Is it worth testing all configurations?

A Case Study (JHipster)

Axel Halin, Alexandre Nuttinck, Mathieu Acher,
Xavier Devroey, Gilles Perrouin, and Benoit Baudry

https://arxiv.org/abs/1710.07980

Configurable Software Systems

- Most of software is configurable (Linux, Firefox, Eclipse, ffmpeg, JHipster...)
  - compilation options, command line parameters, configuration files
  - different needs in terms of functionalities and performance; self-adaptation @ run.time
  - **configur-ability or variability**: a good source for synthesizing variants (diversity)
- Software is working (sometimes)
  - yes but perhaps for one specific configuration (the default one)
  - is it working for **all configurations**?

```
? (3/15) Which *type* of authentication would you like to use? (Use arrow keys)
❯ HTTP Session Authentication (stateful, default Spring Security mechanism)
  HTTP Session Authentication with social login enabled (Google, Facebook, Twitter).
  OAuth2 Authentication (stateless, with an OAuth2 server implementation)
  Token-based authentication (stateless, with a token)
```

```
? (7/15) Do you want to use Hibernate 2nd level cache?
  No
  Yes, with ehcache (local cache, for a single node)
❯ Yes, with HazelCast (distributed cache, for multiple nodes)
```

Configurable Software Systems

- At each modification/commit/push/release, do you test all configurations?
  - No and you certainly have very good reasons
    - needs lots of resources (machines!); don't want to burn the planet
    - needs an engineering effort to instrument testing of all configurations
    - the number of configurations is too important (eg 2^14000 for Linux)

```
? (3/15) Which *type* of authentication would you like to use? (Use arrow keys)
❯ HTTP Session Authentication (stateful, default Spring Security mechanism)
  HTTP Session Authentication with social login enabled (Google, Facebook, Twitter).
  OAuth2 Authentication (stateless, with an OAuth2 server implementation)
  Token-based authentication (stateless, with a token)
```

```
? (7/15) Do you want to use Hibernate 2nd level cache?
  No
  Yes, with ehcache (local cache, for a single node)
❯ Yes, with HazelCast (distributed cache, for multiple nodes)
```

```
44      @Autowired(required = false)
45  +   private MetricRegistry metricRegistry;<% if (clusteredHttpSession == 'hazelcast' || hibernateCache == 'hazelcast') { %>
```

```
75              FilterRegistration.Dynamic hazelcastWebFilter = servletContext.addFilter("hazelcastWebFilter", new
    SpringAwareWebFilter());
76              Map<String, String> parameters = new HashMap<>();
77  +           parameters.put("instance-name", hazelcastInstance.getName());
78              // Name of the distributed map storing your web session objects
79              parameters.put("map-name", "clustered-http-sessions");
80
```

Configurable Software Systems

- At each modification/commit/push/release, do you test all configurations?
  - No since too much resources and effort (impossible and unpractical)
  - **“Sampling”** techniques (subset of configurations)
    - Apel et al. ICSE’16, Kaestner et al. ICSE’14 and ASE’16, Ana B. Sánchez et al. SoSyM 2017, Perrouin et al. ICST’10, Cohen et al. TSE’06, Henard et al. TSE’14, etc.
    - Many papers at SPLC, FSE, ASE, ICSE, ESE, TSE on this topic
  - Evaluations were carried out on a small subset of the total number of configurations or faults, constituting a threat to validity

What is the cost-effective sampling strategy to test configurations of a system?

Is it Worth testing All Configurations?

Testing with the community

ALL

Sampling

What is the cost-effective sampling strategy to test configurations of a system?

Empirical studies needed!

Approach: testing all configurations on a real-world case

1. In prior works, the testing cost can only be estimated. We typically **ignore the exact computational cost** (e.g., time needed) or **how difficult** it is to instrument testing for any configuration (RQ1+RQ3).
2. An unique opportunity to accurately assess the error-detection capabilities of sampling techniques with a **ground truth.** (RQ2)

What is the cost-effective sampling strategy to test configurations of a system?

Research method: Case Study
Approach: first to exhaustively test configurations
Results: quantitative insights (ground truth) and qualitative insights (incl. human effort and discussions with developers)

Case study: JHipster

- Web-apps generator
  - Spring-Boot
  - Bootstrap / AngularJS
  - 100 % Open Source

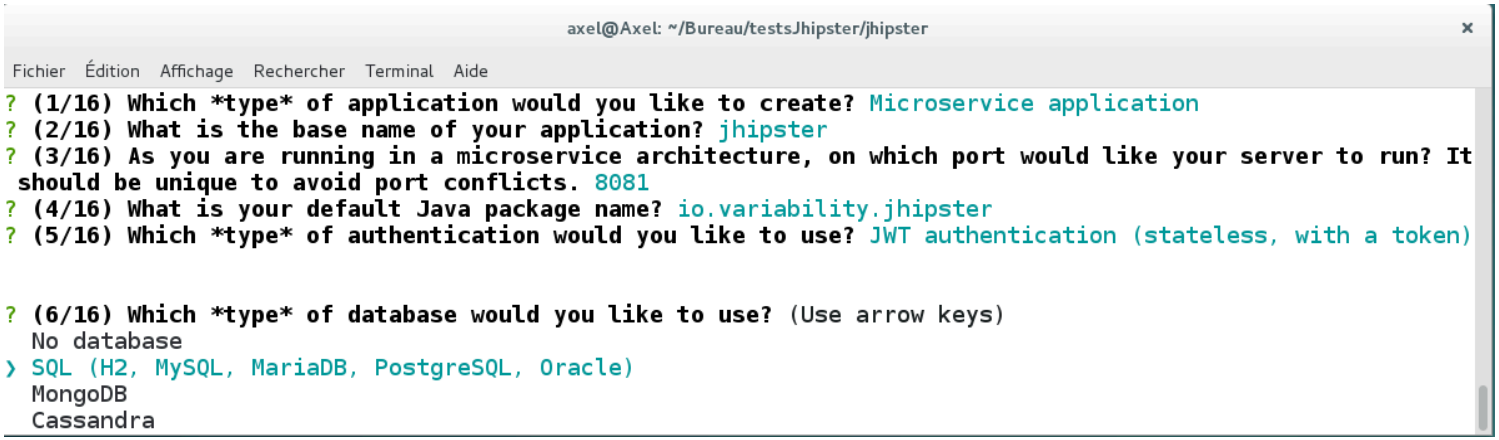
- Yeoman
  - Bower, npm
  - yo
- Used all over the world
  - Large companies (HBO, Google, Adobe)[1]
  - Independent developers
  - Our students
- GitHub
  - 6000+ stars
  - 118 releases (JHipster 3.6.1, 18 Aug 2016)
  - 300+ contributors

[1] https://jhipster.github.io/companies-using-jhipster/

Case study: JHipster

- Configuration
  - Command-line interface
  - *.yo-rc.json*
- Generation
  - Dependencies fetching
  - Yeoman templates

```
(...)
when: function (response) {
    return applicationType === 'microservice';
  },
  type: 'list',
  name: 'databaseType',
  message: function (response) {
      return getNumberedQuestion('Which *type* of database would
          you like to use?', applicationType ===
          'microservice');},
  choices: [
      {value: 'no', name: 'No database'},
      {value: 'sql', name: 'SQL (H2, MySQL, MariaDB, PostgreSQL,
          Oracle)'},
      {value: 'mongodb', name: 'MongoDB'},
      {value: 'cassandra',name: 'Cassandra'}
  ],
  default: 1
(...)
```



Research questions

(RQ1) What is the cost of testing all configurations of JHipster?

(RQ2) How many and what kinds of failures/ faults can be found in all configurations?

(RQ3) What is the most cost-effective sampling strategy? What are the recommendations for the JHipster project?

(RQ1) What is the cost of testing all configurations of JHipster?

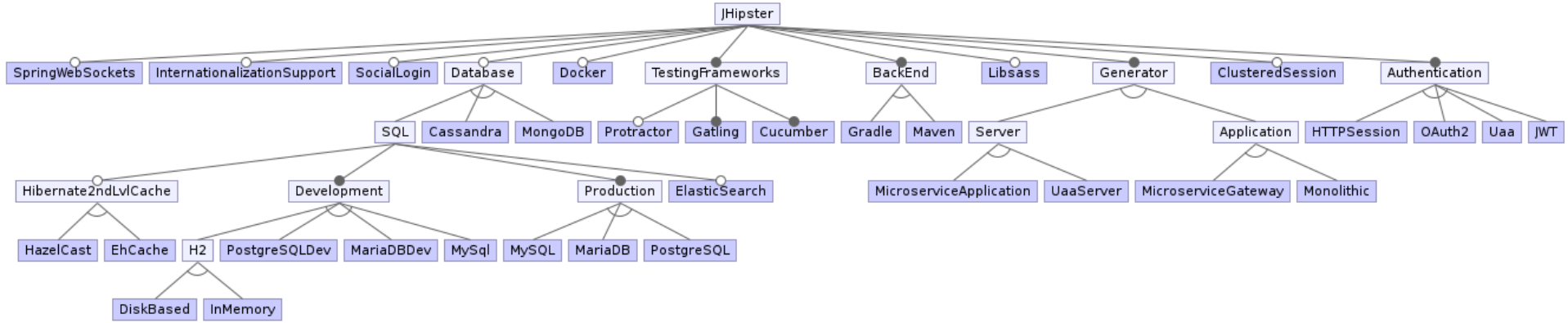

```
? (3/15) Which *type* of authentication would you like to use? (Use arrow keys)
❯ HTTP Session Authentication (stateful, default Spring Security mechanism)
  HTTP Session Authentication with social login enabled (Google, Facebook, Twitter).
  OAuth2 Authentication (stateless, with an OAuth2 server implementation)
  Token-based authentication (stateless, with a token)
```

```
? (7/15) Do you want to use Hibernate 2nd level cache?
  No
  Yes, with ehcache (local cache, for a single node)
❯ Yes, with HazelCast (distributed cache, for multiple nodes)
```

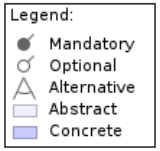
RQ1: What is the "cost" of testing all configurations of JHipster?

Variability Modeling

- **26256** different variants for the refined FM
  - excluded client and server standalone variants
  - excluded variants relying on an Oracle database
  - included all three testing frameworks in all variants
  - added Docker for the build

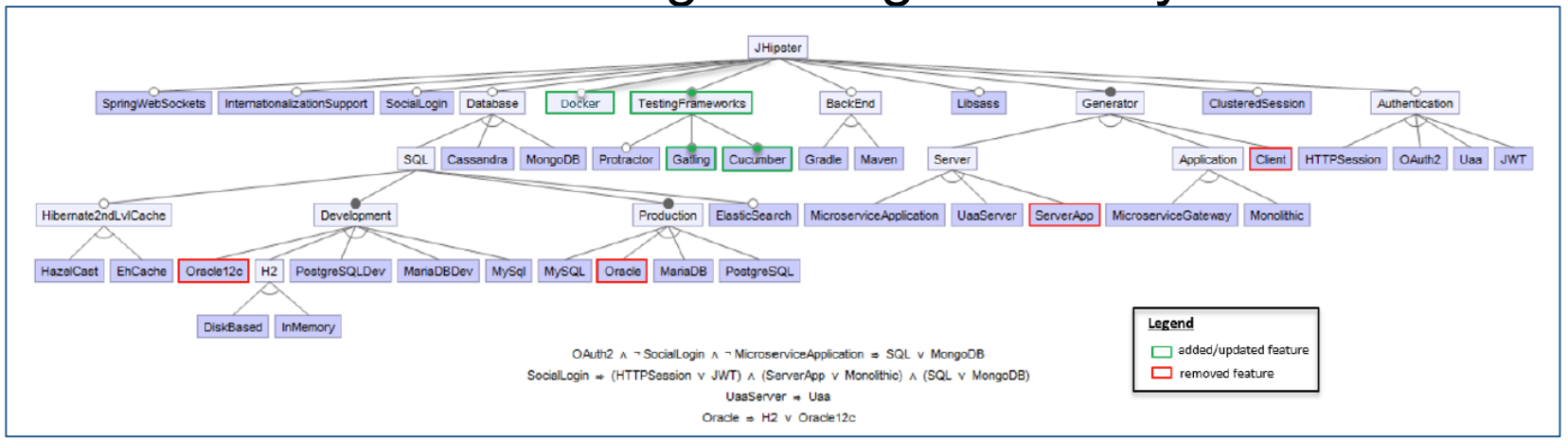


$$\text{OAuth2} \wedge \neg \text{SocialLogin} \wedge \neg \text{MicroserviceApplication} \Rightarrow \text{SQL} \vee \text{MongoDB}$$
$$\text{SocialLogin} \Rightarrow (\text{HTTPSession} \vee \text{JWT}) \wedge \text{Monolithic} \wedge (\text{SQL} \vee \text{MongoDB})$$
$$\text{UaaServer} \Rightarrow \text{Uaa}$$
$$\neg \text{OAuth2} \wedge \neg \text{SocialLogin} \wedge \neg \text{MicroserviceApplication} \Rightarrow \text{SQL} \vee \text{MongoDB} \vee \text{Cassandra}$$
$$\text{Server} \Rightarrow \neg \text{Protractor}$$
$$\neg \text{Server} \Rightarrow \text{Protractor}$$
$$\text{MySQL} \Rightarrow \text{H2} \vee \text{MySql}$$
$$\text{MicroserviceApplication} \vee \text{MicroserviceGateway} \Rightarrow \text{JWT} \vee \text{Uaa}$$
$$\text{Monolithic} \Rightarrow \text{JWT} \vee \text{HTTPSession} \vee \text{OAuth2}$$
$$\text{MariaDB} \Rightarrow \text{H2} \vee \text{MariaDBDev}$$
$$\text{PostgreSQL} \Rightarrow \text{H2} \vee \text{PostgreSQLDev}$$
$$\text{SpringWebSockets} \vee \text{ClusteredSession} \Rightarrow \text{Application}$$
$$\text{Libsass} \Rightarrow \text{Application}$$

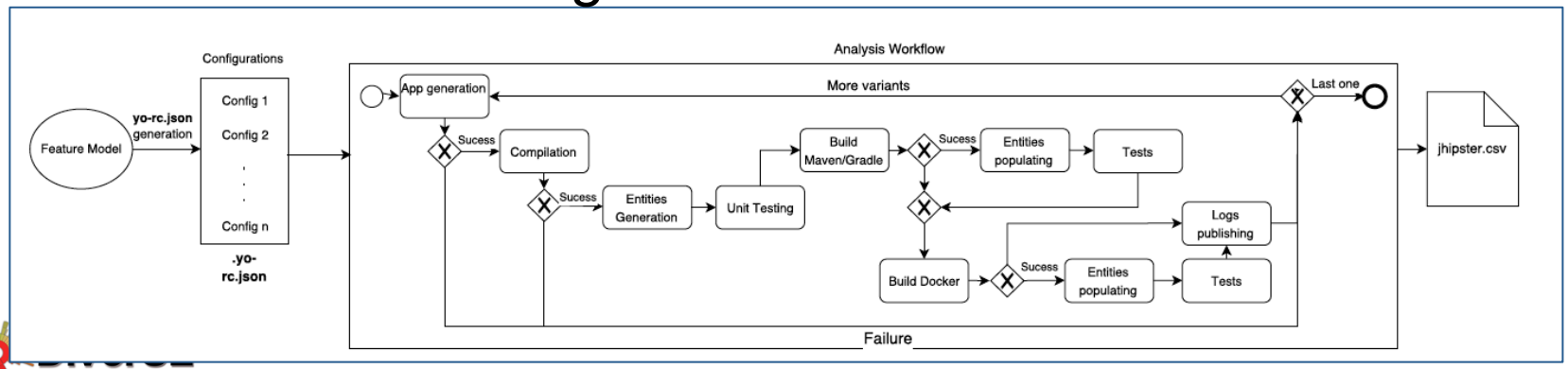


(RQ1.1) What is the (human) cost of engineering an infrastructure capable of automatically deriving and testing all configurations?

From reverse engineering variability models...

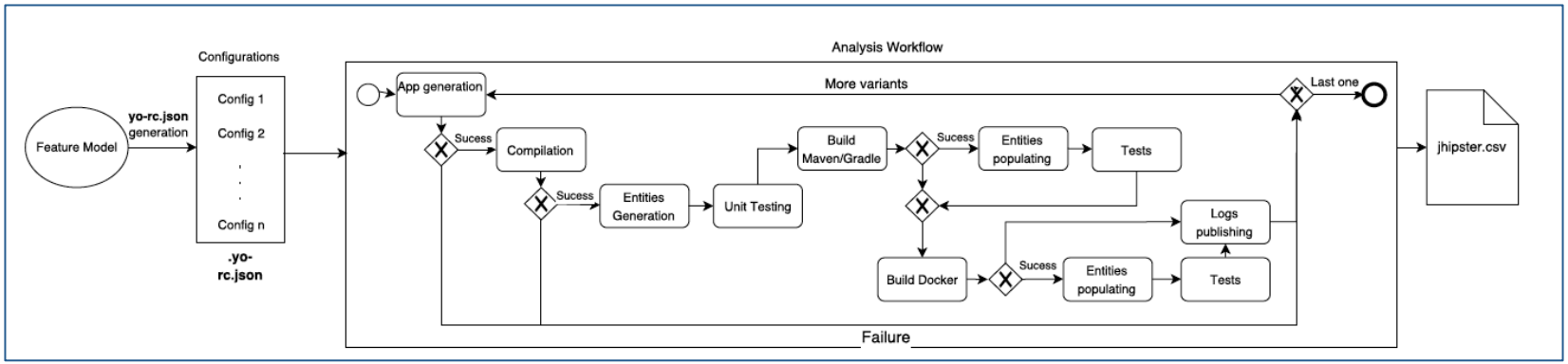


...to fully automated and distributed testing of 26K+ configurations: 8 man/month



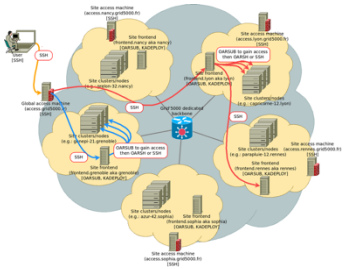
(RQ1.1) What is the human cost of engineering an infrastructure capable of automatically deriving and testing all configurations?

- Reverse engineering variability models
- The **testing infrastructure is itself a configurable system**
  - testing procedures applicable to all configurations
  - “all-inclusive” testing environment
  - distributing the tests
  - opportunistic optimizations and sharing
- 8 man/month (2 MSc students during 4 months, full time)
- Most difficult task: **validation of the testing infrastructure** (numerous iterations based on “tries and errors”)



(RQ1.2): What are the computational resources needed to test all configurations?

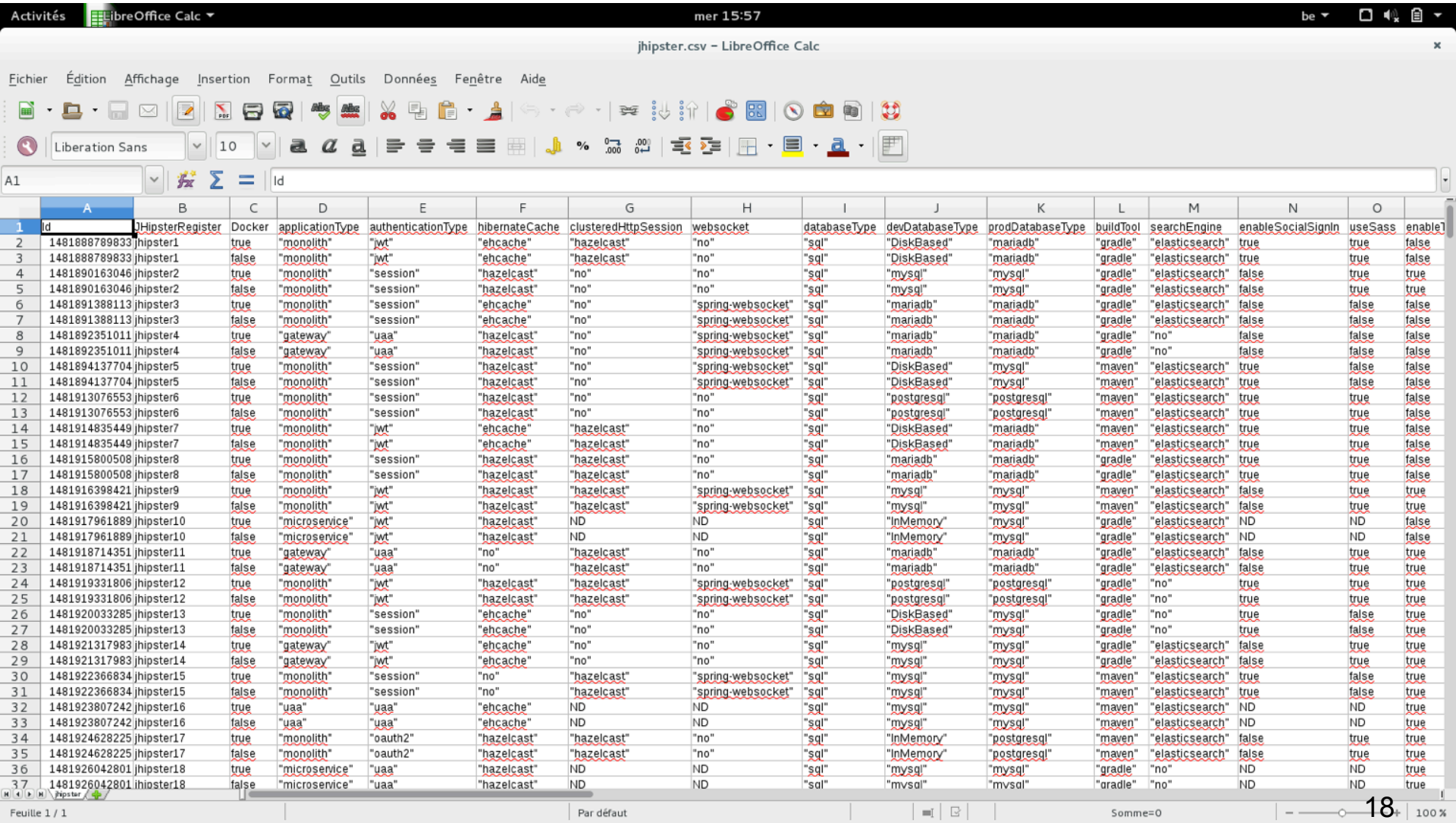
- Grid'5000 ("french Grid", distributed infrastructure with thousands of machines for scientific experiments)
- 80 machines for 4 periods (nights) of 13 hours
- 6 configurations ~ 60 minutes
- 26K+ configurations: **4376 hours/machine**
- Each configuration: between 400Mb and 450Mb
- 5.2 terabytes in total



RQ1: What is the cost of testing all configurations of JHipster? (wrap up)

- 8 man/month
  - challenge ahead: follow the evolution of JHipster
- Despite optimizations (eg pre-caching of dependencies), 26K+ configurations:
  - 4376 hours/machine
  - 5.2 terabytes in total
- Is it worth?
  - Yes from the researcher point of view (insights on the cost of testing configurations)
- Is it worth for practitioners?
  - Moment ;)

Snapshot of JHipster CSV file...



Id	JHipsterRegister	Docker	applicationType	authenticationType	hibernateCache	clusteredHttpSession	websocket	databaseType	devDatabaseType	prodDatabaseType	buildTool	searchEngine	enableSocialSignIn	useSass	enable
1481888789833	jhipster1	true	"monolith"	"jwt"	"ehcache"	"hazelcast"	"no"	"sql"	"DiskBased"	"mariadb"	"gradle"	"elasticsearch"	true	true	false
1481888789833	jhipster1	false	"monolith"	"jwt"	"ehcache"	"hazelcast"	"no"	"sql"	"DiskBased"	"mariadb"	"gradle"	"elasticsearch"	true	true	false
1481890163046	jhipster2	true	"monolith"	"session"	"hazelcast"	"no"	"no"	"sql"	"mysql"	"mysql"	"gradle"	"elasticsearch"	false	true	true
1481890163046	jhipster2	false	"monolith"	"session"	"hazelcast"	"no"	"no"	"sql"	"mysql"	"mysql"	"gradle"	"elasticsearch"	false	true	true
1481891388113	jhipster3	true	"monolith"	"session"	"ehcache"	"no"	"spring-websocket"	"sql"	"mariadb"	"mariadb"	"gradle"	"elasticsearch"	false	false	false
1481891388113	jhipster3	false	"monolith"	"session"	"ehcache"	"no"	"spring-websocket"	"sql"	"mariadb"	"mariadb"	"gradle"	"elasticsearch"	false	false	false
1481892351011	jhipster4	true	"gateway"	"uaa"	"hazelcast"	"no"	"spring-websocket"	"sql"	"mariadb"	"mariadb"	"gradle"	"no"	false	false	false
1481892351011	jhipster4	false	"gateway"	"uaa"	"hazelcast"	"no"	"spring-websocket"	"sql"	"mariadb"	"mariadb"	"gradle"	"no"	false	false	false
1481894137704	jhipster5	true	"monolith"	"session"	"hazelcast"	"no"	"spring-websocket"	"sql"	"DiskBased"	"mysql"	"maven"	"elasticsearch"	true	false	false
1481894137704	jhipster5	false	"monolith"	"session"	"hazelcast"	"no"	"spring-websocket"	"sql"	"DiskBased"	"mysql"	"maven"	"elasticsearch"	true	false	false
1481913076553	jhipster6	true	"monolith"	"session"	"hazelcast"	"no"	"no"	"sql"	"postgresql"	"postgresql"	"maven"	"elasticsearch"	true	true	false
1481913076553	jhipster6	false	"monolith"	"session"	"hazelcast"	"no"	"no"	"sql"	"postgresql"	"postgresql"	"maven"	"elasticsearch"	true	true	false
1481914835449	jhipster7	true	"monolith"	"jwt"	"ehcache"	"hazelcast"	"no"	"sql"	"DiskBased"	"mariadb"	"maven"	"elasticsearch"	true	true	false
1481914835449	jhipster7	false	"monolith"	"jwt"	"ehcache"	"hazelcast"	"no"	"sql"	"DiskBased"	"mariadb"	"maven"	"elasticsearch"	true	true	false
1481915800508	jhipster8	true	"monolith"	"session"	"hazelcast"	"hazelcast"	"no"	"sql"	"mariadb"	"mariadb"	"gradle"	"elasticsearch"	true	true	false
1481915800508	jhipster8	false	"monolith"	"session"	"hazelcast"	"hazelcast"	"no"	"sql"	"mariadb"	"mariadb"	"gradle"	"elasticsearch"	true	true	false
1481916398421	jhipster9	true	"monolith"	"jwt"	"hazelcast"	"hazelcast"	"spring-websocket"	"sql"	"mysql"	"mysql"	"maven"	"elasticsearch"	false	true	true
1481916398421	jhipster9	false	"monolith"	"jwt"	"hazelcast"	"hazelcast"	"spring-websocket"	"sql"	"mysql"	"mysql"	"maven"	"elasticsearch"	false	true	true
1481917961889	jhipster10	true	"microservice"	"jwt"	"hazelcast"	ND	ND	"sql"	"InMemory"	"mysql"	"gradle"	"elasticsearch"	ND	ND	false
1481917961889	jhipster10	false	"microservice"	"jwt"	"hazelcast"	ND	ND	"sql"	"InMemory"	"mysql"	"gradle"	"elasticsearch"	ND	ND	false
1481918714351	jhipster11	true	"gateway"	"uaa"	"no"	"hazelcast"	"no"	"sql"	"mariadb"	"mariadb"	"gradle"	"elasticsearch"	false	true	true
1481918714351	jhipster11	false	"gateway"	"uaa"	"no"	"hazelcast"	"no"	"sql"	"mariadb"	"mariadb"	"gradle"	"elasticsearch"	false	true	true
1481919331806	jhipster12	true	"monolith"	"jwt"	"hazelcast"	"hazelcast"	"spring-websocket"	"sql"	"postgresql"	"postgresql"	"gradle"	"no"	true	true	true
1481919331806	jhipster12	false	"monolith"	"jwt"	"hazelcast"	"hazelcast"	"spring-websocket"	"sql"	"postgresql"	"postgresql"	"gradle"	"no"	true	true	true
1481920033285	jhipster13	true	"monolith"	"session"	"ehcache"	"no"	"no"	"sql"	"DiskBased"	"mysql"	"gradle"	"no"	true	false	true
1481920033285	jhipster13	false	"monolith"	"session"	"ehcache"	"no"	"no"	"sql"	"DiskBased"	"mysql"	"gradle"	"no"	true	false	true
1481921317983	jhipster14	true	"gateway"	"jwt"	"ehcache"	"no"	"no"	"sql"	"mysql"	"mysql"	"gradle"	"elasticsearch"	false	true	true
1481921317983	jhipster14	false	"gateway"	"jwt"	"ehcache"	"no"	"no"	"sql"	"mysql"	"mysql"	"gradle"	"elasticsearch"	false	true	true
1481922366834	jhipster15	true	"monolith"	"session"	"no"	"hazelcast"	"spring-websocket"	"sql"	"mysql"	"mysql"	"maven"	"elasticsearch"	true	false	true
1481922366834	jhipster15	false	"monolith"	"session"	"no"	"hazelcast"	"spring-websocket"	"sql"	"mysql"	"mysql"	"maven"	"elasticsearch"	true	false	true
1481923807242	jhipster16	true	"uaa"	"uaa"	"ehcache"	ND	ND	"sql"	"mysql"	"mysql"	"maven"	"elasticsearch"	ND	ND	true
1481923807242	jhipster16	false	"uaa"	"uaa"	"ehcache"	ND	ND	"sql"	"mysql"	"mysql"	"maven"	"elasticsearch"	ND	ND	true
1481924628225	jhipster17	true	"monolith"	"oauth2"	"hazelcast"	"hazelcast"	"no"	"sql"	"InMemory"	"postgresql"	"maven"	"elasticsearch"	false	true	true
1481924628225	jhipster17	false	"monolith"	"oauth2"	"hazelcast"	"hazelcast"	"no"	"sql"	"InMemory"	"postgresql"	"maven"	"elasticsearch"	false	true	true
1481926042801	jhipster18	true	"microservice"	"uaa"	"hazelcast"	ND	ND	"sql"	"mysql"	"mysql"	"gradle"	"no"	ND	ND	true
1481926042801	jhipster18	false	"microservice"	"uaa"	"hazelcast"	ND	ND	"sql"	"mysql"	"mysql"	"gradle"	"no"	ND	ND	true

RQ2.1: How many and what kinds of failures/ faults can be found in all configurations?

Out of the 26,256 configurations, 9376 failed.
(224 during compilation, 9152 at execution time)

36% failures

RQ2.1: How many and what kinds of failures/ faults can be found in all configurations?

Out of the 26,256 configurations, 9376 failed. **36%**
Individual features do not explain this high percentage.

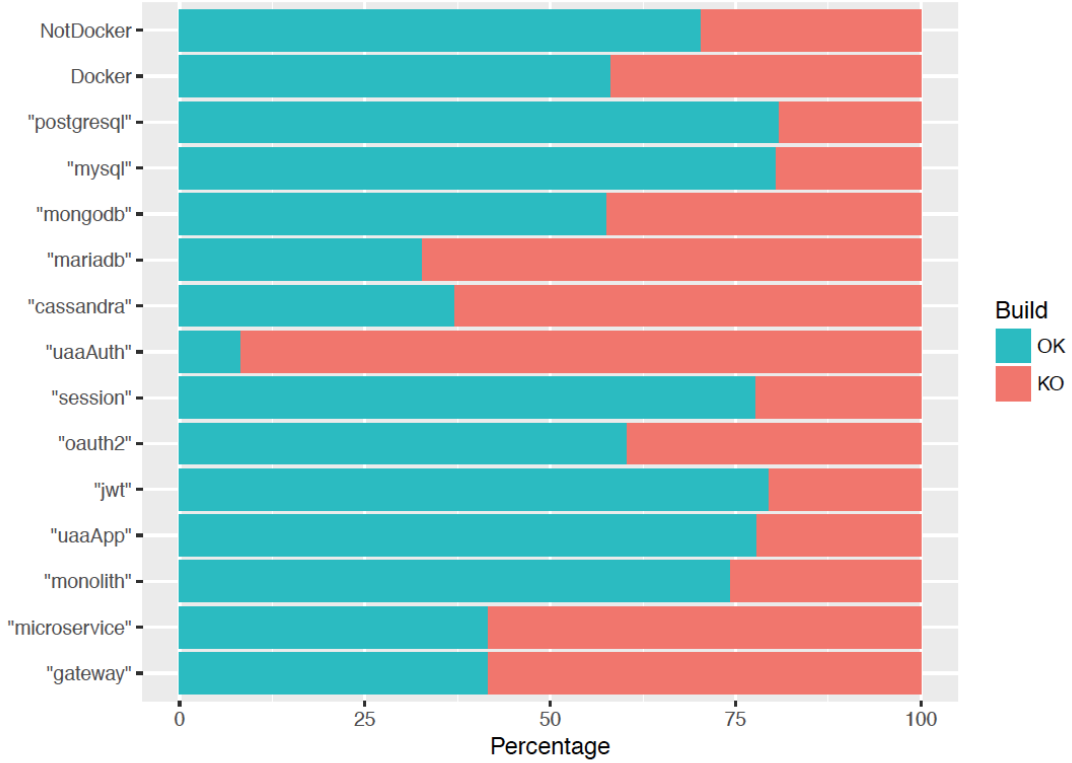
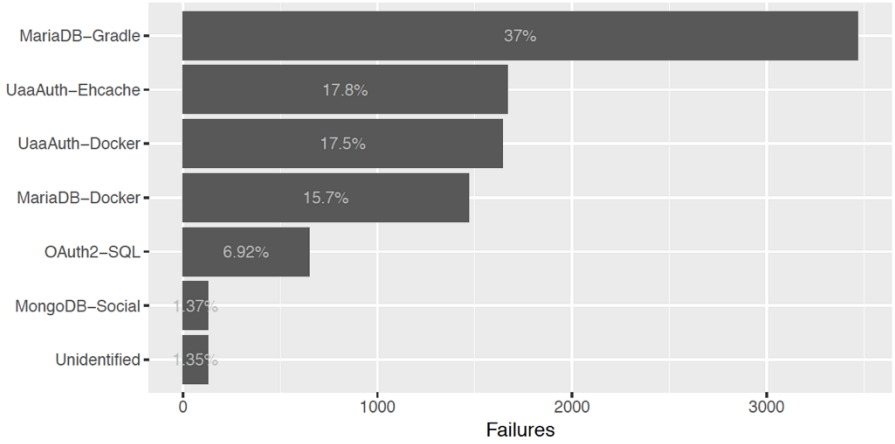


Figure 3: Proportion of build failure by Feature

RQ2.1: How many and what kinds of failures/ faults can be found in all configurations?

Statistical and manual analysis show that only **6 feature interactions** explain most of the 36% failures

Others: only 0.4% of failures (interactions w/ 4+ features)



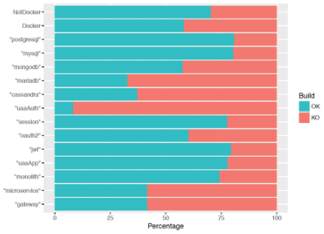


Figure 3: Proportion of build failure by Feature

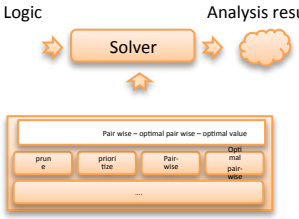
Figure 4: Proportion of failures by fault

Left-hand side	Right-hand side	Support	Confidence	Github Issue	Report/Correction date
DatabaseType="mongodb", EnableSocialSignIn=true	Compile=KO	0.488 %	1	4037	27 Aug 2016 (report and fix for milestone 3.7.0)
prodDatabaseType="mariadb", buildTool="gradle"	Build=KO	16.179 %	1	4222	27 Sep 2016 (report and fix for milestone 3.9.0)
Docker=true, authenticationType="uaa"	Build=KO	6.825 %	1	UAA is in Beta	Not corrected
authenticationType="uaa", hibernateCache = "no"	Build=KO	2.438 %	1	4225	28 Sep 2016 (report and fix for milestone 3.9.0)
authenticationType="uaa", hibernateCache = "ehcache"	Build=KO	2.194 %	1	4225	28 Sep 2016 (report and fix for milestone 3.9.0)
prodDatabaseType="mariadb", applicationType = "monolith", searchEngine = "false", Docker = "true"	Build=KO	5.59%	1	4543	24 Nov 2016 (report and fix for milestone 3.12.0)

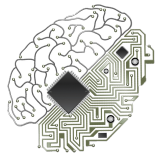
Table 1: Association rules involving compilation and build failures

(RQ2.2) How effective are sampling techniques comparatively?

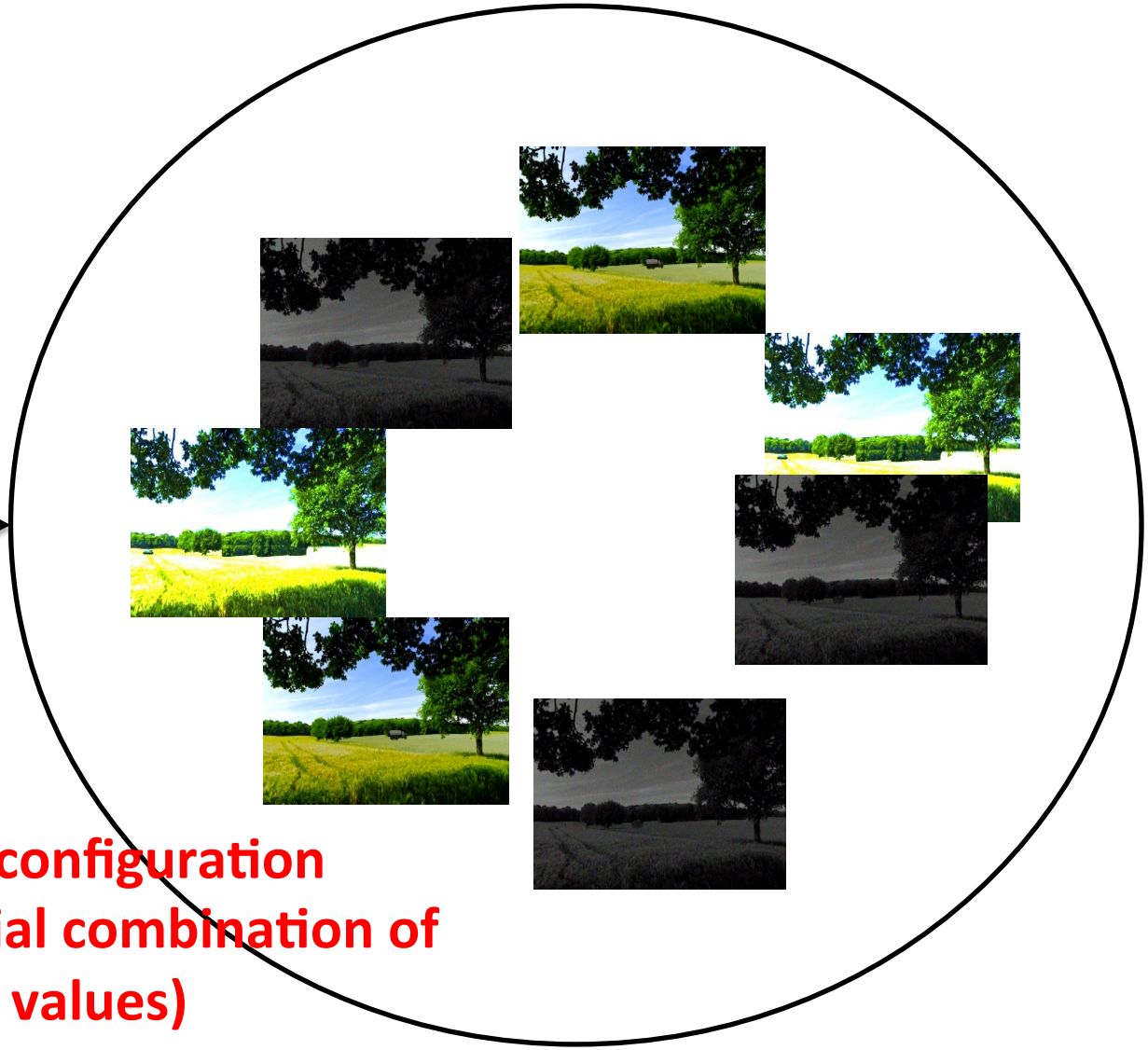
First: Some Background about Sampling Configurations



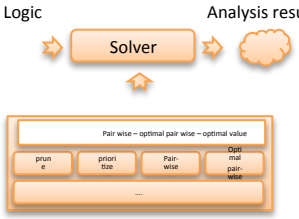
Reasoning operations



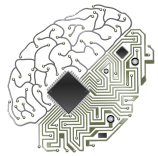
Abstraction of



Too many configuration (exponential combination of parameter values)



Reasoning operations

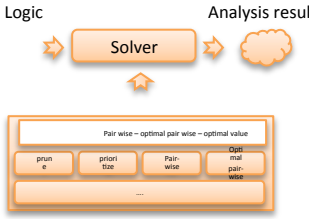


Possible strategies:

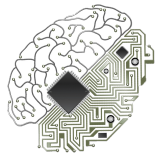
*** constraints**

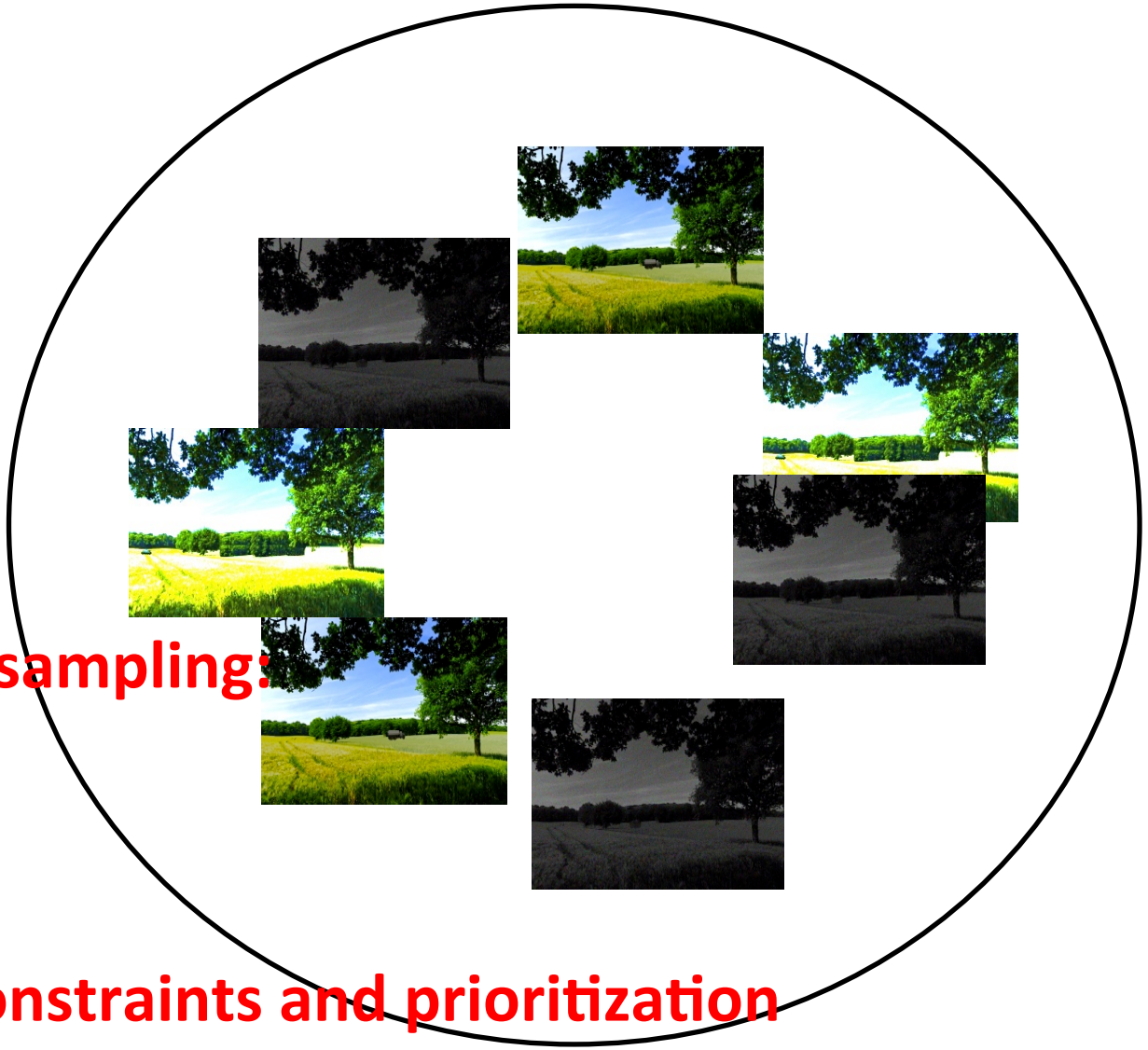
*** priority over some parameters**

*** irrelevant parameters**



Reasoning operations





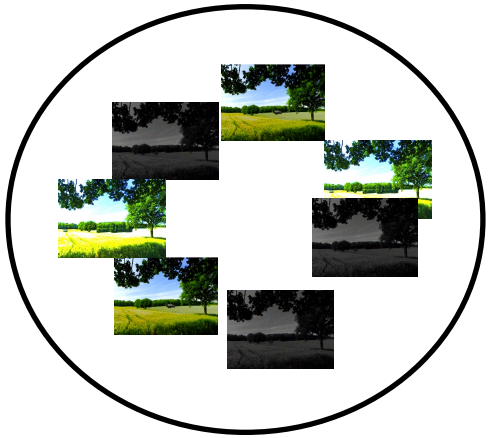
Then configurations sampling:

* **random**
* **pair-wise**
* **T-wise**

with respect of constraints and prioritization

Configuration Sampling

- Number of configurations dramatically grows with the number of parameters
- Select a subset of combinations



- Key idea: focus on specific interactions among values for parameters
  - Pair-wise interactions
  - T-wise interactions

Configuration Sampling

- $X_1,\ldots, X_n$ n parameters
- $\forall\, i \in [1..n]\ X_i \subset \{V_{i1},\ldots,V_{im}\}$
  - m can be different for every Xi
- A configuration is a set of values for each X_i
- Pairwise covering
  - A set TC of configurations such that all values for every pair of parameters are in one configuration
  - $\forall\, X_j, X_k \mid \forall\, X_{ja}, X_{kb} \mid \exists\, c \in TC \mid TC \subset X_{ja}, X_{kb}$
- T-wise covering
  - Generalization to all tuples of parameters

Illustrative Example

Characteristics	Values			
Background	CountryS	Desert	Urban	Forest
Luminosity	Normal	High	Low	
Speed	0.1	0.2	0.3	
Detractors Level	0.2	0.4	0.6	

There are **108** combinations of factors

12 combinations can satisfy pair-wise

Example

CountryS	normal	0.1	0.2
CountryS	High	0.2	0.4
CountryS	Low	0.3	0.6
Urban	normal	0.3	0.4
Urban	High	0.1	0.2
Urban	Low	0.2	0.6
Desert	normal	0.2	0.6
Desert	High	0.3	0.2
Desert	Low	0.1	0.4
Forest	normal	*	0.4
Forest	High	0.1	0.6
Forest	Low	0.2	0.2

Example

CountryS	normal	0.1	0.2
CountryS	High	0.2	0.4
CountryS	Low	0.3	0.6
Urban	normal	0.3	0.4
Urban	High	0.1	0.2
Urban	Low	0.2	0.6
Desert	normal	0.2	0.6
Desert	High	0.3	0.2
Desert	Low	0.1	0.4
Forest	normal	*	0.4
Forest	High	0.1	0.6
Forest	Low	0.2	0.2

Example

CountryS	normal	0.1	0.2
CountryS	High	0.2	0.4
CountryS	Low	0.3	0.6
Urban	normal	0.3	0.4
Urban	High	0.1	0.2
Urban	Low	0.2	0.6
Desert	normal	0.2	0.6
Desert	High	0.3	0.2
Desert	Low	0.1	0.4
Forest	normal	*	0.4
Forest	High	0.1	0.6
Forest	Low	0.2	0.2

Example

CountryS	normal	0.1	0.2
CountryS	High	0.2	0.4
CountryS	Low	0.3	0.6
Urban	normal	0.3	0.4
Urban	High	0.1	0.2
Urban	Low	0.2	0.6
Desert	normal	0.2	0.6
Desert	High	0.3	0.2
Desert	Low	0.1	0.4
Forest	normal	*	0.4
Forest	High	0.1	0.6
Forest	Low	0.2	0.2

Example

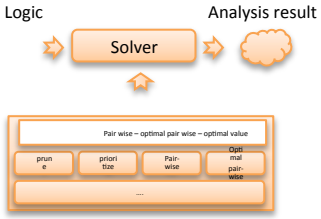
CountryS	normal	0.1	0.2
CountryS	High	0.2	0.4
CountryS	Low	0.3	0.6
Urban	normal	0.3	0.4
Urban	High	0.1	0.2
Urban	Low	0.2	0.6
Desert	normal	0.2	0.6
Desert	High	0.3	0.2
Desert	Low	0.1	0.4
Forest	normal	*	0.4
Forest	High	0.1	0.6
Forest	Low	0.2	0.2

Example

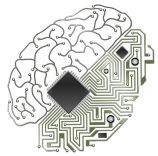
CountryS	normal	0.1	0.2
CountryS	High	0.2	0.4
CountryS	Low	0.3	0.6
Urban	normal	0.3	0.4
Urban	High	0.1	0.2
Urban	Low	0.2	0.6
Desert	normal	0.2	0.6
Desert	High	0.3	0.2
Desert	Low	0.1	0.4
Forest	normal	*	0.4
Forest	High	0.1	0.6
Forest	Low	0.2	0.2

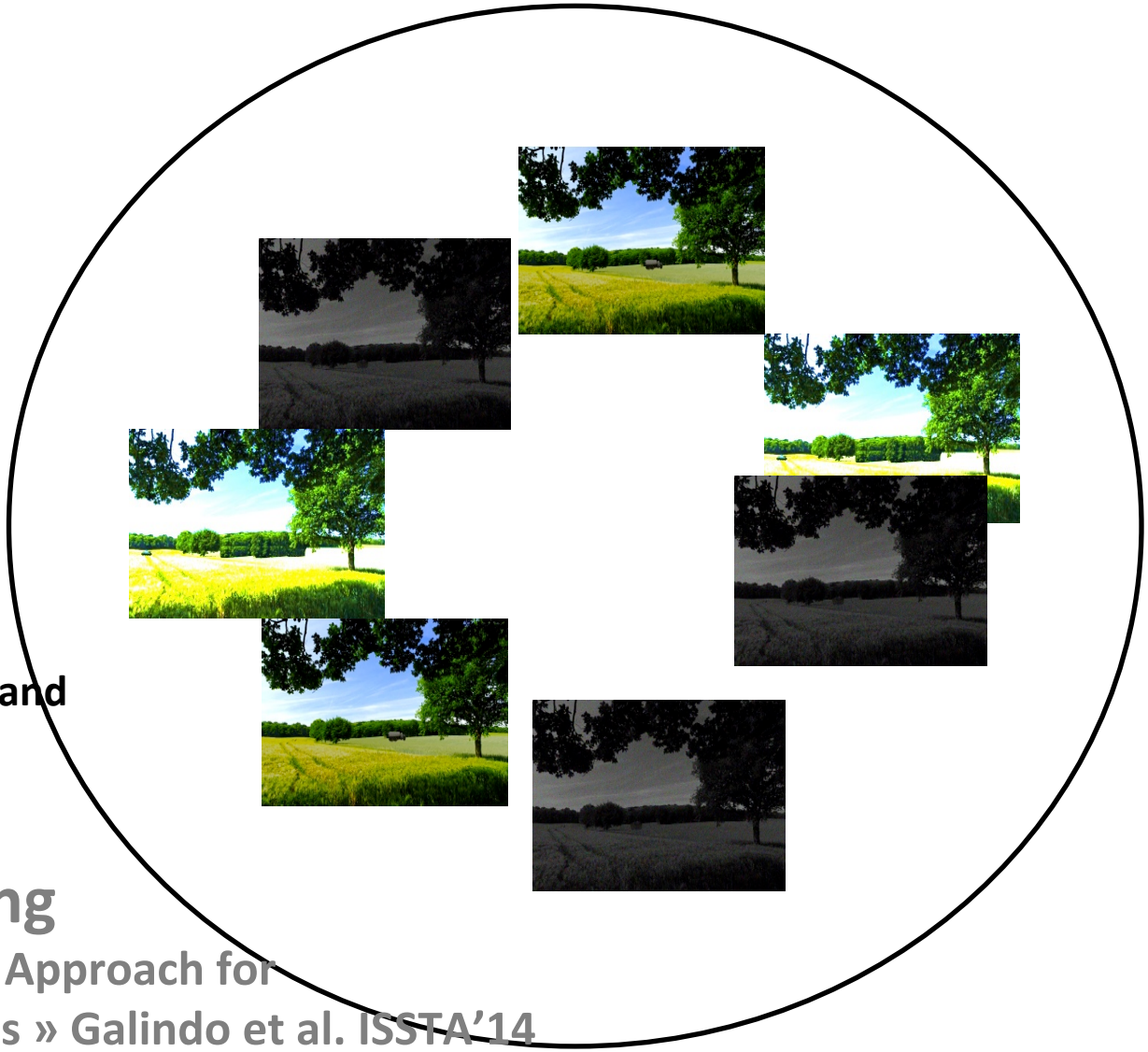
Example

CountryS	normal	0.1	0.2
CountryS	High	0.2	0.4
CountryS	Low	0.3	0.6
Urban	normal	0.3	0.4
Urban	High	0.1	0.2
Urban	Low	0.2	0.6
Desert	normal	0.2	0.6
Desert	High	0.3	0.2
Desert	Low	0.1	0.4
Forest	normal	*	0.4
Forest	High	0.1	0.6
Forest	Low	0.2	0.2



Reasoning operations





Configurations sampling:
*** pair-wise/T-wise**
with respect of constraints and prioritization

Complex CSP encoding

« A Variability-Based Testing Approach for Synthesizing Video Sequences » Galindo et al. ISSTA'14

Effectiveness of pair-wise?

- Mathematical argument: every pair of parameters are in one configuration
- Intuitively: « diversity » of the dataset
  - disimilarities/differences among configurations
- Software engineering argument
  - pair-wise has proved to be an effective sampling criterion for testing programs
  - need more empirical evidence

(RQ2.2) How effective are sampling techniques comparatively?

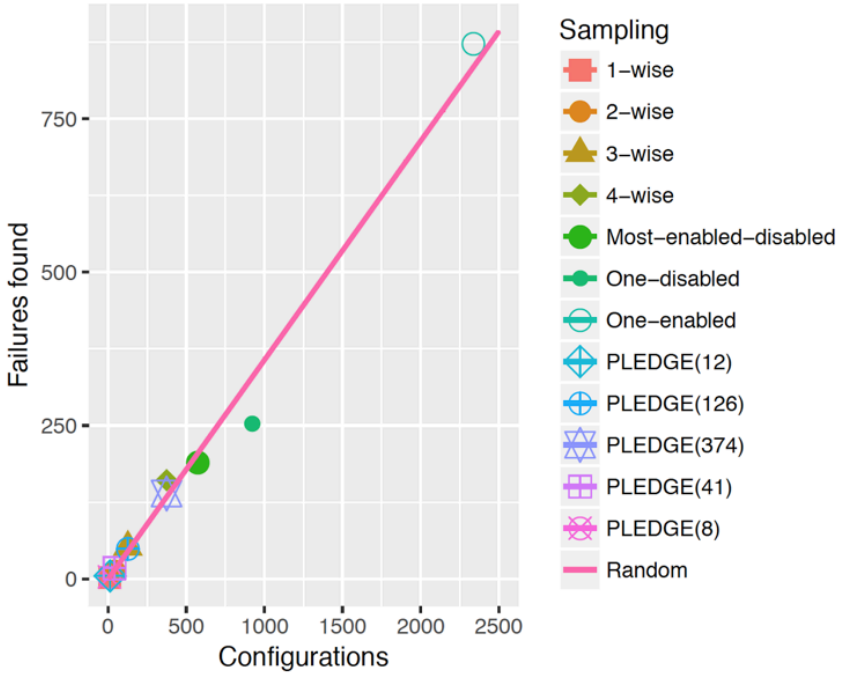
- JHipster team used a sample of 12 configurations
  - Used for continuous integration (Github/Travis CI)
  - **No failure found** with the 12 configs (mid-august/mid-november)
- Automated sampling: 41 configurations are sufficient to cover the 5 most important faults

Sampling technique	Sample size	Failures (σ)	Failures efficiency	Faults (σ)	Fault efficiency
Random(8)	8	2.8568 (1.313296)	35.71%	2.18 (0.9783515)	27.25%
PLEDGE(8)	8	3.16 (1.23028)	39.50%	2.14 (0.82454)	26.75%
1-wise	8	2	25.00%	2	25.00%
Random(12)	12	4.2852 (1.789532)	35.71%	2.7 (1.039619)	22.5%
PLEDGE(12)	12	4.92 (1.23028)	41.00%	2.82 (0.9087)	23.50%
2-wise	41	14	34.15%	5	12.20%
Random(41)	41	14.6411 (3.182322)	35.71%	4.49 (0.7176702)	10.95%
PLEDGE(41)	41	17.64 (2.50407)	43.02%	4.70 (0.83066)	11.46%
3-wise	126	52	41.27%	6	4.76%
Random(126)	126	44.9946 (4.911376)	35.71%	5.28 (0.5333333)	4.19%
PLEDGE(126)	126	49.08 (11.5807)	38.95%	4.66 (0.697664)	3.70%
Random(374)	374	133.5554 (8.406371)	35.71%	5.58 (0.496045)	1.49%
PLEDGE(374)	374	139.20 (31.79748)	37.17%	4.62 (1.18135)	1.24%
4-wise	374	161	43.05%	6	1.60%
Most-enabled-disabled	574	190	33.10%	2	0.35%
One-disabled	922	253	27.44%	5	0.54%
One-enabled	2,340	872	37.26%	6	0.26%
ALL	26,256	9,376	35.71%	6	0.02%

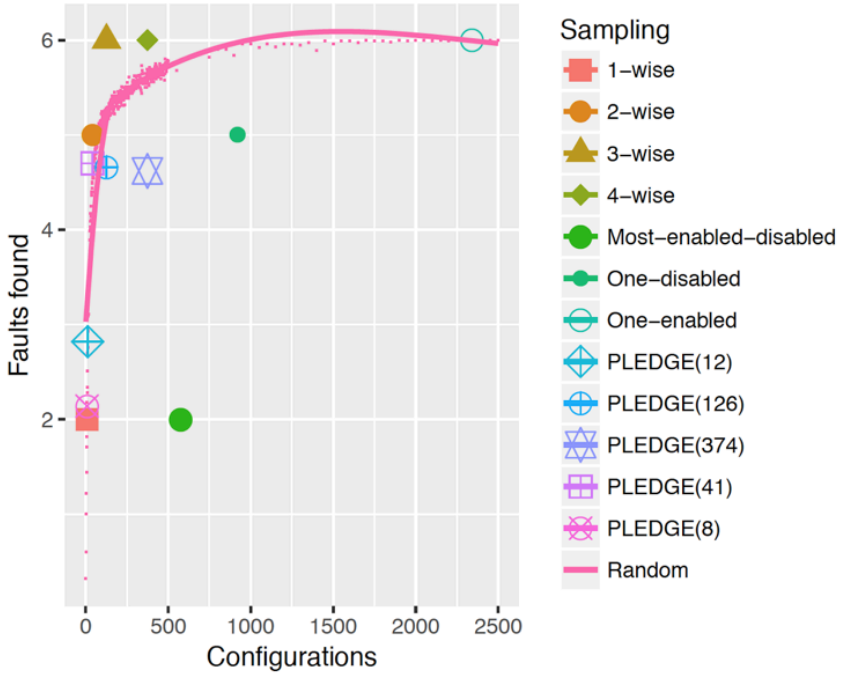
Table 2: Efficiency of different sampling techniques

(RQ2.2) How effective are sampling techniques comparatively?

- Dissimilarity (PLEDGE) and t-wise sampling are the most effective
- Failures != Faults
  - No correlation between failure and fault efficiencies
- Tradeoff between failures and faults



(a) Failures found by sampling techniques



(b) Faults found by sampling techniques

(RQ3) What is the most cost-effective sampling strategy? What are the recommendations for the JHipster project?

- Despite qualitative and quantitative insights (RQ1 and RQ2), we ignore some information:
  - What is the testing budget of JHipster? Why 12 configurations?
  - How do they select this sample?
  - Why do they fail to identify faults?
  - …
- **Practitioners viewpoint**
  - Skype meeting (90'), recorded, w/ Julien Dubois, tech leader of JHipster
  - Mails w/ 2 other core developers (Deepu K Sasidharan, Pascal Grimaud)

(RQ3) What is the most cost-effective sampling strategy? What are the recommendations for the JHipster project?

- **Practitioners viewpoint**
- What is the testing budget of JHipster? Why 12 configurations?
  - *“you can only run 5 concurrent jobs so having more options would take more time to run the CI”*
  - *“If we let each CI build to run for few hours then we would have to wait that long to merge PR and to make releases etc. So turn around IMO is something you need to consider for CI."*
  - *“We only test the 12 combinations because we focus on most popular options and leave the less popular options out.”*
- How do they select this sample?
  - *“intimate technical knowledge of the technologies and a statistical prioritization approach”* (logs)

(RQ3) What is the most cost-effective sampling strategy? What are the **recommendations** for the JHipster project?

- Our results show that eg random sampling is effective
  - *“from a practical standpoint, a random sampling has possibility of us missing an issue in a very popular option thus causing huge impact, forcing us to make emergency releases etc, where as missing issues in a rarely used option does not have that implication”*
  - multi-objective problem: cost, effectiveness, and popularity
- Testing budget (12) clearly too low, need to increase
- Diversify the sample
- Crowdsource the testing effort (contributors/users, etc.)
- **In any case, maintenance of a configuration-aware testing infrastructure needed: deserves dedicated effort (see RQ1)**

Threats to Validity

- How can we generalize our results?
  - single but industrial-strength and complex system
  - expect more insights into characteristics of real-world systems than using diverse but smaller or synthetic benchmarks
  - ground truth that allows us to precisely assess sampling

- Internal validity: quality of our testing infrastructure.
  - An error in the feature model or in the configuration-aware testing workflow can typically produce wrong failures.
  - To mitigate this threat
    - validation of our solution has been a major concern during 8 man/months of development (from statistical computations to manual reviews of individual failures).
    - we do found all faults reported by the JHipster community and new failures.

Conclusion

- We are the first to test all configurations of an industrial-strength, open-source generator (JHipster)
  - Novel insights on engineering difficulties/computational cost
    26K+ configurations, 4376 hours/machine, 8 man/month
  - "Ground truth" allows us to *precisely* assess sampling
    36% failures explained by 6 feature interactions (faults)
- What is the most cost-effective sampling strategy?
  - T-wise or dissimilarity are very effective
  - a new problem: cost, failure/fault effectiveness, popularity, and time at the CI level; hard to not exceed the budget
- Challenges ahead
  - new sampling techniques
  - ways of increasing the testing budget (reusable testing infrastructure)

Is it Worth Testing All Configurations?

- Actionable results
  - researchers interested in building evidenced-based theories or tools for testing configurable systems
  - practitioners in charge of establishing a suitable strategy for testing their systems at each commit or release

Thank you!

https://arxiv.org/abs/1710.07980

Any Questions?

RQ2: Results Bugs

- Authentication related bugs
  - UAA authentication with Docker
    - currently no ready-to-use UAA Docker image / not yet supported
    - known issue : UAA is in “Beta”
  - UAA authentication with Ehcache as Hibernate 2nd level cache / corrected
    - https://github.com/jhipster/generator-jhipster/pull/4225/commits (28 Sep 2016)
  - OAuth2 authentication with SQL database / corrected
    - issue reported but JHipster team can’t reproduce
    - https://github.com/jhipster/generator-jhipster/issues/4009 (20 Aug 2016)
- Social Login
  - Social Login with MongoDB / corrected
    - missing import in _SocialUserConnection.java template
    - https://github.com/jhipster/generator-jhipster/pull/4037 (27 Aug 2016)

RQ2: Results Bugs

- Database related bugs
  - MariaDB with Docker
    - missing line in .yml file / corrected
    - https://github.com/jhipster/generator-jhipster/commit/9f320bc86ab6209585e24b4db8abecd3ecf29ba2 (24 Nov 2016)
  - MariaDB using Gradle
    - missing support in gradle template / corrected
    - https://github.com/jhipster/generator-jhipster/pull/4222 (27 Sep 2016)
  - SQL with Docker
    - initialization of the database failed (2064 variants) / not corrected
    - JHipster team unable to reproduce