\mathtt{P}_{\mathtt{nearOpt}}$ is in the top ρ percentile:

$$\mathtt{c} \;=\; 1-(1-\rho)^{\mathtt{n}} \tag{16}$$

Solving for `n`:

$$\mathtt{n} \;=\; \frac{\ln(1-\mathtt{c})}{\ln(1-\rho)} \tag{17}$$

The table of Fig. 9 lists the sample set size to use for a given confidence (`c`) and accuracy (ρ) ***no matter how colossal the space***. **Note:** 90% and 95% are common degrees of confidence; 99.7% is known as ***near certainty*** since it encompasses 3σ, virtually all values [51]:

n=sample set size	%confidence			
%accuracy	90.0%	95.0%	98.0%	99.7%
5.00%	45	58	76	113
4.00%	56	73	96	142
3.00%	76	98	128	191
2.00%	114	148	194	288
1.00%	229	298	389	578
0.50%	459	598	780	1159
0.30%	766	997	1302	1933
0.20%	1150	1496	1954	2902
0.10%	2301	2994	3910	5806

Fig. 9. Sample Set Size to Achieve `%accuracy` with `%confidence`.

Example: A product:

- in the top 5% is returned in 45 samples with 90% confidence;
- in the top 2% is returned in 148 samples with 95% confidence; and
- in the top .20% is returned in 1954 samples with 98% confidence.

Eqn. `(16)` has three variables; given values of two, one can solve for the third. The previous discussion showed how to determine `n` given confidence `c` and accuracy ρ. Here are the two other possibilities:

Given `c` and `n`, what is the expected accuracy ρ? Solving `(16)` for ρ:

$$\rho \quad = \quad 1 - (1 - \mathtt{c})^{\frac{1}{\mathtt{n}}} \tag{18}$$

The table of Fig. 10 has rows for confidence `c` values, columns are the number of samples taken `n`, and entries are the accuracy ρ of returned answers:

%ρ	n = sample set size						
%confidence	25	50	100	200	400	800	1600
90.0%	8.80%	4.50%	2.28%	1.14%	0.57%	0.29%	0.14%
95.0%	11.29%	5.82%	2.95%	1.49%	0.75%	0.37%	0.19%
98.0%	14.49%	7.53%	3.84%	1.94%	0.97%	0.49%	0.24%
99.7%	20.73%	10.97%	5.64%	2.86%	1.44%	0.72%	0.36%

Fig. 10. Expected Accuracy ρ given `c` and `n`.

Example ❮ Suppose a total of `n=100` samples are to be taken and 95% confidence is desired in an answer. The returned solution has accuracy in the top 2.95% of all solutions. ❯

Given n and ρ, what is expected confidence c? Eqn. (16) is already solved for c and is repeated below:

$$c = 1 - (1 - \rho)^n$$

The table of Fig. 11 has rows for accuracy values ρ, columns are the total number of samples taken n, and entries are the confidence c of returned answers:

%c = confidence	n = number of samples						
%ρ = accuracy	25	50	100	200	400	800	1600
4.000%	63.96%	87.01%	98.31%	99.97%	100.00%	100.00%	100.00%
2.000%	39.65%	63.58%	86.74%	98.24%	99.97%	100.00%	100.00%
1.000%	22.22%	39.50%	63.40%	86.60%	98.20%	99.97%	100.00%
0.500%	11.78%	22.17%	39.42%	63.30%	86.53%	98.19%	99.97%
0.250%	6.07%	11.76%	22.14%	39.38%	63.26%	86.50%	98.18%
0.125%	3.08%	6.06%	11.76%	22.13%	39.37%	63.24%	86.48%

Fig. 11. Expected Confidence c given ρ and n.

> **Example** ❮ Taking n=100 samples and wanting accuracy ρ=1%, the confidence of a returned answer is 63.4%. That is, there is a 63.4% chance that the returned answer is within the top 1% of all products. ❯

There are other analyzes for the Best-of-n-Samples algorithm, like how many samples are needed to have two samples returned in the top ρ% with c% confidence, is much like the above. Also, a recursive search of a space is another interesting possibility – performing a search and using the data collected to reduce the size of the space to search in the next recursion. A noticeable improvement in $P_{nearOpt}$ was observed by recursive searching [30] – with two important caveats. We have not yet determined how to guarantee that P_{best} is not pruned in a space restriction, nor do we have mathematics to compute the confidence of returned results. These remain open problems.

10 Given a Limit of n Samples...

Benchmarking is *by far* the greatest cost in sampling. Suppose a client is willing to pay the cost of benchmarking 100 samples to find a near optimal product for a particular precision ρ and confidence c.

Question: Would it be better to conduct one experiment E of 100 samples for a given ρ accuracy, or two experiments E_1 and E_2 of 50 examples each again with the same ρ accuracy, and take the best result? In the latter case, would the confidence change by using two experiments?

Answer: There is no difference! It is not difficult to see that the $P_{nearOpt}$ answer in either case would be the same: $P_{nearOpt}$ would be the result of experiment E_1 or E_2, and would be the best-of-both result.

It is a bit harder to see is how the confidence of the two-experiment result is the same as a one-experiment result. Let $\mathtt{c}$ be the confidence of both $\mathtt{E}_1$ and $\mathtt{E}_2$. The confidence $\mathtt{c2}$ we would have in the result of taking the best-of-both experiments is weighted. Namely, sum the product of confidences where at least one experiment succeeds:

$$\begin{aligned} \mathtt{c2} = \;& \mathtt{c}\cdot\mathtt{c}\cdot\mathtt{1} \;+\; \mathtt{c}\cdot(\mathtt{1}-\mathtt{c})\cdot\mathtt{1} + (\mathtt{1}-\mathtt{c})\cdot\mathtt{c}\cdot\mathtt{1} + (\mathtt{1}-\mathtt{c})\cdot(\mathtt{1}-\mathtt{c})\cdot\mathtt{0} \\ =\;& \mathtt{2}\cdot\mathtt{c} - \mathtt{c}^2 \end{aligned} \tag{19}$$

Let $\mathtt{c}(\mathtt{n},\rho) = \mathtt{1} - (\mathtt{1}-\rho)^{\mathtt{n}}$, Eqn. (16), be the confidence of an experiment for a fixed $\mathtt{n}$ and ρ. It is easy to prove the following equality that shows the confidence of a single experiment with $\mathtt{2}\cdot\mathtt{n}$ samples and ρ accuracy and the confidence of two smaller experiments with $\mathtt{n}$ samples and ρ accuracy, Eqn. (19), are the same:

$$\mathtt{2}\cdot\mathtt{c}(\mathtt{n},\rho) \;-\; \mathtt{c}(\mathtt{n},\rho)^2 \;=\; \mathtt{c}(\mathtt{2}\cdot\mathtt{n},\rho) \tag{20}$$

10.1 PCS Graphs of Real SPLs

What do real PCS graphs look like? This is not a fundamental question, but one asked out of curiosity. Several small SPLs were analyzed by Siegmund et al. [41,42] that took several months of benchmarking:

- **H264** is a video encoder library for H.264/MPEG-4 AVC format written in C. With 16 features and 1152 configurations, Sintel trailer encoding times were measured, see Fig. 12a and
- **BerkeleyDBC** is an embedded database system written in C. With 18 features and 2560 configurations, benchmark response times were measured. Note its multiple "*stairs*" or vertical leaps. See Fig. 12b.

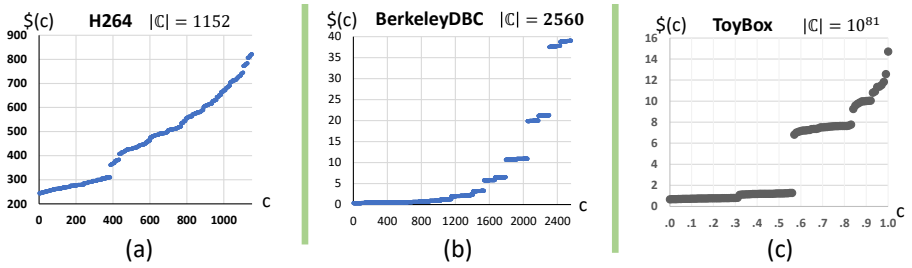

Fig. 12. H264, BerkeleyDBC, and ToyBox PCS Graphs.

Fig. 12a-b are ***Complete PCS graphs*** – meaning all products are plotted. This is possible when configuration spaces are tiny. But what about SPLs with colossal spaces? What then? A number of techniques were tried, and the simplest performed best:

- Randomly select $\mathtt{n}$=100 or $\mathtt{n}$=200 configurations, as 100-200 points are sufficient resolution for a graph in a paper,

- Predict the performance or build-and-benchmark each sample,
- Sort the samples from best-performing to worst,
- Let p_i be the i^{th} best performance. Plot a PCS graph using the n points $\{(x_i, y_i)\}_{i=1}^{n} = \{(\frac{i}{n+1}, p_i)\}_{i=1}^{n}$.

Example. ToyBox 0.7.5 provides Android systems with a collection of Linux command-line utilities within a single executable. It has 316 features and 10^{81} configurations [45]. Build size was measured. Its PCS graph, Fig. 12c, was produced with n=100, although the graph for n=200 was identical.

11 Future Work and Next Steps

There is a hunger in Software Engineering research for more scientific approaches to be used, where mathematics can help solve fundamental design problems. The use of mathematics is evident in the work of Börger et al. on ASMs and the JBook [44]; so too in the area of SPLs. Software design indeed has a mathematical foundation, but perhaps not how Dijkstra, Hoare, and Wirth initially envisioned. Science must deliver quite a lot before it can overcome Cowboy Programming [48]. The Science of Software Design will answer questions that were unanswerable previously.

This holds for finding near-optimal products in colossal SPL product spaces, a practical problem whose roots are found in early work on SWD. Given the ability to URS such spaces, an entire world of prior results on $\mathbb{RS}$ is now applicable. The simplest $\mathbb{RS}$ algorithm, Best-of-n-Samples, can answer scientific questions that prior approaches could not. Namely, given any two of (confidence of answer, accuracy of answer, and number of samples to take), the third can be computed. Perhaps other $\mathbb{RS}$ algorithms may be analyzable as well.

Software Engineering research is fad-driven – the latest is ***Machine Learning*** **(ML)**. ML also can provide answers to questions that couldn't be answered before. We showed in Sec. 7 that near-optimal results can be accompanied with accuracy or confidence metrics – to give precision about returned results that could not be determined before. Or that the number of samples to take is no longer a guess – it can be precisely computed. And in Sec. 8, URS can also provide more accurate answers than performance models with less work (fewer samples), although these results need to be refreshed as they used small product spaces (what was available at that time). Today's open question is whether the provisional results in this paper scale to colossal spaces.

In this paper, URS may have been offered unintentionally as a tool to solve all analysis problems. Far from the truth, URS is but one in an ever-increasing sophisticated arsenal of techniques that can be used. Coupled with domain-specific knowledge, URS tools will be even better. URS will likely become a lower-bound on what can be accomplished and accepted (w.r.t. accuracy, confidence, and work) in future work. If so, we have indeed made progress.

To Egon. You and your work continue to inspire me and others. Thank you.

Acknowledgments. We thank the referees for their helpful comments on this paper.

References

1. S. Agrawal, S. Chaudhuri, and V. R. Narasayya. Automated Selection of Materialized Views and Indexes in SQL Databases. In *VLDB*, 2000.
2. S. Apel, D. Batory, C. Kästner, and G. Saake. *Feature-Oriented Software Product Lines*. Springer, 2013.
3. D. Batory. Feature Models, Grammars, And Propositional Formulas. In *SPLC*, 2005.
4. D. Batory. *Automated Software Design Volume 1*. Lulu.com, 2020.
5. D. Batory. *Automated Software Design Volume 2*. in development, 2022.
6. D. Batory and E. Börger. Modularizing Theorems for Software Product Lines: The JBook Case Study. *JUCS*, 14(12), June 2008.
7. D. Batory and S. O'Malley. The Design and Implementation of Hierarchical Software Systems with Reusable Components. *ACM TOSEM*, 1992.
8. D. Batory, V. Singhal, J. Thomas, and M. Sirkin. Scalable Software Libraries. In *ACM SIGSOFT*, 1993.
9. D. S. Batory and C. C. Gotlieb. A Unifying Model of Physical Databases. *ACM TODS*, 1982.
10. J. Bergstra and Y. Bengio. Random Search for Hyper-Parameter Optimization. *Journal of Machine Learning Research*, 2012.
11. S. Chakraborty, D. J. Fremont, K. S. Meel, S. A. Seshia, and M. Y. Vardi. On Parallel Scalable Uniform SAT Witness Generation. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*. Springer Berlin Heidelberg, 2015.
12. S. Chakraborty, K. Meel, and M. Vardi. A Scalable and Nearly Uniform Generator of SAT Witnesses. In *CAV*, 2013.
13. S. Chaudhuri. An Overview of Query Optimization in Relational Systems. In *PODS*, 1998.
14. J. Devore. *Probability and Statistics for Engineering and the Sciences*. Cengage Learning, 2021.
15. E. W. Dijkstra. The Structure of 'THE'-multiprogramming System. *CACM*, 11, May 1968.
16. J. Emmanuel. Integer Linear Programming - Binary (0-1) Variables. `https://www.youtube.com/watch?v=-3my1TkyFiM`, 2021.
17. J. Guo, K. Czarnecki, S. Apel, N. Siegmund, and A. Wasowski. Variability-aware Performance Prediction: A Statistical Learning Approach. In *ASE*, 2013.
18. J. Guo, J. White, G. Wang, J. Li, and Y. Wang. A Genetic Algorithm for Optimized Feature Selection with Resource Constraints in Software Product Lines. *Journal of Systems and Software*, 2011.
19. C. Henard, M. Papadakis, M. Harman, and Y. Le Traon. Combining Multi-objective Search and Constraint Solving for Configuring Large Software Product Lines. In *ICSE*, 2015.
20. R. Heradio. Derivation of Sample Set Size. Private Correspondence with Batory, 2020.
21. R. Heradio, D. Fernandez-Amoros, J. Galindo, D. Benavides, and D. Batory. Uniform and Scalable Sampling of Highly Configurable Systems. In *Submitted*, 2021.

22. L. Hodes. Solving problems by formula manipulation in logic and linear inequalities. *Artificial Intelligence*, 3:165–174, 1972.
23. P. Jamshidi et al. Transfer Learning for Performance Modeling of Configurable Systems: An Exploratory Analysis. In *ASE*, 2017.
24. C. Kaltenecker, A. Grebhahn, N. Siegmund, J. Guo, and S. Apel. Distance-Based Sampling of Software Configuration Spaces. In *ICSE*, 2019.
25. D. E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks and Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 2009.
26. S. Krieter, T. Thüm, S. Schulze, R. Schröter, and G. Saake. Propagating Configuration Decisions with Modal Implication Graphs. In *ICSE*, 2018.
27. B. Marker, D. Batory, and R. van de Geijn. Understanding Performance Stairs: Elucidating Heuristics. In *ASE*, 2014.
28. MathIsFun. Percentiles. `https://www.mathsisfun.com/data/percentiles.html`.
29. MathIsFun. Standard Deviation Formulas. `https://www.mathsisfun.com/data/standard-deviation-formulas.html`, 2019.
30. J. Oh, D. Batory, M. Myers, and N. Siegmund. Finding Near-optimal Configurations in Product Lines by Random Sampling. In *FSE*, 2017.
31. J. Oh, P. Gazzillo, D. Batory, M. Heule, and M. Myers. Percentile Calculations for Randomly Searching Colossal Product Spaces. Technical Report TR-18-05, Dept. of Computer Science, University of Texas at Austin, 2018.
32. J. Oh, P. Gazzillo, D. Batory, M. Heule, and M. Myers. Scalable Uniform Sampling for Real-World Software Product Lines. Technical Report TR-20-01, Dept. of Computer Science, University of Texas at Austin, 2020.
33. Q. Plazar, N. Acher, G. Perrouins, X. Devroey, and M. Cordy. Uniform Sampling of SAT Solutions for Configurable Systems: Are We There Yet? In *Software Testing, Verification, And Validation*, 2019.
34. S. Saibian. Sbiis Saibian's Large Number Site. `https://sites.google.com/site/largenumbers/home`.
35. S. Saibian. The Largest Numbers Theoretically Possible in Modern Cosmology. `https://sites.google.com/site/largenumbers/home/2-1/Largest_Numbers_in_Science`.
36. S. Saibian. Larger Numbers in Probability, Statistics, And Combinatorics. `https://sites.google.com/site/largenumbers/home/2-1/Large_Numbers_in_Probability`, 2021.
37. A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki. Cost-efficient Sampling for Performance Prediction of Configurable Systems. In *ASE*, 2015.
38. A. S. Sayyad, J. Ingram, T. Menzies, and H. Ammar. Scalable Product Line Configuration: A Straw to Break the Camel's Back. In *ASE*, 2013.
39. A. S. Sayyad, T. Menzies, and H. Ammar. On the Value of User Preferences in Search-based Software Engineering: A Case Study in Software Product Lines. In *ICSE*, 2013.
40. S. Sharma, R. Gupta, S. Roy, and K. Meel. Knowledge Compilation Meets Uniform Sampling. In *Logic for Programming, Artificial Intelligence, And Reasoning (LPAR)*, 2018.
41. N. Siegmund et al. Dataset for Siegmund2012. `http://fosd.de/SPLConqueror`, 2012.
42. N. Siegmund et al. Predicting Performance Via Automated Feature-interaction Detection. In *ICSE*, 2012.
43. N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner. Performance-influence Models for Highly Configurable Systems. In *FSE*, 2015.

44. R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification, Validation*. Springer-Verlag, 2001.
45. Toybox Website. `http://landley.net/toybox/`, 2018.
46. J. Villanueva. How Many Atoms Are There in the Universe? `https://www.universetoday.com/36302/atoms-in-the-universe/`.
47. Wikipedia. Beta Distribution. `https://en.wikipedia.org/wiki/Beta_distribution`.
48. Wikipedia. Cowboy Coders. `https://en.wikipedia.org/wiki/Cowboy_coding`.
49. Wikipedia. Lego. `https://en.wikipedia.org/wiki/Lego`.
50. Random Search. `https://en.wikipedia.org/wiki/Random_search`.
51. Wikipedia. 68-95-99.7 Rule. `https://en.wikipedia.org/wiki/68%E2%80%9395%E2%80%9399.7_rule`, 2019.
52. Wikipedia. Integer Programming. `https://en.wikipedia.org/wiki/Integer_programming`, 2021.
53. N. Wirth. Program Development by Stepwise Refinement. *CACM*, 1971.
54. Y. Zhang, J. Guo, E. Blais, and K. Czarnecki. Performance Prediction of Configurable Software Systems by Fourier Learning. In *ASE*, 2015.