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

Angelo Gargantini[1], Justyna Petke[2], Marco Radavelli[1], and Paolo Vavassori[1]

[1] University of Bergamo, Bergamo, Italy
{angelo.gargantini,marco.radavelli,paolo.vavassori}@unibg.it
[2] University College London, London, UK j.petke@ucl.ac.uk

Abstract. The appeal of highly-configurable software systems lies in their adaptability to users' needs. Search-based Combinatorial Interaction Testing (CIT) techniques have been specifically developed to drive the systematic testing of such highly-configurable systems. In order to apply these, it is paramount to devise a model of parameter configurations which conforms to the software implementation. This is a non-trivial task. Therefore, we extend traditional search-based CIT by devising 4 new testing policies able to check if the model correctly identifies constraints among the various software parameters. Our experiments show that one of our new policies is able to detect faults both in the model and the software implementation that are missed by the standard approaches.

Keywords: Combinatorial testing, feature models, configurable systems, CIT

1 Introduction

Most software systems can be configured in order to improve their capability to address user's needs. Configuration of such systems is generally performed by setting certain parameters. These options, or *features*, can be created at the 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) or while the software is running (e.g., to allow the user to switch on/off a particular functionality). A configuration file can also be used to decide which features to load at startup.

Large configurable systems and software product lines can have hundreds of features. It is infeasible in practice to test all the possible configurations. Consider, for example, a system with only 20 Boolean parameters. One would have to check over one million configurations in order to test them all (2^{20} to be exact). Furthermore, the time cost of running one test could range from fraction of a second to hours if not days. In order to address this combinatorial explosion problem, Combinatorial Interaction Testing (CIT) has been proposed for testing configurable systems [4]. It is a very popular black-box testing technique that

tests all interactions between any set of t parameters. There have been several studies showing the successful efficacy and efficiency of the approach [13, 14, 21].

Furthermore, certain tests could prove to be infeasible to run, because the system being modelled can prohibit certain interactions between parameters. Designers, developers, and testers can greatly benefit from modelling parameters and constraints among them by significantly reducing modelling and testing effort [21] as well as identifying corner cases of the system under test. Constraints play a very important role, since they identify parameter interactions that need not be tested, hence they 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 are valid, but need not be tested for other reasons. For example, there's no point in testing the *find* program on an empty file by supplying all possible strings.

Constructing a CIT model of a large software system is a hard, usually manual task. Therefore, discovering 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 behaviour will not find that error. In such situations tests that exercise those corner cases are desirable.

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.

2 Combinatorial Models of Configurable Systems

Combinatorial Interaction Testing (CIT), or simply combinatorial testing, aims to test the software or the system with selected combinations of parameter values. There exist several tools and techniques for CIT. Good surveys of ongoing research in CIT can be found in [9, 19], while an introduction to CIT and its efficacy in practice can be found in [15, 21].

A model for a combinatorial problem consists of several parameters which can take several domain values. In most configurable systems, dependencies exist between parameters. Such constraints may be introduced for several reasons, e.g., to model inconsistencies between certain hardware components, limitations of the possible system configurations, or simply design choices [4]. In our approach, tests that do not satisfy the constraints in the CIT model are considered *invalid*.

We assume that the models are specified using CitLab [7, 3]. This is a framework for combinatorial testing which provides a rich abstract language with precise formal semantics for specifying combinatorial problems, and an eclipse-based

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

(a) Washing Machine example

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

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

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

(b) Compile time configurable example, its CIT model (left) and the source code (right)

Fig. 1. Combinatorial interaction CitLab models

editor with a rich set of features. CitLab does not have its own test generators, but it can utilise, for example, the search-based combinatorial test generator CASA[3][8]. CIT problems can be formally defined as follows.

Definition 1 *Let $P = \{p_1, \ldots, p_m\}$ be the set of parameters. Every parameter p_i assumes values in the domain $D_i = \{v_1^i, \ldots, v_{o_i}^i\}$. Every parameter has its name (it can have also a type with its own name) and every enumerative value has an explicit name. We denote with $C = \{c_1, \ldots, c_n\}$ the set of constraints.*

Definition 2 *The objective of a CIT test suite is to cover all parameter interactions between any set of t parameters. t is called the strength of the CIT test suite. For example, a pairwise test suite covers all combinations of values between any 2 parameters.*

Constraints c_i are given in general form, using the language of propositional logic with equality and arithmetic. Fig. 1a shows the CitLab model of a simple washing machine consisting of 3 parameters. The user can select if the machine has `HalfLoad`, the desired `Rinse`, and the `Spin` cycle speed. There are two constraints, including, if `HalfLoad` is set then the speed of spin cycle cannot exceed `maxSpinHL`.

Software systems can be configured by setting specific parameter values at different stages of the software testing process.

Compile time Configurations can be set at compile time. An example is shown in Figure 1b. Depending on the value settings of the Boolean variables `HELLO` and `BYE` different messages will be displayed when the program is run.

Design time Configurations can also be set at design time. For example, in case of a SPL, a configurability model is built during the design.

Runtime Another way of setting parameter configurations is at runtime. This can be usually done by means of a graphical user interface (GUI). In a chat client, e.g., you can change your availability status as the program is running.

[3] `http://cse.unl.edu/~citportal/`

Launch time We also differentiate the case where parameters are read from a separate configuration file or given as program arguments, *before* the system is run. We say that these parameters are set at launch time of the given application. They decide which features of the system should be activated at startup. Examples of such systems include chat clients, web browsers and others.

3 Basic Definitions

We assume that the combinatorial model represents the specification of the parameters and their constraints for a real system as it has been implemented. We are interested in checking whether this system specification correctly represents the software implementation. We assume that the parameters and their domains are correctly captured in the specification, while the constraints may contain some faults. Specification S belongs to the problem space while software implementation I belongs to the solution space [18].

Formally, given an assignment $\bar{p}$ that assigns a value to every parameter in P of the model S, we introduce two functions:

Definition 3 *Given a model S and its implementation I, val_S is the function that checks if assignment $\bar{p}$ satisfies the constraints in S, while $oracle_I(\bar{p})$ checks if $\bar{p}$ is a valid configuration according to implementation I.*

We assume that the oracle function $oracle_I$ exists. For instance, in case of a compile-time configurable system, we can assume that the compiler plays the role of an oracle: if and only if the parameters $\bar{p}$ allow the compilation of the product then we say that $oracle(\bar{p})$ holds. We may enhance the definition of oracle by considering also other factors, for example, if the execution of the test suite completes successfully. However, executing $oracle_I$ may be very time consuming and it may require, in some cases, human intervention.

On the model side, the evaluation of $val_S(P)$ is straightforward, that is, $val_S(\bar{p}) = c_{1[P \leftarrow \bar{p}]} \wedge \ldots \wedge c_{n[P \leftarrow \bar{p}]}$.

Definition 4 *We say that the Constrained CIT (CCIT) model is correct if, for every p, $val_S(p) = oracle_I(p)$. We say that a specification contains a* conformance *fault if there exists a $\bar{p}$ such that $val_S(\bar{p}) \neq oracle_I(\bar{p})$.*

3.1 Finding Faults by Combinatorial Testing

In order to find possible faults as defined in Definition 4, the exhaustive exploration of all the configurations of a large software system is usually impractical. In many cases, the evaluation of $oracle_I$ is time consuming and error prone, so the number of tests one can check on the implementation can be very limited. Instead, we can apply combinatorial testing in order to select the parameters values and check that for every generated CIT test $val_S(p) = oracle_I(p)$ holds. This approach does not guarantee, of course, finding all possible conformance

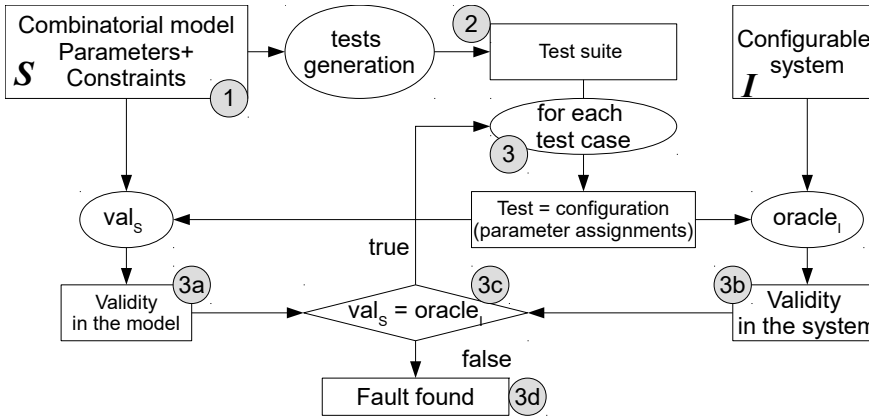

Fig. 2. Validating constraints by CIT

faults, but we can assume that faults are due to the interaction of parameters and we can leverage the success of CIT in finding faults in real configurable systems [14, 21].

We have devised a process that is able to find possible conformance faults. It is depicted in Figure 2 and consists of the following steps:

1. Create a CIT model S that takes constraints into account.
2. Generate a CIT test suite according to one of the policies (see Section 4).
3. For every test in the test suite,
   (a) Compute its validity as specified by the constraints in the CIT model.
   (b) Compute $oracle_I$, by executing the software system under each configuration to check if it's acceptable.
   (c) Compare the validity, as defined by the model, with the actual result.
   (d) If $val_S \neq oracle_I$ a fault (either in the model or in the system) is found.

A discrepancy between the model and the real system means that a configuration is correct according to the model but rejected by the real system (or the other way around) and this means that the constraints in the model do not correctly describe constraints in the system under test.

Invalid Configuration Testing. In classical combinatorial interaction testing, only valid tests are generated, since the focus is on assessing if the system under test produces valid outputs. However, we believe that invalid tests are also useful. In particular, they address the following issues.

The CIT model should minimise the number of constraints and the invalid configuration set: invalid configurations, according to the model, should only be those that are actually invalid in the real system. This kind of test aims at discovering faults of over-constraining the model. This problem is a variant of the bigger problem of over-specification. Moreover, critical systems should be

tested if they safely fail when the configuration is incorrect. This means that the system should check that the parameters are not acceptable (i.e. it must *fail*) and it should fail in a safe way, avoiding crashes and unrecoverable errors (it must fail *safely*). Furthermore, creation of a CIT model for a large real-world software system is usually a tedious, error-prone task. Therefore, invalid configurations generated by the model at hand can help reveal constraints within the system under test and help refine the CIT model. In line with the scientific epistemology, our research focuses on generating not only tests (i.e., valid configurations) that confirm our theory (i.e., the model), but also tests that can *refute* or *falsify* it. Since the number of invalid configurations might be huge, such configurations must be chosen in accordance with some criteria. We choose to use the same t-way interaction paradigm as in standard CIT.

4 Combinatorial Testing Policies

We propose to use search-based combinatorial interaction testing techniques to verify the validity of CIT models. In particular, given a CIT model, we modify it according to one of the policies introduced in this section. Next, we use CASA to generate the test suite satisfying the modified CIT model. We use the term "valid test" to denote the generated configuration that satisfies all the constraints of the original CIT model. Conversely, the term "invalid test" is used for a configuration that does not satisfy at least one of the constraints of the original CIT model. Words "test" and "configuration" are used interchangeably, though we note in real-world systems one configuration may lead to multiple tests.

UC: Unconstrained CIT. In unconstrained CIT, constraints are ignored during CIT test generation. They are used only to check the validity of the configuration selected during generation. The main advantage is that test generation is simplified and efficient methods that work without constraints can be used. Moreover, in principle, both valid and invalid configurations can be generated - there is no control over model validity. It may happen that the test generation algorithm generates only valid combinations (i.e., $val_S(t)$ for every t in the test suite). This may reduce the effectiveness of the test suite: if only valid tests are generated, one can miss faults only discoverable by invalid tests, as explained in Section 3.1. On the other hand, only invalid tests can be equally useless.

Example 1. In the washing machine example shown in Fig. 1a, UC policy will produce a pairwise test suite with at least 9 test cases, including an invalid test case where `HalfLoad` is set to true in combination with `Spin` equal to 1800.

Test generation. UC can be applied by simply removing the constraints c_i from the original CIT model. The validity of each test can be later computed by checking if the generated configuration satisfies all the c_i. There are several CIT tools that do not handle constraints (for example, those that use algebraic methods for CIT test suite generation), hence can be used with this policy.

CC: Constrained CIT. In this classical approach, constraints are taken into account and only valid combinations among parameters are chosen. Among these parameters a certain level of desired strength is required. The rationale behind this policy is that one wants to test only valid combinations. If a certain interaction among parameters is not possible, then it is not considered even if it would be necessary in order to achieve the desired level of coverage. The main advantage is that no error should be generated by the system. However, this technique can only check one side of equation given in Def. 4, namely that $val_S(p) \rightarrow oracle_I(p)$, since val_S is always true. If the specification is too restrictive, no existing fault will be guaranteed to be found, if it refers to configurations that are invalid.

Example 2. In the washing machine example shown in Fig. 1a, the CC policy produces 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.

Test generation. CC is the classical constrained combinatorial testing (CCIT), and CASA can correctly deal with the constraints and generate only valid configurations. However, CASA requires the constraints in Conjunctive Normal Form (CNF), so CitLab must convert the constraints from general form to CNF.

CV: Constraints Violating CIT. In case one wants to test the interactions of parameters that produce errors, only tests violating the constraints should be produced. This approach is complementary with respect to the CC in which only valid configurations are produced. In CV, we ask that the maximum possible CIT coverage for a given strength is achieved considering only tuples of parameter values that make at least one constraint false (i.e. each test violates the conjunction $c_1 \wedge \ldots \wedge c_n$).

Example 3. In the example presented in Fig. 1a, the CV policy produces 6 test cases, all of which violate some constraint of the model. For instance, a test has `Rinse`=Delicate, `Spin` =800, and `HalfLoad`=false.

Test generation. CV can be applied by modifying the model by replacing all the constraints with $\neg(c_1 \wedge \ldots \wedge c_n)$ and then classical CC is applied.

CuCV: Combinatorial Union. One limitation of the CC technique is that with an over-constrained model, certain faults may not be discovered. On the other hand, by generating test cases violating constraints only, as in CV, certain parameter interactions may not be covered by the generated test suite. In order to overcome these limitations we propose the combination of CC and CV.

Test generation. CuCV is achieved by generating tests using policy CC and policy CV and then by merging the two test suites. Since every test is either valid (in CC) or invalid (in CV), merging the test suites consists of simply making the union of the two test suites.

ValC: CIT of Constraint Validity. CuCV may produce rather big test suites, since it covers all the desired parameter interactions that produce valid configurations *and* all those that produce invalid ones according to the given CIT model. On the other hand, UC may be too weak since there is no control over the final constraint validity and therefore there is no guarantee that the parameter values will influence the final validity of the configuration. On one extreme, UC might produce a test suite without any test violating the constraints. We propose the ValC policy that tries to balance the validity of the tests without requiring the union of valid and invalid tests. ValC requires the interaction of each parameter with the validity of the whole CIT model. That is, both tests that satisfy all the constraints will be generated as well as those that don't satisfy any of the constraints in the given CIT model. Formally, ValC requires that the validity of each configuration $\bar{p}$ (i.e., $val_S(\bar{p})$) is covered in the same desired interaction strength (see Definition 2) among all the parameters.

Example 4. For the WashingMachine, CuCV generates 13 test cases (6+7). ValC requires only 11 test cases.

Test generation. ValC requires to modify the original CIT model by introducing a new Boolean variable validity and replacing all the constraints with one constraint equal to $\mathsf{validity} \leftrightarrow (c_1 \wedge \ldots \wedge c_n)$

CCi: CIT of the Constraints. Every constraint may represent a condition over the system state. For instance, the constraint HalfLoad => Spin < maxSpinHL identifies the critical states in which the designer wants a lower spin speed. One might consider each constraint as a property of the system and be interested in covering how these conditions interact with each other and with the other parameters. The goal is to make the constraints interact with the other system parameters.

Test generation. CCi requires the introduction of a new Boolean variable $\mathsf{validity}_i$ for every constraint, and replacing every constraint c_i with $\mathsf{validity}_i \Leftrightarrow c_i$.

5 Experiments

In order to test our proposed approach we conducted the following experiments. We used 4 case studies to evaluate our proposed approach:

1. **Banking1** represents the testing problem for a configurable Banking application presented in [22].

2. **libssh** is a multi-platform library implementing SSHv1 and SSHv2 written in C[4]. 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 analysed the `cmake` files and identified 16 parameters and the relations among them. We have built a feature model for it in [1] and we have derived from that a CitLab model.

[4] `https://www.libssh.org/`

Model Heartbeat
Parameters:
  Range REQ_Length [0 .. 65535] step 4369;
  Range REQ_PayloadData_length [0 .. 65535] step 4369;
  Range RES_Length [0 .. 65535] step 4369;
  Range REQ_PayloadData_length [0 .. 65535] step 4369;
end
Constraints:
// the declared length in the REQUEST is correct
REQ_Length==REQ_PayloadData_length
// the declared length in the RESPONSE is correct
RES_Length==RES_PayloadData_length
// the RESPONSE has the same length as the REQUEST
REQ_Length==RES_Length
end

Fig. 3. HeartbeatChecker CIT model

3. **HeartbeatChecker** 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 tests secure communication links by allowing a computer at one end of a connection to send a "Heartbeat Request" message. Such a message consists of a payload, typically a text string, along with the payload's length as a 16-bit integer. The receiving computer then must send exactly the same payload back to the sender. HeartbeatChecker reads the data to be used in the Heartbeat from a configuration file with the following schema:

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

Configuration 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). HeartbeatChecker can be considered as an example of a *runtime* configurable system, since thanks to the parameters one can perform different types of tests (with different lengths and payloads). We have written an abstract version of HeartbeatChecker in the combinatorial model shown in Fig. 3: we ignore the actual content of the PayloadData and we model only the lengths: Length represents the declared lengths and PayloadData_length is the actual length of the PayloadData. The constraints represent successful exchanges of messages in the Heartbeat test. The oracle is true if the Heartbeat test has been successfully performed with the specified parameters.

4. **Django** is a free and open source web application framework, written in Python, consisting of over 17k lines of code, that supports the creation of complex, database-driven websites, emphasizing reusability of components[5]. Each

[5] `https://www.djangoproject.com/`

Table 1. Benchmark data. *vr* is the validity ratio, defined as the percentage of configurations that are valid.

name	#var	#constraints	#configurations	*vr*	#pairs
Banking1	5	112	324	65.43%	102
Libssh	16	2	65536	50%	480
HeartbeatChecker	4	3	65536	0.02%	1536
Django	24	3	33554432	18.75%	1196

Django project can have a configuration file, which is loaded every time the web server that executes the project (e.g. Apache) is started. Therefore, the configuration parameters are loaded at *launch time*. In the model we made, among all the possible configuration parameters, we selected and considered 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).

Table 1 presents various benchmark data: number of variables and constraints, size of the state space (the total number of possible configurations), the percentage of configurations that are valid (i.e. the ratio *vr*), the number of pairs that represent the pairwise testing requirements (ignoring constraints). Note that a low ratio indicates that there are only few valid configurations (see, for example, the HeartbeatChecker benchmark). We collected models of real-world systems from different domains, with a good level of diversity (in terms of size, constraints, etc.) in order to increase the validity of our findings.

Experiments were executed on a Linux PC with two Intel(R) i7-3930K CPU (3.2 GHz) and 16 GB of RAM. All reported results are the average of 10 runs with a timeout for a single model of 3600 secs. Test suites were produced using the CASA CIT test suite generation tool according to the pairwise testing criterion.

5.1 Test generation and coverage

In our first experiment, we are interested in comparing the policies in terms of test *effort* measured by the number of tests and by the test suite generation time. Table 2 presents the following data:

- The *time* required to generate the tests and to evaluate their validity (it does not include the evaluation of the $oracle_I$) in seconds.
- The *size* in terms of the number of tests and how many of those are valid (#Val), i.e. val_S returns true.
- The percentage of parameter interactions (pairs) that are covered. In the count of the pairs to be covered, we ignore constraints as in Table 1.

From Table 2 we can draw the following observations:

Table 2. Valid pairwise parameter interactions covered by six test generation policies. (Shaded cells are covered in the prose.) Out of memory errors are due to constraint conversion into the CNF format required by CASA. In particular, as known in the literature, the size in CNF of the negation of a constraint can grow exponentially.

	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%	0.30	8	2	100%	460	268	0	100%

– UC usually produces both valid and invalid tests. However, it may produce all invalid tests (especially if the constraints are strong - see HeartbeatChecker). Having all invalid tests may reduce test effectiveness.

– CC usually does not cover all the parameter interactions, since some of them are infeasible because they violate constraints in the original model. 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 generally does not cover all the parameter interactions, since it produces only invalid configurations. However, in one case (Django) CV covered all the interactions. This means that 100% coverage of the tuples in some cases can be obtained with no valid configuration generated and this may reduce the effectiveness of testing. Sometimes CV is too expensive to perform.

– 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.

– 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 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. If the strength of combinatorial testing is greater or equal to the number of constraints, it guarantees also that valid and invalid configurations are generated.

5.2 Fault detection capability

We are interested in evaluating the fault detection capability of the tests generated by the policies presented above. We have applied mutation analysis [12]

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 ||	I
	H5 == to != (all)	I
	H6 == to != (one)	I
	H7 HeartBleed	I

(a) Seeded Faults (S: in Spec, I: in implementation)

	Is the fault detected?													mut.
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

(b) Fault detection capability of the policies (- means that the test suite was not generated.)

Fig. 4. Fault detection capability

which consists of introducing artificial faults and checking if the proposed technique is able to find them. In our case, we have introduced the faults by hand and then we have applied our technique described in Section 3.1 in order to check if the fault is detected (or *killed*). Tables in Fig. 4 present a brief description of each introduced fault and if each policy was able to kill it.

In principle, our technique is able to find conformance faults both in the model and in the implementation. Indeed, when a fault is found, it is the designer's responsibility to decide what is the source of the fault. For libssh we have modified both the model and the code (the `cmake` script) (faults Lx). For the HeartbeatChecker we have modified the model and the source code (faults Hx). Table in Fig. 4a presents the details of each injected fault, including if it refers to the specification (S) or to the implementation (I).

Table in Fig. 4b reports which faults were killed by each policy. We can observe that the unconstrained CIT (UC) policy performs better than some policies that consider constraints (CC and CV) even if normally their test suites have the same dimensions. However, in some cases (L6) CC detected a fault where UC failed. For CV, CuCV, and ValC we can analyze only the results for libssh, since they did not complete the test generation for HeartbeatChecker. However, even if we restrict to libssh, CuCV has a very good fault detection capability (but it produces the biggest test suite) while ValC and CV scored as well as UC, although they are more expensive, so according to our studies there is no particular reason to justify the use of ValC and CV alone. However, in one case (L3) CV detected a fault that UC did not.

Overall CCi was the best in terms of fault detection, even with test suites as big as those for UC. However, it missed one of the injected faults (L6). CCi was the only one to find the fault H7 (*HeartBleed*). The HeartBleed fault simulates the famous Heartbleed security bug of the OpenSSH implementation of the TLS protocol. It results from improper input validation (due to a missing bounds check) in the implementation of the TLS Heartbeat extension. In detail, the implementation built the payload length of message to be returned based on the

length field in the requesting message, without regard to the actual size of that message's payload. In our implementation, the faulty HeartbeatChecker missed to check that REQ_Length==RES_Length. This proves that testing how parameters can interact with single constraints increases the fault detection capability of combinatorial testing. Our new policies may thus prove useful in detecting faults missed by standard approaches due to the so-called *masking effects* [25].

6 Related Work

The problem of modelling and testing the configurability of complex systems is non-trivial. There has been much research done in extracting constraints among parameter configurations from real systems (problem space) and modelling system configurability [23, 11, 25]. For instance, the importance of having a model of variability and having the constraints in the model aligned with the implementation is discussed in [18]. However, in that paper, authors try to identify the sources of configuration constraints and to automatically extract the variability model. Our approach is oriented towards the validation of a variability model that already exists. Moreover, they target C-based systems that realise configurability with their build system and the C preprocessor. A similar approach is presented in [24], where authors extract the compile-time configurability from its various implementation sources and examine for inconsistencies (e.g., dead features and infeasible options). We believe that our approach is more general (not only compile-time and C-code) and can be complementary used to validate and improve automatically extracted models.

Testing configurable systems in the presence of constraints is tackled in [4] and [21]. In these papers, authors argue that CIT is a very efficient technique and that constraints among parameters should be taken into account in order to generate only valid configurations. This allows to reduce the cost of testing. Also in [2], authors have shown how to successfully deal with the constraints by solving them by using a constraint solver such as a Boolean satisfiability solver (SAT). However, the emphasis of that research is more on testing of the final system not its model of configurability. CIT is also widely used to test SPLs [20].

In SPL the validation and extraction of constraints between features is generally given in terms of feature models (FMs). Synthesis of FMs can be performed by identifying patterns among features in products and in invalid configurations and build hierarchies and constraints (in limited form) among them. For instance, Davril et al. apply feature mining and feature associations mining to informal product descriptions [5]. There exist several papers that apply search based techniques, which generally give better results [10, 17, 6, 16]. However, checking and maintaining the consistency between a SPL and its feature model is still an open problem. A preliminary proposal is presented in [1], which however does not use CIT but a more complex logic based approach. We plan to compare our approach with [1] in order to check if CIT can provide benefits in terms of easiness in test generation and shorter generation times.

7 Conclusions

We proposed a novel approach that extends CIT and aims to automatically check the validity of the configurability model of the system under test. In particular, we described how combinatorial interaction testing techniques can be utilised for this purpose. We devised four original policies that can help software testers discover faults in the model of system configurations as well as faults in the software implementation that the model describes. Several experiments conducted show the efficacy of our approach. We confirm that constraints play an important role in configurability testing, but the experiments show that also invalid configurations should be considered in order to avoid some problems (like over-specification) and to detect a wider range of faults. Our experiments suggest that techniques including both valid and invalid tests (as CuCV) have a better fault detection capability than techniques including only valid (as CC) or invalid tests (as CV). However, producing invalid tests may be not feasible. In these cases we would suggest the tester to use CCi instead of UC and CC. The experiments suggest that CCi is not very expensive and it offers a superior fault detection capability. The techniques presented should significantly help software developers in the modelling and testing process of software systems configurations.

References

1. P. Arcaini, A. Gargantini, and P. Vavassori. Automatic detection and removal of conformance faults in feature models. In *Software Testing, Verification and Validation (ICST), 2016 IEEE 9th International Conference on*, April 2016.
2. A. Calvagna and A. Gargantini. A formal logic approach to constrained combinatorial testing. *Journal of Automated Reasoning*, 45(4):331–358, 2010. Springer.
3. A. Calvagna, A. Gargantini, and P. Vavassori. Combinatorial interaction testing with CitLab. In *Sixth IEEE International Conference on Software Testing, Verification and Validation - Testing Tool track*, 2013.
4. M. Cohen, M. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *Software Engineering, IEEE Trans. on*, 34(5):633–650, 2008.
5. J.-M. Davril, E. Delfosse, N. Hariri, M. Acher, J. Clelang-Huang, and P. Heymans. Feature model extraction from large collections of informal product descriptions, Aug. 22 2013.
6. J. M. Ferreira, S. R. Vergilio, and M. A. Quináiaferreia. A mutation approach to feature testing of software product lines. In *The 25th International Conference on Software Engineering and Knowledge (SEKE) Engineering, Boston, MA, USA, June 27-29, 2013*, pages 232–237, 2013.
7. A. Gargantini and P. Vavassori. CitLab: a laboratory for combinatorial interaction testing. In *Workshop on Combinatorial Testing (CT) In conjunction with International Conference on Software Testing (ICST 2012, April 17-21)*, pages 559–568, Montreal, Canada, 2012.
8. B. J. Garvin, M. B. Cohen, and M. B. Dwyer. An improved meta-heuristic search for constrained interaction testing. In *1st International Symposium on Search Based Software Engineering*, pages 13–22, May 2009.

9. M. Grindal, J. Offutt, and S. F. Andler. Combination testing strategies: a survey. *Softw. Test, Verif. Reliab*, 15(3):167–199, 2005.
10. M. Harman, Y. Jia, J. Krinke, W. B. Langdon, J. Petke, and Y. Zhang. Search based software engineering for software product line engineering: A survey and directions for future work. In *Proceedings of the 18th International Software Product Line Conference (SPLC '14)*, pages 5–18. ACM, 2014.
11. C. Henard, M. Papadakis, G. Perrouin, J. Klein, and Y. L. Traon. Towards automated testing and fixing of re-engineered feature models. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*, pages 1245–1248, 2013.
12. Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011.
13. D. R. Kuhn and V. Okun. Pseudo-exhaustive testing for software. In *30th Annual IEEE / NASA Software Engineering Workshop (SEW-30 2006), 25-28 April 2006, Loyola College Graduate Center, Columbia, MD, USA*, pages 153–158, 2006.
14. D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software fault interactions and implications for software testing. *IEEE Trans. Software Eng*, 30(6):418–421, 2004.
15. R. Kuhn, R. Kacker, Y. Lei, and J. Hunter. Combinatorial software testing. *Computer*, 42(8):94 –96, aug. 2009.
16. R. E. Lopez-Herrejon, J. A. Galindo, D. Benavides, S. Segura, and A. Egyed. Reverse engineering feature models with evolutionary algorithms: An exploratory study. In *Search Based Software Engineering*, pages 168–182. Springer, 2012.
17. R. E. Lopez-Herrejon, L. Linsbauer, and A. Egyed. A systematic mapping study of search-based software engineering for software product lines. *Information and Software Technology*, 61:33 – 51, 2015.
18. S. Nadi, T. Berger, C. Kästner, and K. Czarnecki. Mining configuration constraints: static analyses and empirical results. In P. Jalote, L. C. Briand, and A. van der Hoek, editors, *ICSE*, pages 140–151. ACM, 2014.
19. C. Nie and H. Leung. A survey of combinatorial testing. *ACM Comput. Surv*, 43(2):11, 2011.
20. G. Perrouin, S. Sen, J. Klein, B. Baudry, and Y. Le Traon. Automated and scalable t-wise test case generation strategies for software product lines. In *Proc. of the International Conference on Software Testing (ICST)*, pages 459–468, Paris, France, April 2010. IEEE.
21. J. Petke, M. B. Cohen, M. Harman, and S. Yoo. Practical combinatorial interaction testing: Empirical findings on efficiency and early fault detection. *IEEE Trans. Software Eng.*, 41(9):901–924, 2015.
22. I. Segall, R. Tzoref-Brill, and E. Farchi. Using binary decision diagrams for combinatorial test design. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 254–264. ACM, 2011.
23. J. Shi, M. B. Cohen, and M. B. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *Fundamental Approaches to Software Engineering - 15th International Conference, FASE 2012*, pages 270–284, 2012.
24. R. Tartler, D. Lohmann, J. Sincero, and W. Schröder-Preikschat. Feature consistency in compile-time-configurable system software: Facing the linux 10,000 feature problem. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 47–60, New York, NY, USA, 2011. ACM.
25. C. Yilmaz, E. Dumlu, M. B. Cohen, and A. A. Porter. Reducing masking effects in combinatorial interaction testing: A feedback driven adaptive approach. *IEEE Trans. Software Eng.*, 40(1):43–66, 2014.