Validation of Constraints Among Configuration Parameters Using (Search-Based) Combinatorial Interaction Testing

Angelo Gargantini, Justyna Petke, Marco Radavelli, Paolo Vavassori

University of Bergamo, Bergamo, Italy

University College London, London, UK

NIST, August 31, 2016

Paper to be presented at SSBSE 2016

Few words about my city

- Bergamo – near to Milan (31 mi, 50 km)
- Around 120k habitants
- Rich in history and art

Venetian walls, candidate for UNESCO site

University of Bergamo

- Public university
- Rather young (1968)
- Schools:
  - Arts and Philosophy,
  - Economics and Business Administration,
  - Engineering,
  - Foreign Languages, Literature and Communication,
  - Law,
  - Human and Social Sciences
- Around 15k students

Motivations

- Most software systems can be configured in order to improve their capability to address user's needs.
- Configuration of such systems is performed by parameters:
  - software design stage (e.g., for software product lines, the designer identifies the features unique to individual products and features common to all products in its category),
  - during compilation (e.g., to improve the efficiency of the compiled code)
  - while the software is running (e.g., to allow the user to switch on/off a particular functionality).
  - during load time, to decide which features to load.

Role of constraints among feature

- Constraints among features play a very important role,
- They identify parameter interactions that lead to invalid configurations
  - Normally invalid configurations need not be tested,
    - hence constraints can significantly reduce the testing effort.
  - Certain constraints are defined to prohibit generation of test configurations under which the system simply should not be able to run.
- Other constraints can prohibit system configurations that could be valid, but need not be tested for other reasons.
  - For example business constraints
- Identifying features is much easier than finding their relationships

Importance of validating constraints

- Constructing a CIT model of a large software system is a hard, usually manual task.
- Modeling constraints among parameters is highly error prone.
- One might run into the problem of not only producing an incomplete CIT model, but also one that is over-constrained.
  - Even if the CIT model only allows for valid configurations to be generated, it might miss important system faults if one of the constraints is over-restrictive.
  - Moreover, even if the system is not supposed to run under certain configurations, if there's a fault, a test suite generated from a CIT model that correctly mimics only desired system behavior will not find that error.

Our former work on constraint validation

- *Validation of Models and Tests for Constrained Combinatorial Interaction Testing*, IWCT2014
- We used a SMT solver to fault faults in the constraints
- We focused on
  - (meta)-errors: regardless the system they model
    - Inconsistent constraints
    - Constraints Vacuity
    - Constraints minimality
- Here we focus on conformance faults
- As before we use CitLab (https://citlab.sf.net)

Main GOAL

- The objective of this work is to use CIT techniques to validate constraints of the model of the system under test (SUT).
- We extend traditional CIT by devising a set of six policies for generating tests that can be used to detect faults in the CIT model as well as the SUT.

Example 1

- Compile time configurable example:

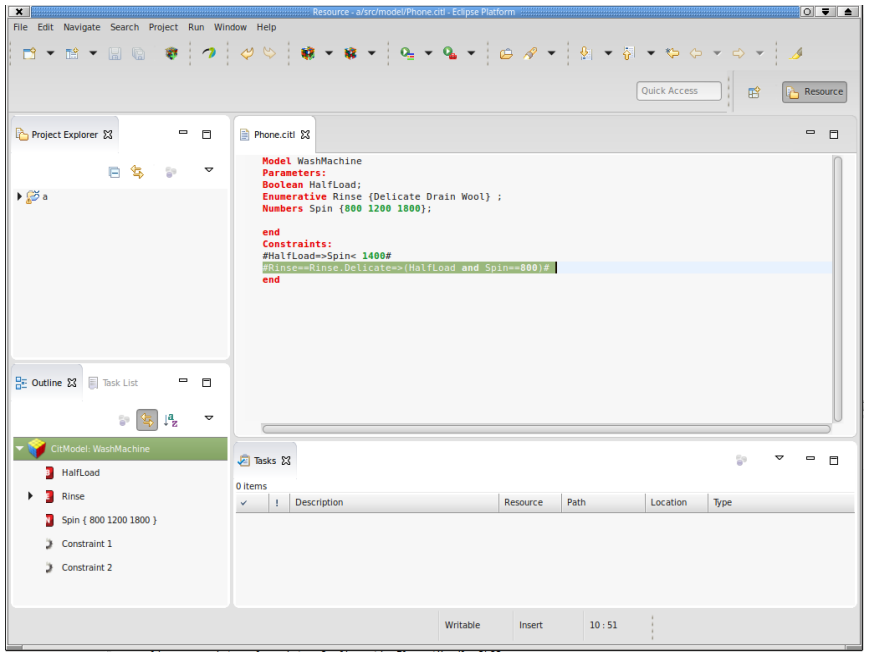
Real software, greetings.c

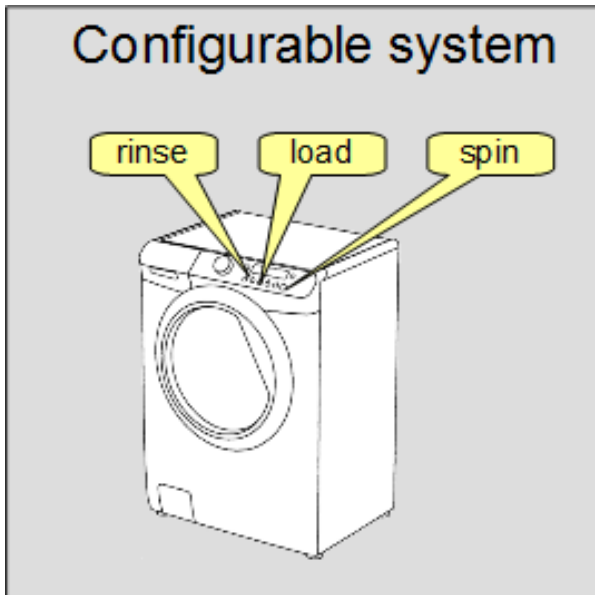
```
#ifdef HELLO
char* msg = "Hello!\n";
#endif

#ifdef BYE
char* msg = "Bye bye!\n";
#endif

void main(void) {
    printf(msg);
}
```

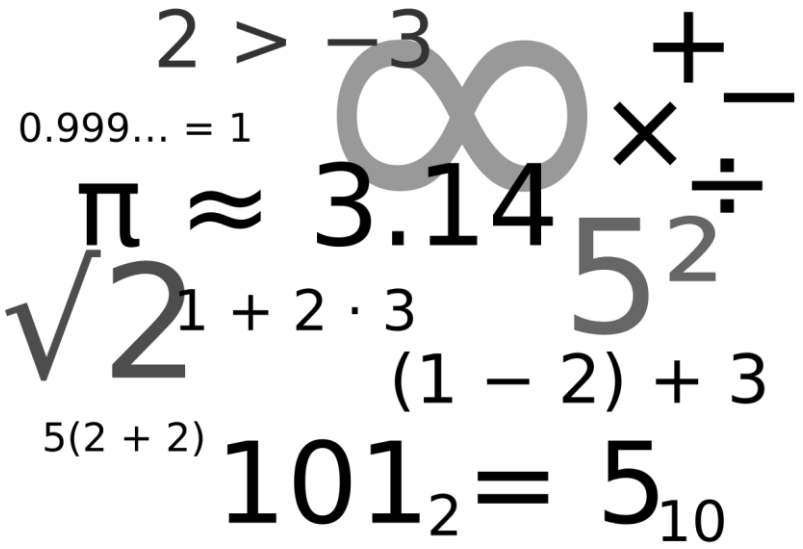
CIT model, greetings.citl

```
Model Greetings
Parameters:
Boolean HELLO;
Boolean BYE;
end
Constraints:
   # HELLO!=BYE #
end
```

(2) It can be a phisical system







Some definitions

Oracle and configuration validity

- **Configuration**: assignment to parameters
- Given a model S and its implementation I,
  - val_S is the function that checks if a configuration satisfies the constraints in S,
    - val_S (p) TRUE if p makes the constraints in S true
  - $orcale_I$ checks if a configuration is valid for the implementation I
    - $orcale_I$(p) is TRUE iff p is a valid configuration for I
- Computation of oracle can be expensive
  - Some human intervention

Correctness and faults

- We say that the Constrained CIT (CCIT) model is correct if, for every configuration p, $\mathbf{val_S(p) = orcale_I(p)}$

- We say that a specification contains a conformance fault if there exists a $\bar{p}$ such that $\mathbf{val_S(\bar{p}) \neq orcale_I(\bar{p})}$

Example

```
#ifdef HELLO
char* msg = "Hello!\n";
#endif

#ifdef BYE
char* msg = "Bye bye!\n";
#endif

void main(void) {
    printf(msg);
}
```

```
Model Greetings
Parameters:
Boolean HELLO;
Boolean BYE;
end
Constraints:
   # HELLO!=BYE #
end
```

FAULT

```
Model Greetings
Parameters:
Boolean HELLO;
Boolean BYE;
end
Constraints:
   # HELLO or BYE #
end
```

A possible configuration

```
HELLO: true
BYE: true
```

$orcale_I$

val_S

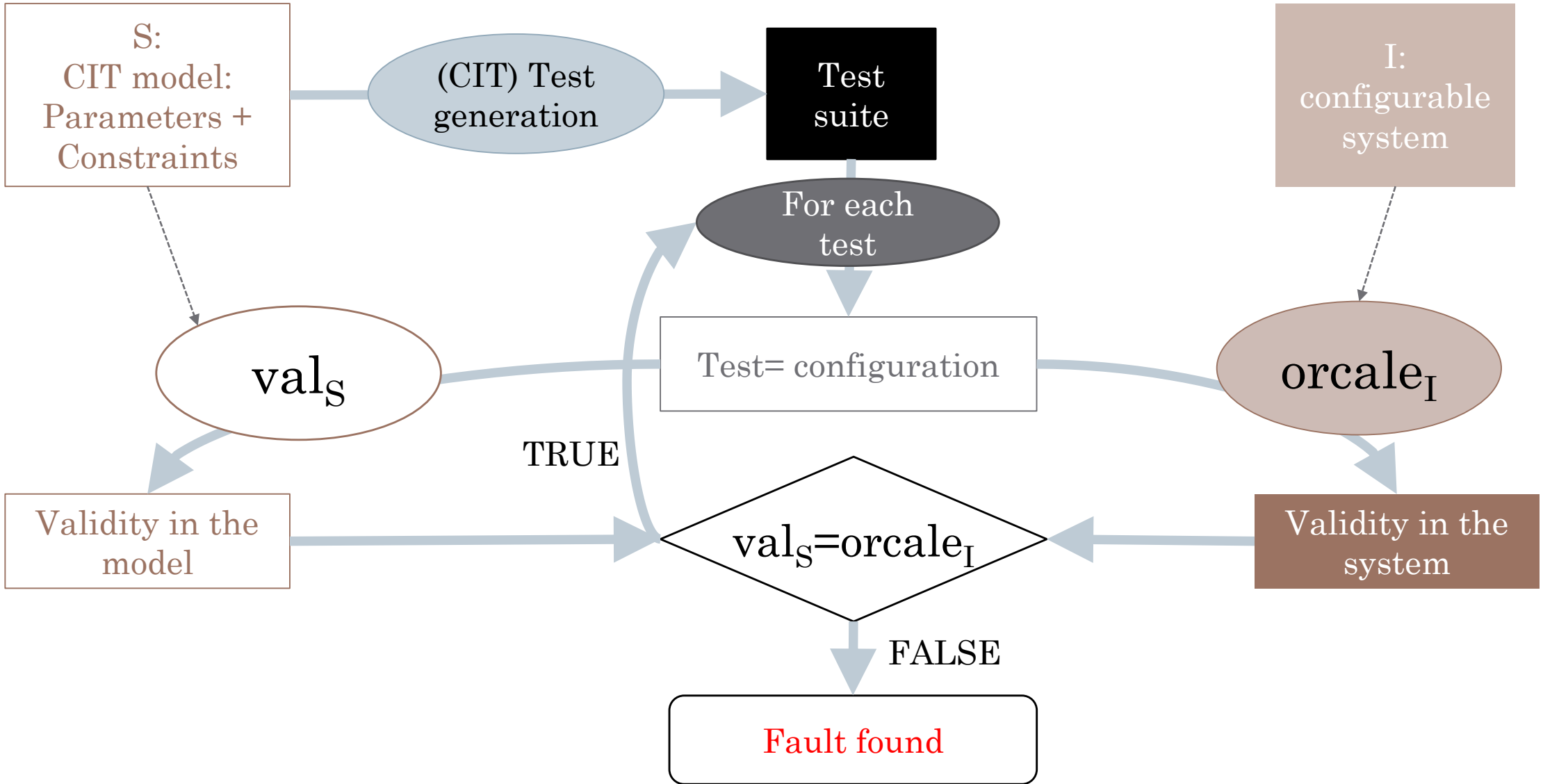
Oracle: the compiler

gcc –DBYE –DHELLO greetings.c → compilation error

FALSE

TRUE

Discovering fault process



Discovering faults

- In order to find possible faults the exhaustive exploration of all the configurations of a large software system is usually impractical.
  - Some static techniques can be adopted
    - TypeChef, …
  - In this work we use **combinatorial testing**
- We include both **valid** and **invalid** configurations
  - Both selected with the same t-way interaction testing

Invalid Configuration Testing

- In classical CIT only valid tests are generated,
  - the focus is on assessing if the system under test produces valid outputs.
- We believe that invalid tests are also useful:

1. The model should minimize the constraints and the invalid configuration set:
   - invalid configurations, according to the model, should only be those that are actually invalid in the real system.
   - Avoid over-constraining the model.
2. Moreover, critical systems should be tested if they safely fail when the configuration is incorrect.
3. Invalid configurations generated by the model at hand can help reveal constraints within the system under test and help refine the CIT model.

- scientific epistemology: not only tests (i.e., valid configurations) that confirm our theory (i.e., the model), but also tests that can refute it.

Combinatorial Testing Policies

Washing Machine eample

```
Model WashingMachine
Parameters:
  Boolean HalfLoad;
  Enumerative Rinse { Delicate Drain Wool };
  Numbers Spin { 800 1200 1800 };
end
Constraints:
# HalfLoad => Spin < 1400 #
# Rinse == Delicate => ( HalfLoad and Spin==800) #
end
```

UC: Unconstrained CIT

- Ignore the constraints
  - Tools that do not handle constraints can be used
  - Both valid and invalid tests will be generated but there is no control

```
Parameters:
  Boolean HalfLoad;
  Enumerative Rinse { Delicate Drain Wool };
  Numbers Spin { 800 1200 1800 };
end
Constraints:
# HalfLoad => Spin < 1400 #
# Rinse==Rinse.Delicate => ( HalfLoad and Spin==800) #
end
```

a pairwise test suite with at least 9 test cases, including an invalid test case where **HalfLoad** = true in combination with **Spin** =**1800**.

CC: Constrained CIT

- Classical approach, constraints are taken into account and only valid combinations among parameters are chosen.
  - If a certain interaction among parameters is not possible, then it is not considered

```
Model WashingMachine
Parameters:
  Boolean HalfLoad;
  Enumerative Rinse { Delicate Drain Wool };
  Numbers Spin { 800 1200 1800 };
end
Constraints:
# HalfLoad => Spin < 1400 #
# Rinse==Delicate => ( HalfLoad and Spin==800) #
end
```

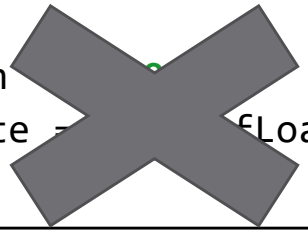
7 tests for pairwise, all of which satisfy the constraints.

Some pairs are not covered: for instance `HalfLoad` =true and **`Spin==1800`** will not be covered.

CV: Constraints Violating CIT

- one wants to test the interactions of parameters that produce errors, → tests violating the constraints
  - complementary with respect to the CC

```
Constraints:
# HalfLoad => Spin
# Rinse == Delicate
end
```

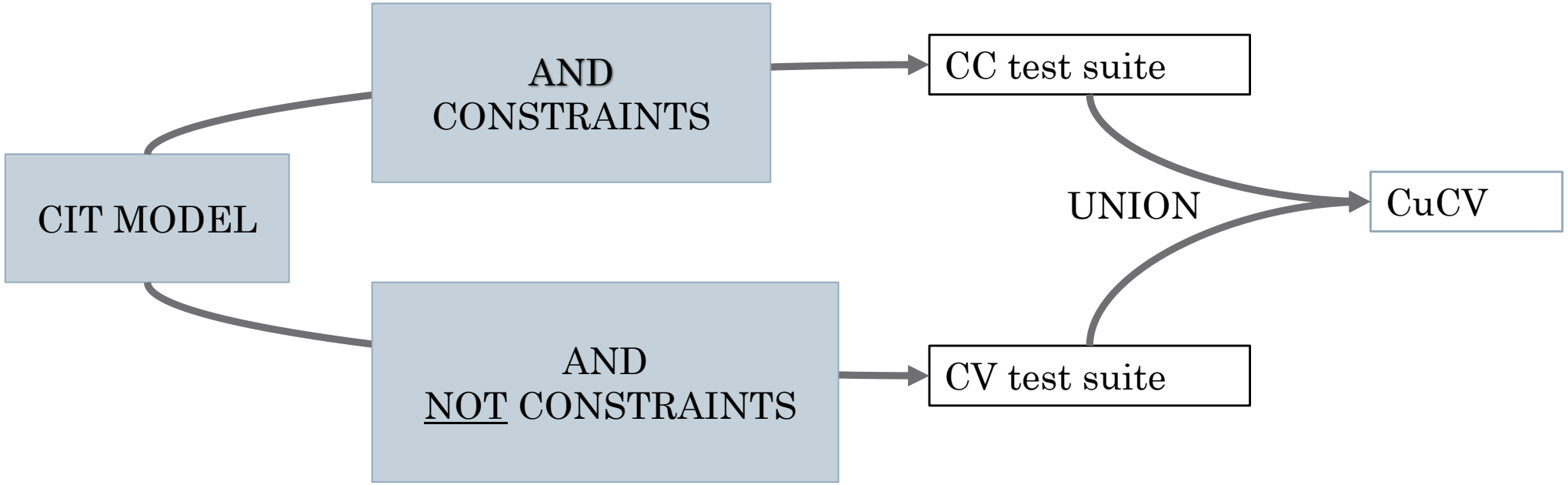


```
Constraints:
# ! ( ( HalfLoad => Spin<maxSpinHL) &&
( Rinse == Delicate => HalfLoad and Spin==800)) #
end
```

6 test cases, all of which violate some constraint of the model.
For instance, a test has `Rinse = Delicate`, **`Spin=800`**, and `HalfLoad` =false.

CuCV: Combinatorial Union

- CC: only valid, CV: only invalid
  - Both may do not cover some requirements (pairs in pairwise)
- Take the union



ValC: CIT of Constraint Validity

- CuCV may produce big test suites:
  - it covers all the desired parameter interactions that produce valid configurations and all those that produce invalid ones
- We propose the ValC policy: requires the interaction of each parameter with the validity of the whole CIT model.
  - Good balance e test on how the single parameters contributes to the validity

```
Parameters:
…
Boolean validity;
…
Constraints:
# validity <=>
( HalfLoad => Spin<maxSpinHL) &&
 ( Rinse== Delicate => HalfLoad and Spin==800) #
end
```

CuCV generates 13 test cases (6+7).
ValC requires only 11 test cases.

CCi: CIT of the Constraints

- Every constraint represents a condition over the system
  - the constraint HalfLoad => Spin < 1800 identifies the critical states in which the designer wants a lower spin speed.
  - Constraint ➔ system state property
- CCi covers how each of these properties interact with each other and with the other parameters.

```
Parameters:
…
  Boolean c0;
  Boolean c1;
end
Constraints:
# c0 <=> ( HalfLoad => Spin<maxSpinHL) #
# c1 <=> ( Rinse==Rinse.Delicate =>
                HalfLoad and Spin==800) #
end
```

CCi generates 11 test cases.

Experiments

Using CASA

with ACTS: bigger test suites but no out of memory problems

Benchmarks

- Banking1 represents the testing problem for a configurable Banking applicationby Itai Segall and Rachel Tzoref-Brill group
- libssh is a multi-platform library implementing SSHv1 and SSHv2 written in C. The library consists of around 100 KLOC and can be configured by several options and several modules (like an SFTP server and so on) can be activated during compile time. We have analyzed the cmake files and identified 16 parameters and the relations among them.

HeartbeatChecker (HC)

- HC is a small C program, written by us, that performs a Heartbeat test on a given TLS server.
  - The Heartbeat Extension is a standard procedure (RFC 6520) that sends a “Heartbeat Request” message.
  - Such a message consists of a payload, a text string, along with its length as a 16-bit integer.
  - The receiving computer then must send exactly the same payload back to the sender.
- HC reads the data to use in the Heartbeat from a configuration file:

```
TLSserver : <IP>
TLS1_REQUEST Length : <n1> PayloadData : <data1>
TLS1_RESPONSE Length : <n2> PayloadData : <data2>
```

Messages with n1 equal to n2 and data1 equal to data2 represent a successful Heartbeat test (when the TLS-server has correctly responded to the request).

HC can be considered as an example of a runtime configurable system,

The oracle is true if the Heartbeat test has been successfully performed with the specified parameters.

- is a free and open source web application framework
- Each django project has a configuration file, which is loaded every time the web server that executes the project (e.g. Apache) is started.
- We modeled the configuration file:
  - one Enumerative and 23 Boolean parameters.
- We elicited the constraints from the documentation, including several forum articles and from the code when necessary.
- We have also implemented the oracle, which is completely automated and returns true if and only if the HTTP response code of the project homepage is 200 (HTTP OK).

Benchmarks features

	#VARS	#Constraints	#configurations	VR	#pairs
Banking1	5	112	324	65.43%	102
Libssh	16	2	65536	50%	480
HeartBeat Checker	4	3	65536	0.02%	1536
Django	24	3	33554432	18.75%	1196

VR - validity ratio: how many configurations are those valid

#pairs: testing requirements how may pairs are to be covered

Results (global)

	Banking1				Django				LibSSH				HeartBeatChecker			
Pol.	time	size	#Val	Cov.	time	size	#Val	Cov.	time	size	#Val	Cov.	time	size	#Val	Cov.
UC	0,22	12	11	100%	0,65	10	2	100%	0,25	8	4	100%	447	267	0	100%
CC	0,26	13	13	100%	1,24	10	10	91.8%	0,28	8	8	99.3%	2,74	141	141	6.2%
CV	Out of memory				0,32	11	0	100%	0,25	8	0	99.3%	Out of memory			
CuCV	Out of memory				1,58	21	10	100%	0,52	16	8	100%	Out of memory			
ValC	Out of memory				0,31	11	4	100%	0,29	8	5	100%	Out of memory			
CCi	6,22	12	9	100%	0,58	13	3	100%	0,3	8	2	100%	460	268	0	100%

Time: generation (ocracle excluded) in seconds.
Size: number of tests and how many of those are valid (#Val),
Cov.: The percentage of parameter interactions (pairs) that are covered.
Out of memory is due to constraint conversion into the CNF format required by CASA.

UC: invalid test

	Banking1			Django			LibSSH			HeartBeatChecker		
Pol.	size	#Val	Cov.	size	#Val	Cov.	size	#Val	Cov.	size	#Val	Cov.
UC	12	11	100%	10	2	100 %	8	4	100 %	267	0	100%

- UC produces both a mix of valid and invalid tests.
  - There is no control though.
  - It may produce all invalid tests (especially if the constraints are strong - see HeartbeatChecker).
- Having all invalid tests may reduce test effectiveness.

CC: coverage and test suite size

	Banking1			Django			LibSSH			HeartBeatChecker		
Pol.	size	#Val	Cov.	size	#Val	Cov.	size	#Val	Cov.	size	#Val	Cov.
UC	12	11	100%	10	2	100%	8	4	100%	267	0	100%
CC	13	13	100%	10	10	91.8%	8	8	99.3%	141	141	6.2%

- CC usually does not cover all the parameter interactions, since some of them are infeasible because they violate constraints
- On the other hand, CC generally produces smaller test suites (as in the case of HeartbeatChecker). However, in some cases, CC is able to cover all the required tuples at the expense of larger test suites (as in the case of Banking1).

CV: coverage and resources

	Banking1				Django			LibSSH			HeartBeatChecker
Pol.					size	#Val	Cov.	size	#Val	Cov.	
CV	Out of memory				11	0	100%	8	0	99.3%	Out of memory

- CV generally does not cover all the parameter interactions, since it produces only invalid configurations.
- However, in one case (Django) CV covers all the interactions.
  - This means that 100% coverage of the tuples in some cases can be obtained with no valid configuration generated: coverage and validity are not strongly correlated
- Sometimes CV is too expensive to perform.

CuCV

	Banking1		Django				LibSSH				HeartBeatChecker	
Pol.			time	size	#Val	Cov.	time	size	#Val	Cov.		
CC			1,24	10	10	91.8%	0,28	8	8	99.3%		
CV	Out of memory		0,32	11	0	100%	0,25	8	0	99.3%	Out of memory	
CuCV	Out of memory		1,58	21	10	100%	0,52	16	8	100%	Out of memory	

- CuCV guarantees to cover all the interactions and it produces both valid and invalid configurations.
- However, it produces the bigger test suites
- and it may fail because it relies on CV
  - With ACTS this was not the case !

ValC: faster and smaller

	Banking1		Django				LibSSH				HeartBeatChecker	
Pol.			time	size	#Val	Cov.	time	size	#Val	Cov.		
CuCV	Out of memory		1,58	21	10	100%	0,52	16	8	100%	Out of memory	
ValC	Out of memory		0,31	11	4	100%	0,29	8	5	100%	Out of memory	

- ValC covers all the interactions with both valid and invalid configurations.
- It produces test suites smaller than CuCV and it is generally faster, but as CuCV may not terminate.

CCi

	Banking1				Django				LibSSH				HeartBeatChecker			
Pol.	time	size	#Val	Cov.	time	size	#Val	Cov.	time	size	#Val	Cov.	time	size	#Val	Cov.
CCi	6,22	12	9	100%	0,58	13	3	100%	0,3	8	2	100%	460	268	0	100%

- CCi covers all the interactions,
- it generally produces both valid and invalid test.
  - However, it may produce all invalid tests (see HeartbeatChecker), and
- it produces a test suite comparable in size with UC.
  - however, it guarantees an interaction among the constraint validity.
- It terminates, but it can be slightly more expensive than UC and CC.

Comparison

Pol.	Valid and invalid tests	Test suite size	Time/memory requirements	Coverage
UC	No guarantee	Depends	good	100%
CC	NO	Depends	Best (CASA)	Can be low
CV	NO	Depends	Out of memory	Can be low
CuCV	YES	BIGGEST	Out of memory	100%
ValC	YES	Depends	Out of memory	100%
CCi	No Guarantee	Depends	good	100%

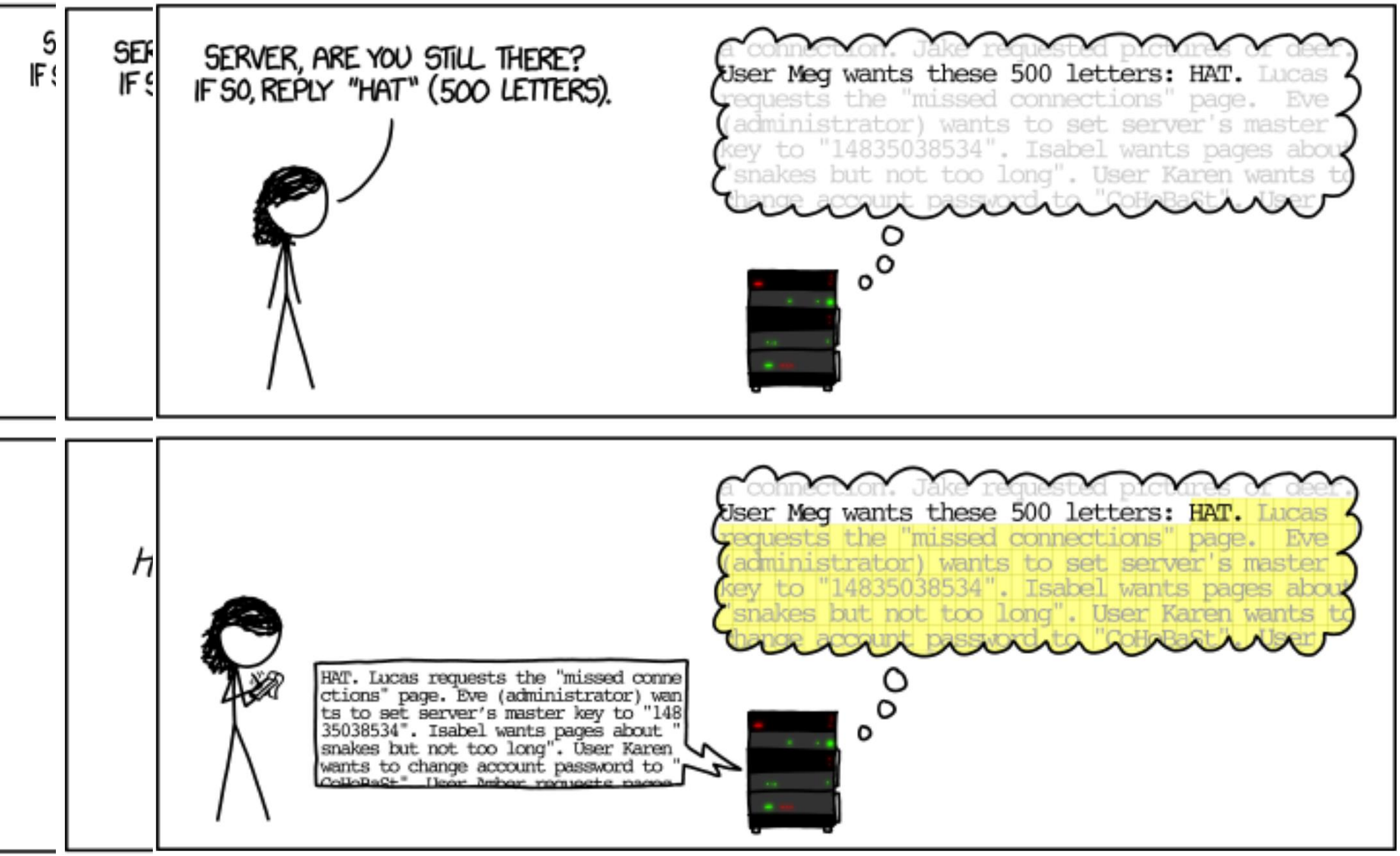
Fault detection capability

- Ok coverage but what about fault detection capability?

- We have applied mutation analysis
- We have introduced (by hand) artificial faults and checking if the proposed technique is able to find (kill) them.
- Our technique is able to find conformance faults both in the model and in the implementation, so we have modified both the specification (S) and the implementation (I)

Faults

		Type of fault	
LibSSH	L1	forgot all the constraints	S
	L2	remove a constraint	S
	L3	add a constraint	S
	L5	remove a dependency	I
	L6	add a dependency	I
HeartBeatChecker	H1	remove one constraint	S
	H2	== to <=	S
	H4	&& to ||	
	H5	== to != (all)	I
	H6	== to != (one)	I
	H7	HeartBleed	I


SERVER, ARE YOU STILL THERE? IF SO, REPLY "HAT" (500 LETTERS).
User Meg wants these 500 letters: HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHeRaSt". User
HAT. Lucas requests the "missed connections" page. Eve (administrator) wants to set server's master key to "14835038534". Isabel wants pages about "snakes but not too long". User Karen wants to change account password to "CoHeRaSt". User

Fault detection capability

Policy	L1	L2	L3	L4	L5	L6	H1	H2	H3	H4	H5	H6	H7	score
UC	✓	✓		✓	✓		✓	✓	✓	✓	✓	✓		10/13
CC	✓	✓				✓	✓	✓	✓		✓			7/13
CV	✓		✓	✓	✓		?	?	?	?	?	?	?	4/13
CuCV	✓	✓	✓	✓	✓	✓	?	?	?	?	?	?	?	6/13
ValC	✓	✓		✓	✓		?	?	?	?	?	?	?	4/13
CCi	✓	✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓	12/13

- Testing invalid configurations is useful
- Overall CCi was the best in terms of fault detection,
  - it missed one of the injected faults (L6).
  - It was the only one to find the fault H7 (HeartBleed).
- This proves that testing how parameters can interact with single constraints increases the fault detection capability

Conclusions

- CIT can be applied to test configurable systems
  - Constraints are important but
  - also invalid configurations should be generated
- There are several way to consider constraints
  - New 4 policies
  - Some proved to be more powerful in coverage and fault detection