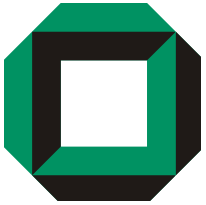
Universität Karlsruhe (TH)
Research University ▪ founded 1825

Heuristics-based Parameter Generation for Java Methods

Michael Kuperberg (mkuper@ipd.uka.de)
Fouad Omri (fouad.omri@stud.uni-karlsruhe.de)

Chair "Software Design and Quality", Faculty of Informatics,
Universität Karlsruhe (TH), Germany

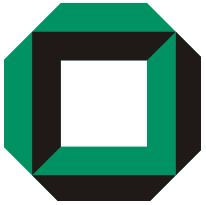


Motivation

- Example: generate parameters for Java API method **`String.substring(int index)`** for benchmarking
  - Parameter constraints: **`index ≥ 0`** and **`index < str.length()`**
  - Need a non-null invocation target (**`String`** instance, e.g. **`str`**)

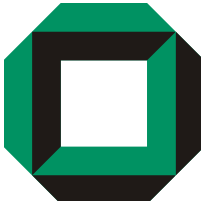
- Constraints clear to humans – unavailable to machines
- Violation of constraints → exceptions at runtime

- Java platform API: thousands of methods
  - Manual parameter generation: too costly
  - Random parameter generation: ignores above constraints



Related Work
Automated Parameter Generation

- Eclat [7]: random generation, based on execution results
  - needs an existing correct execution to start generation
- RANDOOP [8] extends Eclat, e.g. with a pool of values
  - from which an initial **random** input is computed
- Godefroid et al. [9]: a symbolic execution approach
  - uses **random** input generation (also called ”concolic execution”)
- Majumdar and Xu [10] say:
  [7-9] are problematic in practice
  - randomly generated parameters are in *most* cases meaningless for the program execution (out of scope; lead to exceptions)
  - hence, valid inputs must be specified by a context-free grammar



Challenges

SDQ Software Design and Quality

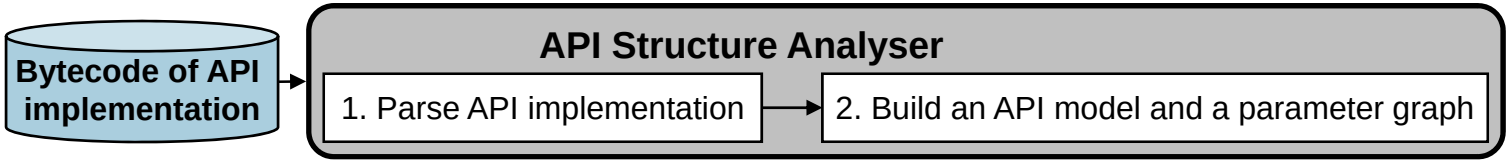
- Input parameter values have to be generated in accordance to the declared static types

- „Easy“: primitive static types (**`int`**, **`long`**, **`double`**, etc.)
- „Hard“: reference static types, e.g. interface types
  - E.g. **`cs`** in **`String.contentEquals(CharSequence cs)`**
  - No methods to get implementors of an interface in Java

- If method is part of an API: reuse API structure?
  - E.g. scan the API to find **`CharSequence`** implementers
  - Java has no corresponding concepts in the Reflection API

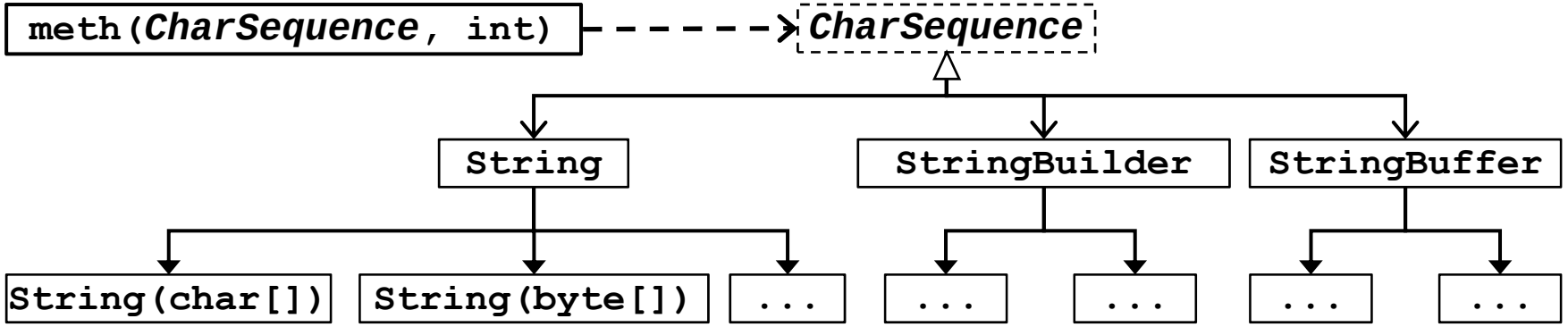
HeuriGenJ: Overview

- **Heuri**stics-based parameter **Gen**eration for **J**ava



Parameter Type Graph

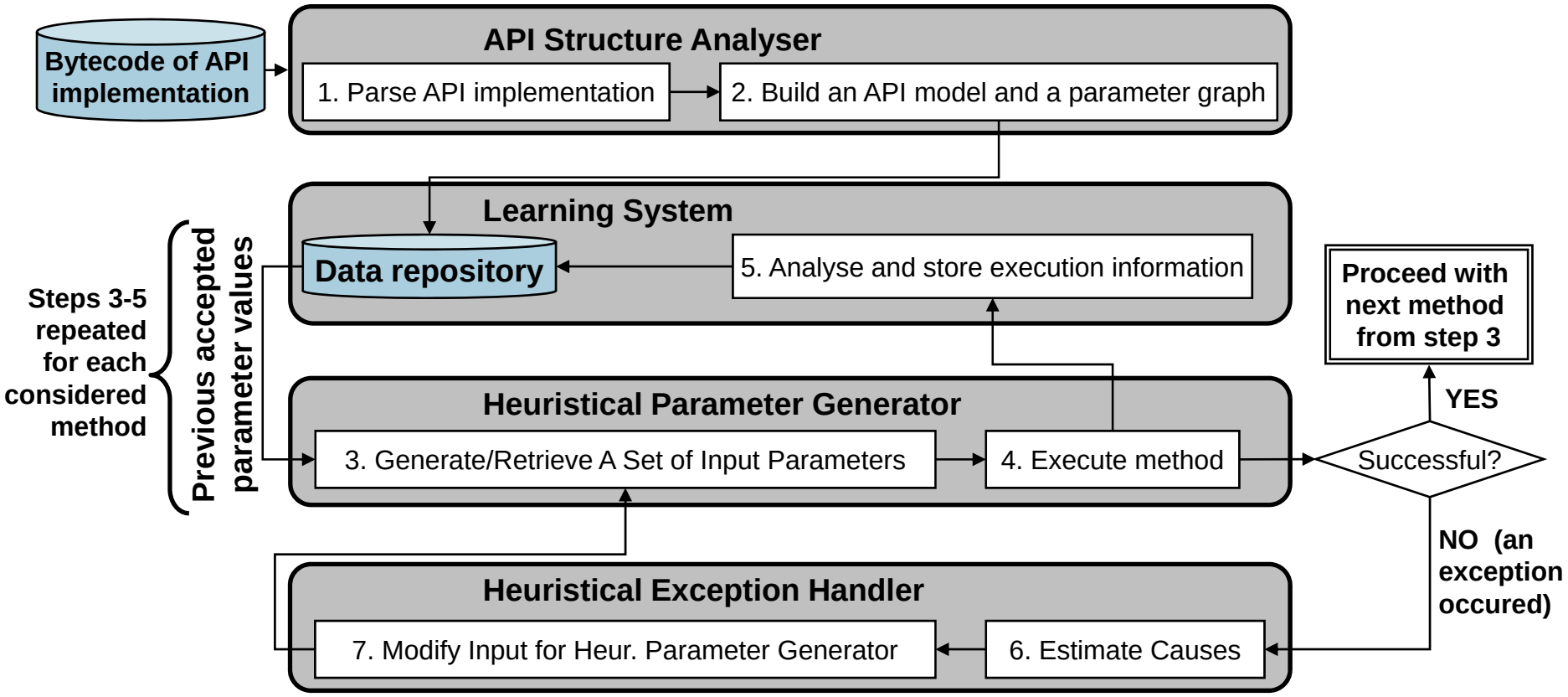
- Example:



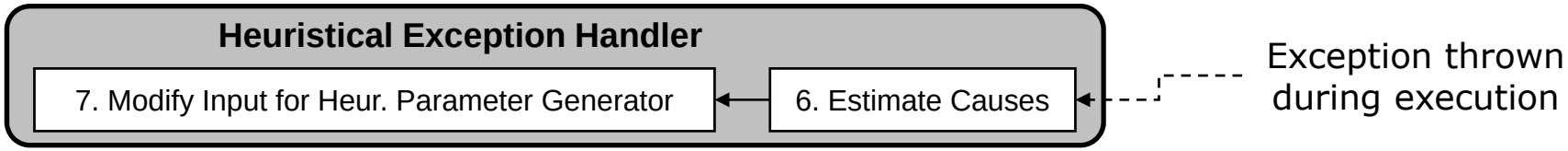
- Parameter type graph built by indexing implementations
  - we have tools to index bytecode and source code
  - incl. method meta-data (obtained using Java Reflection API)
- Knowledge is encoded as Prolog-like inference rules:
  - `String <- String(char[]), String(byte[]),…`

HeuriGenJ: Overview

- **Heuri**stics-based parameter **Gen**eration for **J**ava



Handling Parameter-caused Exceptions



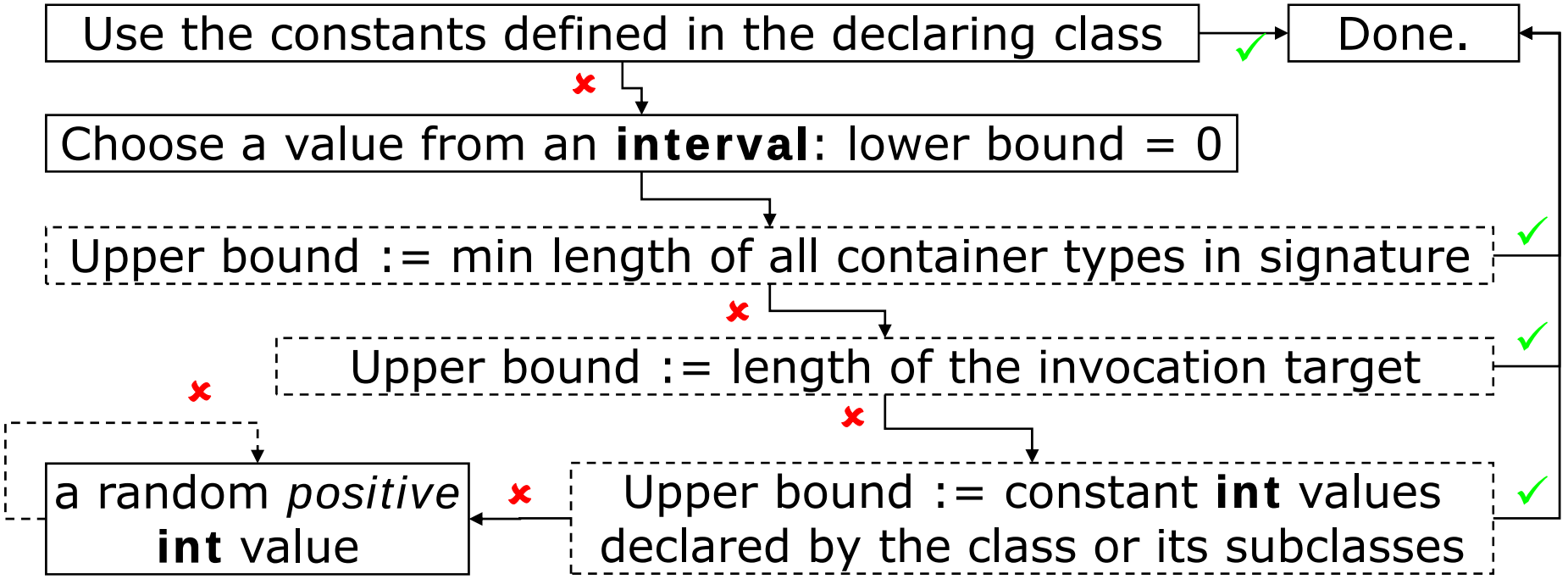
- Exception causes determination and evaluation:
  - example: an **`ArrayIndexOutOfBoundException`**
    $\Rightarrow$ an **`int`**-typed parameter shall match an array parameter
  - **HeuriGenJ formally encodes such rules**
  - heuristically estimating *which* **`int`**-typed parameter in a signature is causing the exception

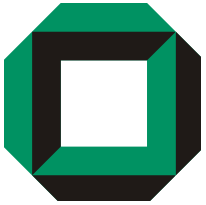
- In general, invocation targets are also considered
  - e.g. **`IndexOutOfBoundsException`** for **`str.substring(int index)`** $\rightarrow$ create **`str`** with sufficient length

Heuristical Parameter Generator (HPG)

- Parameter generation is nondeterministic
- A set of heuristics was created to avoid exceptions
- Basic idea: formalise rules of parameter constraints

- A multi-stage heuristic for generation of **`int`** primitives:





Evaluation

- Visible, implemented methods: public, non-abstract

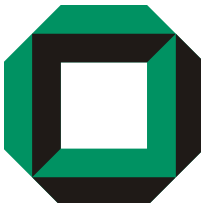
- **Coverage**: the number and share of methods for which arguments are successfully generated
  - without human intervention or external data sources
- **Effectiveness**: the number and share of exceptions at runtime successfully handled by HeuriGenJ
- **Effort**: the duration of the parameter generation

- **Targets**: `java.util` and `java.lang` API packages
- **Setting**: Intel Pentium Dual-Core 1.8 GHz, 1 GB main memory, Windows Vista with JRE 1.6.0_03

Evaluation

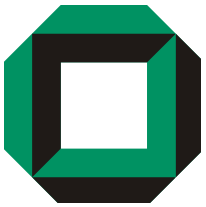
Package	#Public non-abstract classes	#Public non-abstract methods	#Executed methods w/o exceptions	Execution success rate
java.util	58	738	668	90.51%
java.lang	76	861	790	91.75%

Package	Total duration of arguments generation	#Thrown runtime exceptions	#Handled runtime exceptions	Exception handling success rate
java.util	169 s	160	95	59.37%
java.lang	259 s	204	151	74.01%



Related Work
Automated Parameter Generation

- Eclat [7]: random generation, based on execution results
  - needs an existing correct execution to start generation
- RANDOOP [8] extends Eclat, e.g. with a pool of values
  - from which an initial random input is computed
- Godefroid et al. [9] present a symbolic execution approach
  - uses random input generation (also called ”concolic execution”)
- Majumdar and Xu [10]: **[7-9] are problematic in practice**
  - randomly generated parameters are in *most* cases meaningless for the program execution (out of scope; lead to exceptions)
  - hence, valid inputs must be specified by a context-free grammar



Assumptions & Limitations

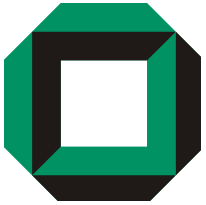
SDQ Software Design and Quality

Assumptions

- Exceptions must be available to HeuriGenJ
- SQL-like **`String`** parameters cannot be generated

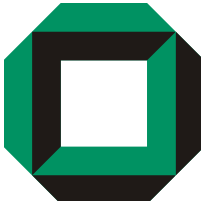
Limitations

- Parameter generation not applied to methods/APIs
  - … if they may destroy sensitive data (e.g. using JDBC)
  - … if they produce a security issue or undesired effects (e.g. the **`java.lang.System.exit`** method)
  - Excluding such API parts is not automated in HeuriGenJ
- Usability in benchmarking: to be demonstrated



Future Work

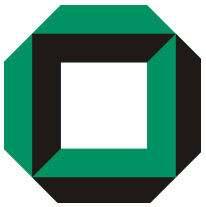
- Generate parameters w.r.t. quantification of parameter performance dependencies
  - several input sets per signature (sensitivity analysis)
  - vary invocation target objects
- Enhance the heuristical parameter generation
  - by incorporating machine learning techniques
  - using search-based software engineering techniques
  - use recorded parameters (e.g. with ByCounter [11])
- Integrate into benchmarking/perf. prediction
- Port to .NET CLR and other environments



Conclusions

- **HeuriGenJ: a novel approach for input parameter generation using heuristics**
- For methods and constructors with implementation in Java bytecode
- Handles runtime exceptions if they occur and learns from them
- Saves execution information and parameter values in a database
- The principles of HeuriGenJ applicable to other languages and APIs, e.g. .NET platform API

Thanks for your attention! Questions?



References

[1] Michael Kuperberg and Steffen Becker. *Predicting Software Component Performance: On the Relevance of Parameters for Benchmarking Bytecode and APIs*. WCOP 2007.

[2] Michael Kuperberg, Klaus Krogmann, Ralf Reussner: *Performance Prediction for Black-Box Components using Reengineered Parametric Behaviour Models*. CBSE 2008 / LNCS

[3] Xiaolan Zhang and Margo Seltzer. *HBench:Java: an application-specific benchmarking framework for Java virtual machines*. ACM 2000 conference on Java Grande, pages 62–70, New York, NY, USA, 2000.

[4] *The Java Grande Forum Sequential Benchmarks 2.0, 2007*, last visit: June 9th, 2008. URL: http://www2.epcc.ed.ac.uk/computing/research activities/java grande.

[5] *Linpack Benchmark (Java Version)*, 2007. URL:http://www.netlib.org/benchmark/linpackjava/, last visit: June 9th, 2008.

[6] *Java SciMark 2.0*, 2007. URL: http://math.nist.gov/scimark2/, last visit: June 9th, 2008.

[7] Carlos Pacheco and Michael D. Ernst. *Eclat: Automatic generation and classification of test inputs*. In 19th European Conference Object-Oriented Programming, pages 504–527, 2005.

[8] Carlos Pacheco and Michael D. Ernst. *Randoop: feedback-directed random testing for Java*. In OOPSLA '07, pages 815–816, New York, NY, USA, 2007. ACM Press.

[9] Patrice Godefroid, Nils Klarlund, and Koushik Sen. *Dart: directed automated random testing*. PLDI 2005.

[10] Rupak Majumdar and Ru-Gang Xu. *Directed test generation using symbolic grammars*. ESEC-FSE 2007.

[11] Michael Kuperberg, Martin Krogmann, and Ralf Reussner. *ByCounter: Portable Runtime Counting of Bytecode Instructions and Method Invocations*. In BYTECODE'08, 2008.