Using Heuristics to Automate Parameter Generation for Benchmarking of Java Methods

Michael Kuperberg[1] Fouad Omri[2]

Chair for Software Design and Quality
Institute for Program Structures and Data Organisation
Faculty of Informatics, Universität Karlsruhe (TH)

Abstract

Automated generation of method parameters is needed in benchmarking scenarios where manual or random generation of parameters are not suitable, do not scale or are too costly. However, for a method to execute correctly, the generated input parameters must not violate implicit semantical constraints, such as ranges of numeric parameters or the maximum length of a collection. For most methods, such constraints have no formal documentation, and human-readable documentation of them is usually incomplete and ambiguous. Random search of appropriate parameter values is possible but extremely ineffective and does not pay respect to such implicit constraints. Also, the role of polymorphism and of the method invocation targets is often not taken into account. Most existing approaches that claim automation focus on a single method and ignore the structure of the surrounding APIs where those exist. In this paper, we present HEURIGENJ, a novel heuristics-based approach for automatically finding legal and appropriate method input parameters and invocation targets, by approximating the implicit constraints imposed on them. Our approach is designed to support systematic benchmarking of API methods written in the Java language. We evaluate the presented approach by applying it to two frequently-used packages of the Java platform API, and demonstrating its coverage and effectiveness.

Keywords: Heuristics, parameter generation, exception handling, automated benchmarking, constraint approximation

1 Introduction

Most software applications developed today build on object-oriented languages and execution platforms. For example, the Java Virtual Machine executes Java bytecode, to which the Java programming language and other programming languages are compiled. For Java, the building blocks of such applications are classes, which contain methods and fields. The functional properties (e.g. correctness) and extra-functional properties (e.g. performance) of methods are subject of ongoing research, and analysis of these properties must consider the impact of method input parameters. Also, the state of the objects and class instances whose methods are invoked must be considered.

For example, in benchmarking, the input parameters often have a strong impact on the method performance, so different parameters must be studied. This task quickly becomes too expensive for manual implementation if the number of methods is very high, as it is

[1] Email: mkuper@ipd.uka.de
[2] Email: omri@ipd.uka.de

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

often the case in the application programming interfaces (APIs): the Java platform API contains several thousands of methods. However, there exists no automated API benchmarking tool or strategy for Java APIs. In this paper, Java API denotes *any* API compiled to and accessible from Java bytecode; we explicitly refer to the *Java **platform** API* when the functionality provided by the Java Runtime Environment is meant.

Where manual generation of method parameters is not suitable, randomised approaches are often tried, but they become ineffective where the potential parameter space is too large. Also, existing randomised approaches mostly focus on testing-oriented cases, i.e. on finding cases where software's behaviour *deviates* from the expected, specified targets. In contrast to maximising failure occurrence, benchmarking needs to find parameters that do *not* deviate from expected execution w.r.t. exceptions and errors. Also, software testing does not need to recover or to learn from failed parameters, while in benchmarking, failures must be minimised as much as possible to achieve good coverage.

If a method requires input parameters, they must be provided in accordance with their static types in the method's signature, e.g. for interface-typed parameters, an instance of a class implementing that interface must be passed. In addition, implicit semantical requirements for these parameters exist: for example, the method `java.lang.String.substring(int beginIndex)` throws an `IndexOutOfBoundsException` for an instance `str` if `beginIndex < 0` or if `beginIndex` $\geq$ `str.length()`.

In the cases where such requirements are given, if at all, they are described informally by humans and for humans, and thus cannot be evaluated by tools due to the complexity and general ambiguity of human language. Also, there are no formal specifications that can be used by an automated approach. Guessing an appropriate value using a random search is intractable given the large range of possible values that could be generated for each single parameter; for the above example, the parameter `beginIndex` has a range of 2^{32} different values. The few existing approaches that claim automation of parameter generation focus on a single method and ignore the structure of the surrounding APIs where those exist.

The contribution of this paper is a novel self-correcting approach for the automatic generation of input parameters for Java methods, based on formally-defined heuristics. The heuristics help to find parameters which can be used in meaningful benchmarks. The presented approach detects inappropriate methods arguments on the basis of thrown exceptions, automatically approximates underlying exception causes using novel heuristics and recovers them by generating new and appropriate input parameters. The generation of the parameter values for a method is not based on a random search but on a *feedback-directed heuristic search*, and its results are reused for other considered methods.

We discuss the current prototype implementation of our approach, which is called HEURIGENJ and evaluate it for the methods declared in classes of two frequently-used Java platform API packages, `java.util` and `java.lang`. For both packages, we evaluate the proportion of methods for which our approach could generate appropriate parameter values, and demonstrate the effectiveness of our approach in handling runtime exceptions.

The remainder of this paper is organised as follows. Section 2 describes the foundations, and Section 3 gives an overview of the presented approach. After presenting the heuristics used for generating arguments in Section 4, advanced heuristic algorithms that are useful to handle runtime exceptions are specified in Section 5. The case study is described in Section 6. Section 7 reviews related work, while Section 8 describes our assumptions and limitations. The paper is concluded in Section 9.

2 Foundations

This paper concentrates on Java bytecode, an intermediate high-level executable format of programs which are compiled for execution on a standard-compliant Java Virtual Machine (JVM). From the Java bytecode point of view, a constructor is a (special) method, so this paper includes constructors into the term "methods".

As mentioned in the previous section, if the input parameters are not within a required range, exceptions will occur at runtime. Another reason for runtime exceptions are wrongly-typed parameters. The signature of a method with a list of its input parameters and their declared types can be retrieved using the Java Reflection API or the Java bytecode engineering tools like Javassist [1]. Due to the polymophism support in Java language, bytecode and the JVM, the runtime (*dynamic*) type of a value/reference parameter often must specialise its declared (*static*) type. Also, the *static* parameter types are often interfaces or abstract classes, which cannot be instantiated (in fact, the runtime type of a parameter is never an interface or an abstract class).

In this paper, we consider three categories of parameter types: (i) primitive *value* types (e.g., `int`, `char`, etc.), (ii) collection types and arrays and (iii) non-collection reference types (i.e., class and interface types). For the primitive types in Java bytecode, static type and dynamic type are always equal. In contrast to that, cases (ii) and (iii) can have a dynamic type that is a subtype of the static type, while (iii) is always of type `Object` or a subtype of `Object`, following the type hierarchy.

To select among appropriate dynamic types, our approach makes use of a *parameter graph* which specifies the non-abstract subtypes of a given parameter that can be used, and their (non-abstract) constructors that exist in the considered API. The construction of a parameter graph is described in the next section, and Fig. 1 illustrates a parameter graph for a fictive method `meth`, where the declared type of the first input parameter is the interface `CharSequence`. In addition to constructors, the parameter graph includes *factory methods*, i.e. static non-abstract methods which return instances of a given type. Factory methods that return abstract or interface types are perfectly acceptable, as they return instances of proper non-abstract subtypes at runtime.

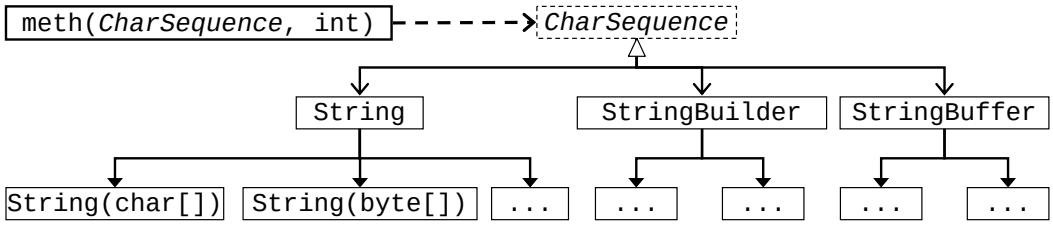

Fig. 1. Subgraph of the parameter graph for the method meth(CharSequence, int)

The functionality for retrieving all *sub*types of a given type is not available in the Java Reflection API or other parts of the Java platform API - it is only possible to get the *super*class of a given class using the method `java.lang.Class.getSuperClass()`. Such a functionality relies only on the compiled bytecode of the class and the contained *extends* relationship, and we are using it to build a *API model* that contains *bidirectional* inheritance relations of the given API. Then, parameter graphs can be constructed from such an API model, even without having the API's source code.

Our approach is designed to work for any Java API, not only the Java platform API. However, the classes of the platform API, such as `java.lang.String`, are very heavily used in *all* Java APIs. Hence, instances of types from the platform API are needed as parameters of methods in other APIs and we have started with the platform API.

3 Overview of the Approach

In this section, we present our self-correcting automatic approach for parameter generation, called HEURIGENJ. Fig. 2 presents an overview of HEURIGENJ's functioning.

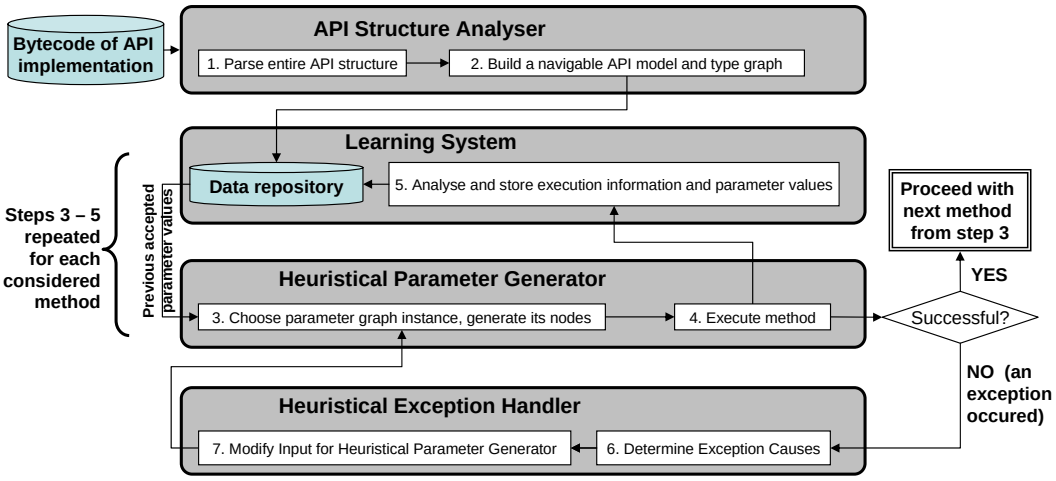

Fig. 2. Simplified overview of the Automatic Self-Correcting Parameter Generation Approach

To establish the inheritance and implementation relations discussed in Sec. 2, our approach starts in **step 1** by parsing the compiled bytecode of the API implementation. Then, in **step 2**, a navigable API model is built, which includes a type graph. To facilitate usage and maintenance, the API model is designed to extend the official Java Reflection API. Step 2 also computes a complexity metric for each method signature, based on the number and on weighted complexities of input parameters (see Sec. 5 for details).

The results of step 2 are stored persistently in a repository that is part of a *learning system*, which also contains knowledge about successful and failed parameter generations and method executions. The data in this repository enables reuse of values and discovered relations, and can serve as the basis for deeper analysis, e.g. with genetic algorithms [2]. The data can also be used in other contexts, such as testing or documentation.

HEURIGENJ processes the methods in increasing order of signature complexity and with respect to dependencies on returned values (e.g. if a method returns a type instance needed as input parameter by other methods, as identified by the parameter graph). For each method, steps 3 through 5 are performed. If a method execution fails with given input parameters, it is decided whether that method should be abandoned or if other input parameter instances should be tried by performing steps 6 and 7. This decision is controlled by the possibility to choose other nodes in the parameter graph, by the importance of the method, and by the number of repetitions spent for the given method signature. The default strategy for this decision proved to be sufficiently successful for the Java platform API (cf. Sec. 6), but it can be replaced by the HEURIGENJ user.

3.1 Preparing Method Parameters

In **step 3**, HEURIGENJ selects a sufficient set of nodes (i.e. implementing types, constructors/methods) from the parameter graph for object-typed parameters of the considered method(cf. Fig. 1). The heuristics that drive this choice aim at selecting the simplest nodes, which are less likely to fail. Then, HEURIGENJ consults the repository and selects from it, if available, known instances of the selected nodes, as well as known values of primitive parameter types. If no values/instances are found in the repository, HEURIGENJ generates parameters using heuristics, as described in more detail in Sec. 4.3.

In **step 4**, the resulting parameters are used to execute the considered method, using Java Reflection API. For non-static methods, the construction of the invocation target is also accomplished in step 4. If the method returns a value (i.e., not `void`), that value needs to be recorded in the repository, as it can be later used as input parameter to other methods.

In **step 5**, information about method execution and used parameters is evaluated and stored, including thrown exceptions, their causes and the relevant part of the stack trace.

For benchmarking, steps 3 through 5 should be repeated for the same method to obtain several different parameter values, e.g. for finding parametric performance dependencies on an input parameter. For this task, a wrapper can be written for HEURIGENJ so that the latter can be invoked repeatedly to (i) attempt to heuristically find n different quasi-random values of one given parameter, while fixing the other input parameters (if any), or (ii) only use a certain range of values for a given input parameter, or (iii) take x samples of the complete parameter space of a given method, whereas all input parameters may be varied, and the parameter space may be bounded tighter than default Java value ranges.

3.2 Dealing with Exceptions

If method execution in step 4 fails, an `Exception` or a subclass of it is thrown and can be caught (we do not consider `Errors`, as they are abnormal conditions that occur very rarely and should not be caught; wrong method parameters are not known to cause `Errors`).

While the (initial) parameter generation uses heuristics to decrease the likelihood of inappropriate parameters, additional "post-mortem" heuristics are needed to deal with exceptions if they occur. Both sorts of heuristics are a novel contribution of our approach.

In **step 6**, the causes of exceptions are approximated by comparing the input parameters and the type of the exception. For example, an `ArrayIndexOutOfBoundsException` is used in HEURIGENJ to reconsider the `int` method parameters to better fit to the array's length, as described in Sec. 5. In general, step 6 considers not only method input parameters, but also (for non-static methods) the state of the invocation target: for example, a `List` may be deemed too small for the desired operation if several input parameter choices failed. Then, the `List` itself has to be adapted. Runtime exceptions that are declared in the Java platform may provide a textual description of the exception causes. However, such descriptions are not formal and there exists no approach known to the authors that offers reasoning on them. Therefore, we provided the mechanism with a formal description of the causes of some selected runtime exceptions declared in the Java platform (cf. Sec. 5).

In **step 7**, the identified exception causes and other available information are used to create input for heuristics that generate new instances of nodes in the parameter graph. In other words, the entire process starts again with another, presumably better configuration. As stated above, there is no guarantee on the success, and our approach imposes limits to

such repetitions - otherwise, it may degenerate into a randomised approach. In the next sections, we describe some of our heuristics in more detail, and evaluate our approach.

4 Heuristic Parameter Generation

In this section, we present the heuristic parameter generator (HPG) which is used in step 3 of our approach (cf. Fig. 2) to generate appropriate parameter values for method and constructors. We denote the signature of an invokable $\mathcal{I}$ (i.e., a method or a constructor) as $\mathcal{SG}$. The declaring class of an invokable $\mathcal{I}$ is referred as $\mathcal{DC}$ and the instance of $\mathcal{DC}$ as $\mathcal{DCI}$. We define *container types*, denoted as $\mathcal{CT}$, as the set of static types whose instance has a length or a capacity, for example arrays, collections, maps, but also strings and buffers.

4.1 Generation of Primitives

The choice of heuristics for the generation of primitives is motivated by two observations:

- often, the constants declared in $\mathcal{DC}$ and its superclasses are the input parameters which are more likely (or even exclusively) accepted by the considered method: for example, the method `java.util.Calendar.set(int year,int month,int date)` should make use of static `int` fields `JANUARY` etc. in that class
- if one of the method parameters is container-typed (e.g. an array or a `List`), the `int`-typed parameters in the method signature are likely to refer to that container, e.g. as 'from' or 'to' indexes: an example is the method `java.lang.String.getChars (int srcBegin, int srcEnd, char[] dst, int dstBegin)`

Accordingly, we describe here the two most important heuristic strategies that HPG defines for generating instances of primitive types as input parameters for an invokable $\mathcal{I}$.

The **first heuristic** of HPG is to use the constants (i.e. static final variables, if available) defined in $\mathcal{DC}$. The constants in the superclasses of $\mathcal{DC}$ are also considered (the set of superclasses is denoted $\mathcal{S}.\mathcal{DC}$). These constants may well be negative; the order of selection is randomised. If no declared constants are available (or if there are less declared constants than primitive parameters in the signature), the primitive values are generated randomly and may be negative as well. A random number generator with uniform distribution is currently used, but we plan to study distributions that favor smaller positive and larger negative values (i.e. values around zero), because it appears that these values are more frequent in practice.

The HPG needs to accounts for the fact that `int` parameter values are often used as indexes and thus are the only primitives likely to throw `IndexOutOfBoundsException`s.

Therefore, a **second heuristic** has been defined for `int`-typed parameter values: a lower and an upper bound are imposed on `int`-typed parameter values *if* container-typed parameters are present in the signature, or if $\mathcal{DC}$ is itself container-typed. For example, for generating the parameters for the method `String.getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`, the `dst` array of `char`s should be generated first, and then the `int` values `srcBegin`, `srcEnd` and `dstBegin` should be generated afterwards, as they have an obvious, important relation to `dst`. Hence, the second heuristic is applied after generating all other parameters in $\mathcal{SG}$.

A simple constraint that is used by the second heuristic is to set the lower bound of `int` values to 0. It should be stressed that this restrictive constraint is only applied if either $\mathcal{DC}$

is of container type, or if at least one of parameters in the signature of $\mathcal{I}$ is container-typed - in other cases, `int` parameters may well be negative.

After the lower bound has been calculated, the heuristic calculation of the upper bound $\mathcal{BOUND}$ for the `int` values is carried out, as specified in the Algorithm 1. In the case of the above method `String.getChars(int srcBegin, int srcEnd, char[] dst, int dstBegin)`, the upper bound that HPG will find is `dst.length` which means that the following three conditions should be true: (i) $0 \leq srcBegin \leq dst.length$, (ii) $0 \leq srcEnd \leq dst.length$ and (iii) $0 \leq dstBegin \leq dst.length$. In the Algorithm 1, if the signature of the target method has container-typed parameters, parameter generation of `int`-typed values does not consider the length or the size of the target class instance on which the method will be invoked, because it assumes that container-typed parameters used in Algorithm 1 have been already generated with consideration to the class instance, as we will demonstrate in the next section while generating container types.

```
/* S_INT is the set of int constants declared by S.DC */
Data: Method I
Result: BOUND: upper bound for generating int parameter values in SG(I)
1  CTS ← {{param|param ∈ SG} ∩ {param|param.TYPE ∈ CT}};
2  if CTS ≠ ∅ then
       /* SG declares container types */
3      BOUND ← min((param.VALUE).LENGTH|∀param ∈ CTS);
4  else
5      if (I is not static) ∩(DCI.TYPE ∈ CT) then
           /* DCI is of container type */
6          BOUND ← DCI.LENGTH;
7      else
8          if S_INT ≠ ∅ then
9              BOUND ← x ∈ S_INT;
10         else
11             BOUND ← random positive int value;
12         end
13     end
14 end
15 return BOUND;
```

Algorithm 1: Finding the Upper Bound for Integer Arguments

4.2 Generation of Container Types

During the generation of container-typed parameters, HPG must decide on the length of the container and the type of its elements. We refer to the static type of the container's elements as *component type* in convention with the Java programming language. For computing the length of the container parameter to generate, HPG selects the *first available* value from the following list as an upper inclusive bound the container size: (i) if the type of the $\mathcal{DC}$ is a container type: the length of $\mathcal{DCI}$ on which $\mathcal{I}$ is invoked, (ii) a positive non-zero `int` constant value declared in $\mathcal{DC}$ or (iii) a random positive non-zero `int` value.

'Non-zero' condition is imposed because containers of size zero (i.e. empty containers) will not allow to call methods like `elementAt`. In practice, we have set an upper bound for case (iii) to 10^5 to limit the size of containers to realistic sizes. Of course, if the benchmarking framework that uses HEURIGENJ needs larger containers, this restriction may be overriden by that framework by specifying larger containers, or by adding elements to the container that HEURIGENJ has generated.

According to the declared component type of the container, HPG *randomly* generates $\mathcal{L}$ elements of the declared component type, except where the component type is `Object` - in such cases, HPG generates `Object` values having the same dynamic type as $\mathcal{DC}$. Details about the generation of reference component types (i.e. `Object` and its subclasses) are described in the next section.

4.3 Generation of Objects

The parameters for which `Object`-typed parameters need to be generated can have different static types: *interface* static type (e.g. `java.util.List`), *abstract class* static type (e.g. `java.util.AbstractList`), or non-abstract class static type (e.g. `java.util.ArrayList`). As discussed in Sec. 2, the Java API does not allow to query which (non-abstract) subclasses of an interface exist. HEURIGENJ collects such information and creates a parameter graph, as described in Sec. 2. However, when several candidates exist, HEURIGENJ still needs to decide which subclass to choose, and which constructor to take.

Interface static types are instantiated by first retrieving the public non-abstract classes implementing the interface, and then instantiating one of them as explained below. For *abstract-class* static types, the subclasses of the type's declaring class are retrieved and one of them is instantiated. If this doesn't work, factory methods returning the interface type/abstract type are tried, and the dynamic type they return is identified and stored.

To generate a parameter whose static type is declared as a *non-abstract class*, HPG first chooses the simplest constructor/factory method based on complexity of its signature. For example, the constructor `String(byte[] bytes, String charsetName)` is complexer than the constructor `String(int[] codePoints,int offset,int count)`. The complexity of a constructor's signature is judged on both the number of parameters it declares and their static type. From the perspective of HPG, signatures that declares only primitive parameters are less complex than the ones that declare fewer but reference type parameters. If the simplest constructor turns out to be inappropriate (i.e. it throws runtime exceptions or returns null objects, or empty objects such as a string of length 0), other constructors or factory methods are tried.

Preferring the simplest constructor means that HEURIGENJ is more likely to be successful in constructing the parameter, because a more complex constructor intuitively offers more 'chances' to fail. At the same time, simpler constructors often sufficiently cover the parameter space: `String(byte[] bytes)` is as powerful as the more complex constructor `String(byte[] bytes, int offset, int length)`. A study to quantify the impact of preference of simpler constructors is planned for future work.

Some API methods declare parameters of `java.lang.Object` type, a generic non-abstract type. As we have observed that the use of objects that implement the interface `java.lang.Comparable` reduces the likelihood of exceptions (because sorting and administration of collections are easier), we prefer `java.lang.Comparable`-implemen-

ting subclasses of `java.lang.Object`, e.g. classes such as `String` and its subclasses.

HPG pays special attention to the generation of *reference* container types (e.g. collections, maps, strings, buffers). Container types are very similar to arrays, hence HPG computes the length of reference container types in the same way as for arrays (cf. Sec. 4.2). Another heuristical strategy is used for initialisation of such types: HEURIGENJ prefers constructors whose input parameters are arrays, for example `String(char[])`.

For collections such as classes implementing `Lists` and `Maps`, HPG constructs empty instances and then fills them with n objects (n smaller than the above fixed capacity/length) in respect to the type parameter bounds they declare. For example, in order to generate a `List<E extends Number>`, HPG constructs an empty `java.util.ArrayList` instance and fills it with objects having a dynamic type that is subtype of the type parameter bound `Number` (`Long` is such a subtype of `Number`).

5 Heuristic Exception Handler

The heuristically generated argument values still can cause runtime exceptions, as heuristics generally offer no guarantee of success. Consequently, in steps 6 and 7 of our approach (cf. Fig. 2), the caught exceptions are analysed and handled by the Heuristic Exception Handler (HEH), which devises new input for the heuristic parameter generator.

The handler and the generator interact closely, but are separate entities to allow for better extendability: the handler is modular and creates input for the generator; the generator can be modified without an effect on the handler as long as the interfaces between them are kept constant. The case study in Sec. 6 will discuss the feasibility of the entire HEURIGENJ approach; including the feasibility of the feedback mechanism in the HEH.

In the Java SE 6 platform API, the `java.lang.Exception` class has almost 80 *direct* subclasses, which in turn may have own subclasses. From our initial experience, the vast majority of exceptions that occur in case of inappropriate method parameters are the 38 subclasses of `java.lang.RuntimeException`. From these, HEURIGENJ currently covers 19 most frequent ones. In this section, we have chosen several of them for illustration purposes, and use the notations from the previous section.

5.1 *Handling `IndexOutOfBoundsExceptions`*

An `IndexOutOfBoundsException` is thrown when an index is out of range for a container class (e.g. `List`, `Queue`, etc.), for an array, or for a `String`. Our heuristics handle `IndexOutOfBoundsExceptions` as well as its subclasses `ArrayIndexOutOfBoundsExceptions` and `StringIndexOutOfBoundsExceptions`. Indexes are `int`-typed parameters, and as discussed in Sec. 4.1, they are generated *after* other parameters have been generated. In particular, we assume that all container-typed parameters have already been generated.

We first define the range $\mathcal{R}$ as the local minimum of positive (non-zero) lengths of the container-typed elements (incl. the length of $\mathcal{DC}$ itself in cases where the $\mathcal{DC}$ is container-typed and where the considered method $\mathcal{I}$ is non-static). Suppose that $\mathcal{I}$ declares n `int` arguments and that the discrete value of argument a_i is v_i ($1 \leq i \leq n$). Let $\mathcal{A} = \{a_1, a_2, ..., a_n\}$ denote the set of `int` arguments, and let $\mathcal{V} = \{v_1, v_2, ..., v_n\}$ denote the value set of $\mathcal{A}$ which should be generated.

Proposition 5.1 *We impose for the generation of $\mathcal{V}$ the following three conditions as described in equations 1, 2 and 3.*

$$\forall v_i \in \mathcal{V} : v_i \geq 0 \tag{1}$$

$$\sum_{v_i \in \mathcal{V}} v_i < \mathcal{R} \tag{2}$$

$$\forall i \in \{2, ..., |\mathcal{A}|\} : v_{i-1} \leq v_i \tag{3}$$

According to the equation 2, the (positive) `int` values that have to be generated should have a sum that is smaller than the range $\mathcal{R}$. This restriction and the sorting order imposed by equation 3 designed to correspond to many method signatures where the "from" index appears before the "to" index, and where the indexes (which start with 0) should not reach beyond the collection's first or last element. To define an individual value interval for each `int` parameter, the heuristic uses equation 4 and proceeds starting with $i = 1$ up to $i = n$, with $\mathcal{R}$ being the aforementioned range and $\mathcal{L}_i$ defined as follows:

$$\mathcal{L}_i = \begin{cases} 0 & \text{if } i = 0 \\ v_i & \text{if } 0 < i \leq n \end{cases}$$

$$\mathcal{L}_{i-1} \leq v_i \leq \frac{(\mathcal{R} - \sum_{k=1}^{|\mathcal{A}|} \mathcal{L}_{k-1})}{(|\mathcal{A}| - i + 1)}. \tag{4}$$

The algorithm tries the generated `int` values by invoking the considerd method $\mathcal{I}$ and recording any eventual exceptions. If the generated values still cause an `IndexOutOfBoundsException`, the algorithm permutates the generated `int` values. The algorithm terminates if no `IndexOutOfBoundsException` is thrown, or if all possible permutations have been tested. The possible number of permutations are defined as follow: for n `int` parameters in a method signature, the algorithm can perform maximal $n!$ parameter value permutations (in general, this is an acceptable value, with $4! = 24$ permutations for a method that has 4 `int`-typed parameters, 24 ranging orders of magnitude below the range of an `int` value in Java).

5.2 Handling `ClassCastExceptions`

`ClassCastExceptions` are thrown to indicate that the code has attempted to cast an object to a class type of which that object is not an instance. In order to handle `ClassCastExceptions`, we designed a heuristic that attempts to determine the appropriate dynamic type of the parameter. If several `Object`-typed parameters exist, the heuristic is applied to all of them.

If the $\mathcal{DC}$ that declares the considered method is non-generic, the heuristic generates the set $\mathcal{S}_{\mathcal{C} \cup \mathcal{IF}}$ of candidate static types for the parameter as follows: $\mathcal{S}_{\mathcal{C} \cup \mathcal{IF}}$ includes $\mathcal{DC}$ and all its subclasses/subinterfaces. Then, for each static type $\mathbf{T} \in \mathcal{S}_{\mathcal{C} \cup \mathcal{IF}}$, the heuristic generates new parameter value of type $\mathbf{T}$ and tests it by invoking the target method with the new parameter value. Interface-typed or abstract $\mathbf{T}$s are skipped in favor of their non-abstract subtypes (if any). The algorithm terminates when no `ClassCastExceptions`

are thrown, or when all possible types from $\mathcal{S}_{\mathcal{C}\cup\mathcal{IF}}$ have been used. If the generated parameter values still lead to exceptions, their handling is delegated to other exception handlers, which can access the execution history stored in the repository.

If $\mathcal{DC}$ is generic, more extensive measures are needed. For example, when executing the method `java.util.concurrent.DelayQueue.add(Object)`, a `ClassCast Exception` can be thrown. The exception indicates that the `Object` parameter cannot be casted to `java.util.concurrent.Delayed`, the latter being an interface. A heuristic thus has to deduce from the declaration of the class `DelayQueue` (`DelayQueue<E extends Delayed>`) that it accepts `Delayed`-implementing parameters only.

The `extends` keyword thus signals an *upper bound* w.r.t. type hierarchy, while the `super` keyword signals a *lower bound*. So in the case of $\mathcal{DC}$ being generic, our heuristic creates $\mathcal{S}_{\mathcal{C}\cup\mathcal{IF}}$ so that it contains (depending on the keyword in the $\mathcal{DC}$ signature) either all subclasses of the upper bound (incl. the bound itself), or all superclasses of the lower bound (including itself, but excluding `Object`). Then, elements of $\mathcal{S}_{\mathcal{C}\cup\mathcal{IF}}$ are processed as just described. Similar techniques are used for casting instances from `String`s.

5.3 Handling State Exceptions for Collections

Collections contain a set or a list of elements. Some collections allow duplicate elements and others do not; some are ordered and others unordered. Most collections have capacity-restricted implementations, which means that exceptions are thrown if the collection capacity is exceeded after an `add` operation, or if a `remove` operation cannot be performed because the collection is empty.

Example of exceptions that can be thrown by collection operation are the `java.util.NoSuchElementException` if there are no more elements in the the collection to enumerate, the `java.lang.IllegalStateException` if the collection class is not in an appropriate state for the requested operation or the `java.util.EmptyStackException` to indicate that the `Stack` is empty and for example no `pop` operations are allowed.

Proposition 5.2 *In order to handle a state exception thrown by a collection operation $\mathcal{OP}$, the relative operation of $\mathcal{OP}$ has to be called to change the state of the collection and prepare it for the target operation $\mathcal{OP}$. In order to handle a `java.util.NoSuchElementException` thrown for example by the `element` operation on a `Queue`, we should fill the queue by calling the relative operation `add` and then call the method `element` again.*

In order to handle such exceptions, we mapped each collection operation to its relative one (e.g. `add` vs. `remove`). Special attention was paid to filling the collections: capacity restrictions should not be violated. The number of elements to add in a collection should not exceed its declared capacity.

6 Case Study and Evaluation

We have conducted a case study to evaluate the following qualities of HEURIGENJ:
Coverage: The number of *public* non-abstract methods for which appropriate arguments are successfully generated without human intervention
Effectiveness: The number of runtime exceptions that were handled by HEURIGENJ and the duration of the parameter generation process

We concentrated on public non-abstract methods because they are the API methods which are used by programmers who use a third-party, "black-box" API. In future work, we will also study the parameter generation for 'protected' and 'package' methods, because these methods are relevant for the programmers that want to extend an open API or a framework API.

We validated the prototype implementation of our approach HEURIGENJ with frequently-used Java platform API packages `java.util` and `java.lang`. All described measurements were done on a computer with Intel Pentium Dual-Core 1.8 GHz CPU, 1 GB of main memory and Windows Vista OS running Sun JRE 1.6.0_03, in `-client` JVM mode.

Coverage for the `java.util` Package

The `java.util` package declares 69 non-abstract classes, of which 58 are public. For the case study, only the 58 public classes (i.e. the public part of the API) were considered, which declare 738 *public* non-abstract methods.

HEURIGENJ successfully generated parameters for 668 of the 738 public methods, resulting in a success rate of 90.51%. For an approach that does not need any formal definitions or specifications of the constraints it has to respect, this is a very respectable result.

For the following four classes, the rate of effectiveness of HEURIGENJ was relatively low, i.e. under 70 %: (1) `java.util.Properties`, (2) `java.util.Scanner`, (3) `java.util.StringTokenizer` and (4) `java.util.Timer`.

The class `java.util.Properties` declares six methods, all of which require special input streams of bytes such as an `InputStream`. The method `loadFromXML( InputStream in)` couldn't be invoked because it requires an input stream parameter value that constitutes a valid XML document. The generation of such a specific parameter value is almost impossible to automate and the preconditions of the method `loadFromXML` couldn't be predicted by the current version of HEURIGENJ.

In the class `java.util.Scanner`, all the methods that HEURIGENJ couldn't execute needed pattern values in the form of `String`s or `java.util.regex.Pattern`s. It was impossible for HEURIGENJ to predict such undeclared conditions and generate the right pattern values needed by the 29 methods that could not been executed successfully.

For the class `java.util.StringTokenizer`, HEURIGENJ couldn't execute three methods. The reason was that the instance that HEURIGENJ has automatically generated contains no tokens and hence the three methods (`nextToken()`, `nextToken(String delim)` and `nextElement()`) that iterate over the tokenizer's string have thrown a `java.util.NoSuchElementException` which couldn't be automatically handled.

Only one method of the class `java.util.Timer` was executed. The remaining seven methods require parameter values of type `java.util.TimerTask`. The Java API provides no classes that sub-class this abstract class. Consequently, HEURIGENJ was not able to generate the required `TimerTask` values for the seven methods.

Coverage for the `java.lang` Package

The `java.lang` package declares 76 public non-abstract classes. These 76 public classes declare 861 *public* non-abstract methods, of which HEURIGENJ could successfuly execute 790. Thus, the success rate of HEURIGENJ was 91.75%, which is a very

promising result. For the following three classes in the `java.lang` package, the coverage rate of HEURIGENJ was relatively low, i.e. under 70 %: (1) `java.lang.Object`, (2) `java.lang. Runtime` and (3) `java.lang.SecurityManager`.

For the class `java.lang.Object`, HEURIGENJ couldn't execute five methods: `notify()`, `notifyAll()`, `wait()`, `wait(long)` and `wait(long, int)`. All these methods throw an `IllegalMonitorStateException` because the thread executing these methods in HEURIGENJ is not the owner of the monitor of the `Object` instance on which the five methods are executed.

The class `java.lang.Runtime` declares the method `exec(String[] cmdarray, String[] envp, File dir)` and five related convenience methods. The two arguments `envp` and `dir` can be both `null`. All six methods check that `cmdarray` is a valid operating system command. Therefore, HEURIGENJ cannot guess the names of valid system commands and consequently a `SecurityException` is thrown.

None of the 34 methods declared in the class `java.lang.SecurityManager` could be executed since the creation of a `SecurityManager` instance is not trivial to automate. The only constructor declared by that class throws a `SecurityException` if a security manager already exists and its `checkPermission` method does not allow the creation of a new security manager.

Effectiveness of the Heuristics

In this section, we describe the effectiveness of our parameter generation approach (HEH refers to the Heuristical Exception Handler, cf. Sec. 5).

We used the following metrics:

- the number of runtime exceptions that were thrown *before* HEH was applied
- the number of runtime exceptions that were thrown *after* HEH was applied
- the duration of the entire process, including initial heuristical parameter generation, and including exception handling by HEH

These metrics were collected for the methods declared in the classes of the Java platform API packages `java.util` and `java.lang`, which were already discussed above. As we are not aware of a reference implementation or approach that uses completely-random parameter generation, we currently cannot analyse the effectiveness of the initial parameter generation (i.e. before the HEH is applied).

The measured time includes the time needed for (i) the generation of arguments (ii) the verification of the arguments by executing the method and listening for runtime exceptions and (iii) the handling of runtime exceptions if they occur. We exclude the time needed for storing the generated values in the HEURIGENJ database to concentrate on the core of our approach, i.e. on the heuristics. The methods for which the parameters were created have been executed using the Java Reflection API.

The parameter generation for the methods in the package `java.lang` took about 259.442 seconds (4.32 minutes). 151 out of 204 thrown runtime exceptions could be successfully handled, resulting in a success rate of 74.01 %.

The parameter generation for the methods in the package `java.util` took about 168.664 seconds (2.81 minutes). For that package, 160 runtime exceptions were thrown, of which HEURIGENJ could handle 95, resulting in a success rate of 59.37 %.

Thus, our approach scales very well, and can serve as a good basis for combination with other approaches, e.g. those described in the following related work section.

7 Related Work

Ferguson and Korel [3] proposed a technique to generate test data (i.e., input parameters) based on the execution of the program under test, beginning from a given input and systematically modifying the input so that it follows a different path. Opposed to our approach, this technique requires an input set of existing appropriate parameters, and aims at identifying possible branches to detects possible errors, and not at finding successful test cases suitable for benchmarking.

The random testing technique of Hamlet [4], an alternative to black-box and static regression testing, avoids complex analyses of program specifications by randomly selecting test cases from the input domain. A partial list for corresponding Java approaches includes RANDOOP[5], Jartege [6], Eclat [7] and JCrasher [8].

JCrasher [8] uses a "parameter-graph" to generate test inputs by finding method calls whose return type can serve as input parameters. JCrasher creates every input from the scratch using certain predefined values such as `1.0`, `0.0` and `-1.0` for `double` primitives and reports the sequences of methods calls that throw certain type of exceptions. Our approach goes a step further by analysing such exceptions and reacting to them.

RANDOOP [5] defines a pool of values from which an initial random input is computed, just like JCrasher. The random input is then used to generate new sequences by extending the old ones and *discarding* the ones that create redundant objects or throw exceptions. RANDOOP uses the sequences that do not throw exceptions or do not violate program contracts to generate regression tests. Such sequences are likely to be found if the random input is a correctly executing one.

Eclat [7] performs random generation of test inputs based on execution results. Like RANDOOP [5], Eclat uses the execution feedback to guide the generation process. However, RANDOOP extends Eclat by using a set of universally applicable object properties that can be extended *by the user* to generate test inputs and consequently does not require an existing test suite and a correct execution to start the generation of new tests. HEURIGENJ automates the generation of such input parameters. Consequently, HEURIGENJ can extend the pool of values defined in RANDOOP with meaningful values specific to the implementation under test avoiding a manual specification by the user.

Godefroid et al. [9,10] present a symbolic execution approach that builds on random input generation (also called "concolic execution"). Their approach executes the program by calculating path constraints on the used parameters. The constraints are then solved to create actual test inputs. However, Majumdar and Xu claim in [11] that current implementations of test generation based on concolic execution are problematic in practice, since a very large number of inputs must be generated in order to reach the part of code not related to input error handling. The randomly generated parameters are in most cases meaningless for the program execution and have to be iteratively refined using symbolic constraints.

Thus, Majumdar and Xu propose in [11] a solution that combines exhaustive enumeration of test inputs and symbolic execution driven test generation. However, their approach targets only programs whose valid inputs are specified by a context free grammar. Opposed to [11], HEURIGENJ can operate even when no such grammar exists.

8 Assumptions and Limitations

We have assumed an exception mechanism which allows to catch exceptions and to continue program execution. While many modern languages and execution platforms provide such mechanisms, some do not, e.g. C++ and older operating system that execute unmanaged binary code. In such contexts, our approach is not applicable.

In the work presented in this paper, we assume that work on parameter generation starts from the scratch. If execution data already exists, HEURIGENJ could make use of it to find additional parameter values, or find values faster. The current implementation of HEURIGENJ has no support for this.

Our approach is not well suited for APIs that encode semantically rich parameters in `Strings`, as done by the JDBC API or in XML processing. Likewise, our approach is not well-suitable for GUI APIs, APIs for file system access, or APIs that have an effect on the security or the integrity of a computer system.

The detailed effect of the decisions made by HEURIGENJ in choosing nodes in the parameter tree (cf. Sec. 2) remains to be studied, for which metrics for comparability and appropriateness need to be defined. Also, the utility, coverage and appropriateness of parameter values generated by HEURIGENJ remain to be studied.

Often, a method is executed successfully with a given set of input parameters, but an immediately following second invocation could fail, as for example with the method `java.util.List.remove(int index)` which throws an `IndexOutOfBoundsException` if `index >= size()`. Such a "parameter-consuming" behaviour is undesirable in scenarios where multiple repeated invocations of a method are needed, e.g. in benchmarking. However, investigation of such cases has been deemed future work on HEURIGENJ, though its current implementation already supports specifying an option to make repeated calls to the method in step 4, up to a configurable maximum number of calls. When this option is enabled, HEURIGENJ reports the highest successful number of calls before either the specified maximum was reached, or an exception has been thrown. Varying the input parameters to achieve successfull multiple execution of a method is theoretically possible, but leads to additional challenges.

9 Conclusions and Future Work

In this paper, we have presented HEURIGENJ, a novel approach for automated generation of input parameters for API methods in Java. The presented approach is suitable for the Java platform API (whose implementation is provided by the Java Runtime Environment), but also for third-party APIs accessible from Java.

In HEURIGENJ, method input parameters are obtained through combined usage of a novel heuristic parameter generator together with a self-correcting mechanism that handles runtime exceptions if they occur as a result of invalid parameter values. The presented mechanism allows to decrease the need for manual intervention during method parameter generation, and finds appropriate parameters faster than a brute-force search.

This paper provides a first evaluation of HEURIGENJ on the basis of two frequently-used packages of the Java platform API; `java.util` and `java.lang`. The results of the evaluations are promising and indicate a coverage (i.e. number of methods which could be executed without throwing runtime exceptions) of more than 90% for both packages.

The heuristics used for handling runtime exceptions have shown an effectiveness rate (i.e. number of runtime exceptions succefully handled by HEURIGENJ) of about 60%. In future work, we plan to define and to collect metrics on polymorphism coverage to see how far the space of an object-typed parameter is covered w.r.t. subclasses of the parameter type.

In the future, HEURIGENJ can be extended by incorporating machine learning and other techniques of search-based software engineering. In addition to coverage of methods in a package and effectiveness of execution, other metrics should be used to evaluate HEURIGENJ, e.g. the coverage of the parameter space w.r.t. given constraints.

We also plan to connect RANDOOP [5] to HEURIGENJ to enhance the generation of input parameters of a given method by providing RANDOOP with valid input values which it uses to generate further values for the same method. This will allow to broaden the coverage of the parameter space.

The principles of HEURIGENJ can be applied to APIs that are made available through other languagues than Java. For example, there exists no automated approach for input parameter generation for methods declared in the .NET runtime API. Many object-oriented principles and problems addressed by HEURIGENJ (e.g. polymorphism, abstract-typed parameters, complexity of method signatures, exception handling) are similar in .NET and other modern managed languages, especially those that compile to bytecode.

Acknowledgements: the authors would like to thank Klaus Krogmann, Anne Martens and other member of the SDQ research group for insightful discussions and suggestions.

References

[1] S. Chiba, "Javassist (Java Programming Assistant)," 2008, URL: http://www.csg.is.titech.ac.jp/ chiba/javassist/, last visit: June 9th, 2008. [Online]. Available: http://www.csg.is.titech.ac.jp/~chiba/javassist/

[2] J. R. Koza, *Genetic Programming – On the Programming of Computers by Means of Natural Selection*, 3rd ed. The MIT Press, Cambridge, Massachusetts, 1993.

[3] R. Ferguson and B. Korel, "The Chaining Approach for Software Test Data Generation," *ACM Trans. Softw. Eng. Methodol.*, vol. 5, no. 1, pp. 63–86, 1996.

[4] R. Hamlet, "Random testing," in *Encyclopedia of Software Engineering*. Wiley, 1994, pp. 970–978.

[5] C. Pacheco and M. D. Ernst, "Randoop: Feedback-Directed Random Testing for Java," in *OOPSLA '07: Companion to the 22nd ACM SIGPLAN conference on Object oriented programming systems and applications companion*. New York, NY, USA: ACM, 2007, pp. 815–816.

[6] C. Oriat, *Jartege: A Tool for Random Generation of Unit Tests for Java Classes*, ser. Lecture Notes in Computer Science. Heidelberg: Springer Berlin, September 2005, vol. 3712/2005, pp. 242–256. [Online]. Available: http://dx.doi.org/10.1007/11558569_18

[7] C. Pacheco and M. D. Ernst, "Eclat: Automatic Generation and Classification of Test Inputs," in *In 19th European Conference Object-Oriented Programming*, 2005, pp. 504–527.

[8] C. Csallner and Y. Smaragdakis, "JCrasher: an automatic robustness tester for Java," *Softw. Pract. Exper.*, vol. 34, no. 11, pp. 1025–1050, 2004.

[9] P. Godefroid, N. Klarlund, and K. Sen, "DART: Directed Automated Random Testing," in *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, vol. 40, no. 6. New York, NY, USA: ACM Press, June 2005, pp. 213–223. [Online]. Available: http://dx.doi.org/10.1145/1065010.1065036

[10] K. Sen, "Concolic Testing," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, 2007, pp. 571–572.

[11] R. Majumdar and R.-G. Xu, "Directed Test Generation using Symbolic Grammars," in *ESEC-FSE companion '07: The 6th Joint Meeting on European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering*. New York, NY, USA: ACM, 2007, pp. 553–556.