Testing Configuration Knowledge-Bases

Franz Wotawa and **Ingo Pill**[1]

Abstract. Writing tests for configuration knowledge-bases is a difficult task. One not minor reason is the huge search space. For exhaustive testing, all possible combinations of configuration parameters must be considered. In practice, exhaustive testing is thus often impossible, due to the sheer, exponential, number of combinations. Consequently it becomes necessary to focus on the most important configurations first. This abstract challenge is well-known in the testing community, and can be addressed by exploiting combinatorial testing. Combinatorial testing deals with reducing the number of test inputs by aiming at exhaustive combinations of parameter subsets. That is, ensuring that a test-suite contains tests covering all value combinations for all parameter subsets for (or up to) a given size. In this paper, we formulate the configuration-testing problem and show how combinatorial testing can be used in a corresponding test case generation process, in order to achieve a huge reduction in the number of required test cases.

1 INTRODUCTION

A configuration, i.e., *something that results from a particular arrangement of parts or components* (according to the Merriam Webster dictionary[2]), can be considered as a system aggregating specific parts in order to implement a desired functionality or behavior. In model-based configuration, we use a knowledge-base in order to represent those components' functionality, given user requirements, and any other knowledge that is necessary for defining or constructing the system. Such additional knowledge encompasses, for instance, constraints prohibiting physically impossible (and thus conflicting) arrangements. Obviously, the outcome of any configuration algorithm depends heavily on the model's quality. In some sense, quality in this case can be considered as being "as close as necessary (and possible) to reality", so that we need to capture the "appropriate" knowledge and do that in the right way.

In case of faults in the knowledge base, e.g., when we miss some constraint that prohibits some impossible configuration, a derived configuration might be incorrect for at least some specific scenarios or corner cases. Thus, testing, which is basically unavoidable for verification and validation problems, is not only essential for hardware and programs, but also for knowledge-bases. We certainly have to ensure that a configuration behaves as desired. The evidence is even stronger when moving from static configuration, e.g., configuring a product based on user needs, to dynamic configurations where the system might adapt itself for a certain situation. For example, a robot might adapt its control behavior in case of a broken wheel, that is, on its view of the world that it stores in an internal knowledge-base as foundation for its reasoning. In such cases, a reliable and, to a certain degree, expected and "safe" behavior has to be ensured.

[1] Technische Universität Graz, email: {wotawa,ipill}@ist.tugraz.at

[2] http://www.merriam-webster.com/dictionary/configuration

In this paper, our focus is on such faults in configuration knowledge-bases an their consequences. Of course, another source for failure is in the configuration algorithm's implementation, i.e., the reasoning engine, itself. While such faults are outside our paper's focus, the generated tests can also be used to test the reasoner.

Regarding fault detection and isolation, the size of a knowledge-base is of certain interest. That is, if the knowledge-base itself, or the configuration space, is very small, exhaustive testing might even be feasible and a valid option for specific situations. However, in case of huge knowledge-bases or huge configuration spaces, exhaustive testing is practically impossible. For instance, and without loosing generality, let us assume the example application of parameter configuration. There the purpose is to find a value assignment to all available system parameters, in order to receive a setup that implements a desired functionality. If we have parameters $p_1, \ldots, p_n$, each taking values from a domain D with size k, an exhaustive search would require us to test k^n possible configurations, which, for the obvious reasons, is most likely infeasible for those values for k and n experienced in practice. Therefore, we require effective alternatives that allow us to systematically focus our testing efforts.

In system testing, where we often have to test in the context of alternative "environments', we suffer from a similar problem. For instance, if we want to test a web page, we have to consider various hardware platforms from PCs to smart phones and tablets, a variety of operating systems, a set of web browsers commonly used, and so on. Testing the web page in the context of all the possible "configurations" is of course an achy task that requires a lot of resources. An empirical study (see e.g., [12]) showed, however, that not all combinations of parameter value assignments are necessary for revealing a bug. Rather, it seems sufficient to consider local parameter configurations. Implementing the concept of combinatorial testing (see also Section 4), we aim to cover all local parameter combinations up to, or of, a given size in a test suite. That is, all the combinations for (all possible) chosen "local" subsets of parameters, which allows us to dramatically reduce the number of required tests. Of course the choice of the subset size directly influences the "locality" of the test case generation process.

In this paper, we discuss the testing problem for configuration knowledge-bases and propose the use of combinatorial testing for automated test input generation. We introduce our preliminary definitions using a simplified example from the e-vehicle domain, and furthermore discuss two different testing aspects. First, we consider testing of different configurations. And second, when considering the desired functionality as being changeable, there arises the question of whether there actually is a valid configuration for a certain combination of functionalities.

Our paper is organized as follows. First, we discuss some related research with a focus on testing of knowledge-based systems in general. Afterwards, we introduce the foundations of configuration using

a running example. We then use the same example to discuss combinatorial testing. After the introduction into combinatorial testing, we discuss testing of configuration knowledge-bases in more detail. Finally, we conclude the paper and outline future research directions.

2 RELATED RESEARCH

Knowledge-based systems are used for various purposes like configuration, diagnosis, and also decision support, e.g., for high-level control of systems. For all these application areas, systems have to be predictable, that is, they have to behave as expected and do not cause any trouble leading to a loss of resources or even harm people. Despite this fact, it is interesting to note that there has not been a huge number of papers dealing with testing, verification, and validation of knowledge-based systems. Robert Plant [17, 18] was one of the first dealing with verification, validation, and testing of expert systems and knowledge-based systems in general. There is also an earlier survey available (see [13]) that deals with tools for validation and verification of knowledge-based systems.

Regarding testing of knowledge-based systems, it is also worth mentioning El-Korany and colleagues' work [5], where their focus is on the testing methodology. There the authors distinguish different cases where testing is required, i.e., inference knowledge testing and task knowledge testing. The objective behind their work was to increase the level of correctness of knowledge-based systems. Other work includes [9], where Hartung and Håkansson discuss test automation for knowledge-based systems. Their approach works for production rules that are extracted from the knowledge-bases.

Hayes and Parzen [10] focused more on the question of "to what degree a knowledge-based systems fulfills its purpose", that is, as indicated in the title of their publication, on achieving the desired behavior. In order to answer the question about the quality of decisions coming from a knowledge-based system, Hayes and Parzen introduced a special metric (QUEM) to judge the quality of the solutions. The proposed approach is essential for measuring the overall performance of a knowledge-based system.

To the best of our knowledge, there is only little work on testing configuration motors or motors that make use of configuration methods like recommenders. Felfernig and colleagues [8, 7] discuss the use of testing, i.e., white-box testing, and development environments in the context of recommender applications. Other work from Felfernig and colleagues [6] mainly focuses on the second step of debugging, i.e., fault localization and correction, but still requires test cases for finding inconsistencies between the behavior coded in a knowledge base and the expected behavior, which originates from knowledge engineers or customers of the configuration system.

Tiihonen and colleagues [20] described a rule-based configurator and also introduced a more or less model-independent testing method. In their approach the configurator is tested using randomly generated requirements given to the configurator. Besides discussing the underlying methodology Tiihonen et al. also presents empirical results gained from 4 different configuration models. In contrast to Tiihonen and colleague the testing approach proposed in this paper is not a random testing approach. Moreover, our focus is more on testing the configuration knowledge-base and not the whole configurator. Although, the obtained tests can be used later for testing concrete implementations.

In our paper, we rely on previous research in the domain of testing knowledge-based systems, but focus on the specific case of knowledge-based systems for configuration. We distinguish different cases for testing and suggest to use a specific testing methodology, i.e., combinatorial testing, which seems to suit configuration very well.

3 THE CONFIGURATION PROBLEM

For illustration purposes, let us consider the following simplified example from the domain of vehicle configurations. In Figure 1, we illustrate an example comprising an electric vehicle that contains an electric motor, electric consumers like an air-condition, and a battery that delivers the required electricity. Battery size and other factors, like the driving mode, substantially influence the range of the vehicle. The configuration knowledge base for our example comprises four components, i.e., an air-condition, a motor, the driving mode, and the battery - each of them offering some options, which then vary from configuration to configuration. Let us now assume that there are two engine types (*standard* and *powerful*), three types of air-condition (*none*, *manual*, and *electronic*), two driving modes (*leisure* and *race*), and three different batteries (*type1*, *type2*, *type3*), each providing a different electric capacity. Clearly, the configured vehicle's range depends heavily on the battery and actual power consumption. That is, for instance, if there is too much power consumption, some particular range can never be achieved. The range, however, reflects an important part of a customer's needs. While customer A is satisfied when she can drive the car for one day in a city for no more than 100 km, customer B expects his car being able to cover more than 200 km before it has to be recharged. Other customer requirements might concern air-conditioning, or the availability of a particular driving mode.

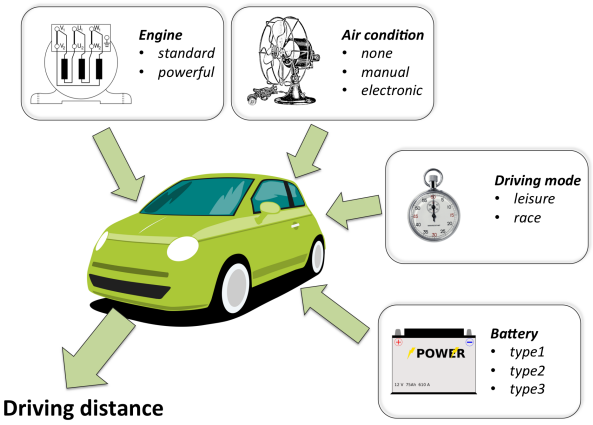

Figure 1. Configuration problem of an electric vehicle

In the following, we discuss the formalization of our e-vehicle configuration example, but let us introduce the definition of a configuration problem first.

Definition 1 (Configuration problem) *A configuration problem is a tuple* $(SD \cup REQ, PARTS, MODES)$ *where SD is the system description, REQ are the requirements, and PARTS are the configurable parts that are allowed to be set to particular modes from MODES.*

We assume that *SD* and *REQ* are first order logic formulae. Other formalisms might also be used requiring the existence of consistency

checks and reasoning capabilities. Our definition of the configuration problem assumes that the functionality or behavior of the parts (from *PARTS*) are defined in *SD* for a particular mode (from *MODES*). For our e-vehicle example, we consider 4 different parts: the electric motor (*emot*), the air-condition (*ac*), the driving mode (*dm*) and the battery (*bat*), i.e., $PARTS = \{emot, ac, dm, bat\}$.

What is missing, is the configuration knowledge and the requirements. Regarding the latter, we assume that we want to distinguish slow acceleration cars (*slowacc*) from fast acceleration cars (*fastacc*), as well as the availability of air-condition cooling (*cooling*). Moreover, a user might specify the maximum distance before recharging, which might be *city* for less than or equal 100 km, *interurban* for distances up to 250 km, and *max*, otherwise. In the following, we depict *SD* for our running example. In the system description, we make use of the predicate *mode* that assigns a component a certain parameter value, *cons* for fixing the power consumption of a part, and *avpow* for stating the available electrical power for batteries.

Electric motor: The standard engine provides slow acceleration only, but draws less electrical power. The powerful motor provides fast acceleration but consumes more electricity as a downside.

$$mode(emot, standard) \rightarrow (cons(emot, 300) \wedge slowacc)$$
$$mode(emot, powerful) \rightarrow (cons(emot, 400) \wedge fastacc)$$

Air-condition: If there is no air-condition, then there is no power consumption and also no cooling. The manual air-condition draws less power than the electronic one. Both provide cooling.

$$mode(ac, none) \rightarrow (cons(ac, 0) \wedge \neg cooling)$$
$$mode(ac, manual) \rightarrow (cons(ac, 100) \wedge cooling)$$
$$mode(ac, electronic) \rightarrow (cons(ac, 150) \wedge cooling)$$

Driving mode: A leisure driver consumes no additional electricity on top of the power required to drive the motor. A racy driver draws more power due to higher acceleration.

$$mode(dm, leisure) \rightarrow (cons(dm, 0))$$
$$mode(dm, race) \rightarrow (cons(dm, 100))$$

Battery: The three battery types have varying capacities.

$$mode(bat, type1) \rightarrow (avpow(bat, 450))$$
$$mode(bat, type2) \rightarrow (avpow(bat, 600))$$
$$mode(bat, type3) \rightarrow (avpow(bat, 800))$$

Other constraints: There are several further domain-dependent constraints: The racy driving mode can only be obtained when having a powerful motor, i.e., it is not possible to have fast acceleration without the right motor.

$$\neg\,(mode(dm, race) \wedge \neg fastacc)$$

In addition, we have to ensure that a component cannot be in more than one mode simultaneously,, and that some available functions are in contradiction, e.g., slow and fast acceleration.

$$\neg\,(mode(emot, standard) \wedge mode(emot, manual))$$
$$\neg\,(mode(none, standard) \wedge mode(manual, manual))$$
$$\neg\,(mode(none, standard) \wedge mode(manual, electronic))$$
$$\neg\,(mode(none, manual) \wedge mode(manual, electronic))$$
$$\neg\,(mode(dm, leisure) \wedge mode(dm, race))$$
$$\neg\,(mode(bat, type1) \wedge mode(bat, type2))$$
$$\neg\,(mode(bat, type1) \wedge mode(bat, type3))$$
$$\neg\,(mode(bat, type2) \wedge mode(bat, type3))$$
$$\neg\,(slowacc \wedge fastacc)$$

The power consumption of all vehicle parts should never exceed the available power, so that we add an integrity constraint:

$$avpow(bat, B) \wedge cons(emot, E) \wedge cons(ac, A) \wedge cons(dm, D) \rightarrow B > (E + A + D)$$

Finally, we have to map power consumption and available power to the vehicle's maximum distance (without recharging) class.

$$avpow(bat, B) \wedge cons(emot, E) \wedge cons(ac, A) \wedge cons(dm, D) \wedge B - (E + A + D) > 99 \rightarrow city$$

$$avpow(bat, B) \wedge cons(emot, E) \wedge cons(ac, A) \wedge cons(dm, D) \wedge B - (E + A + D) > 200 \rightarrow interurban$$

$$avpow(bat, B) \wedge cons(emot, E) \wedge cons(ac, A) \wedge cons(dm, D) \wedge B - (E + A + D) > 400) \rightarrow max$$

It is worth noting that the above definition allows to derive different maximum distances at the same time. If the distance is larger than 400 *city*, *interurban*, and *max* become valid. This could be avoided via integrity constraints or chaining to constraints, in order to get non-overlapping definitions. However, this definition is intended such as to allow to specify a minimum capability.

Now let us we define formally what we understand about a configuration. Intuitively, a configuration has to do with a mode assignment, which corresponds to choosing a certain part, e.g., setting the battery to *type1* means that we want this battery in our configuration.

Definition 2 (Configuration)
Let $(SD \cup REQ, PARTS, MODES)$ be a configuration problem. A configuration is an assignment of a particular mode to each of the parts, i.e., a set C is a configuration, if and only if $|C| = |PARTS|$ and $\forall\, p \in PARTS : \exists\, mode(p, m) \in C$ where $m \in MODES$.

As this definition ignores *REQ* and *SD*, it induces the whole configuration space. Being interested only in valid configurations, i.e., those that do not contradict *REQ* and *SD*, we define them as follows:

Definition 3 (Valid configuration) *Let C be a configuration for the configuration problem $(SD \cup REQ, PARTS, MODES)$. Configuration C is* valid *if and only if $SD \cup REQ \cup C$ is satisfiable.*

Clearly Definition 3 does not ensure that a valid configuration meets the requirements. Hence, we define a suitable configuration.

Definition 4 (Suitable configuration) *Let C be a valid configuration for the configuration problem $(SD \cup REQ, PARTS, MODES)$. C is* suitable *iff the requirements REQ can be derived from the system description and the configuration, i.e., $SD \cup C \models REQ$.*

The user requirements have a direct impact on the space of suitable configurations. Clearly, $REQ = \{city\}$ has more suitable configurations than the requirements $REQ = \{city, cooling\}$. The given definitions of configuration are close to those of reconfiguration and parameter configuration, e.g. from [19, 15]. However, to some extent, generative configuration, e.g., [19], can also be handled, when assuming a boundary for involved components and connections. Each potential component and connection has to be defined in the system description having two modes. One is for indicating the use of a component or connection in a configuration, and the other for stating that the component or connection is not used. In addition, some integrity constraints have to be specified, in order to ensure that in a

final configuration there is no connection without the corresponding components. Note that for larger systems and configurations such a bounded variant might lead to a description that cannot be used for computing configurations in reasonable time, which does not contradict the observation that the given definitions - in principle - allow for specifying different configuration problems.

Let us come back to our running example and the definition of suitable configurations. When stating $REQ = \{city, cooling\}$ we can obtain the suitable configuration

$$\left\{ \begin{array}{c} mode(emot, standard), mode(ac, manual), \\ mode(dm, leisure), mode(bat, type1) \end{array} \right\}$$

but also

$$\left\{ \begin{array}{c} mode(emot, powerful), mode(ac, electronic), \\ mode(dm, leisure), mode(bat, type2) \end{array} \right\}$$

among others. The configuration

$$\left\{ \begin{array}{c} mode(emot, powerful), mode(ac, none), \\ mode(dm, leisure), mode(bat, type1) \end{array} \right\}$$

would be a valid one, but is not suitable as $cooling$ is not established. For computing configurations meeting requirements, we refer the interested reader to [19] or [15].

What remains now, is the question whether the formalized configuration problem represents reality and results in the desired configurations. Hence, we need to test the configuration knowledge-base. To this end, in the next section we introduce a certain testing methodology suitable for this task.

4 COMBINATORIAL TESTING

Combinatorial testing is a method for the algorithmic computation of tests and in particular test input data for a system under test (SUT).

An answer to the question of how much test input data we should generate in order to reveal undetected faults is of great practical importance. As mentioned before, for n inputs with k possible values, an exhaustive approach would require us to test k^n combinations. When missing an important combination, so that a fault remains in the source code, the consequences might be catastrophic, especially for safety-critical systems. Recently, researchers suggested not to consider all input value combinations, but only certain ones focusing on an exhaustive "local" search (see e.g. [3, 23, 24]). The underlying idea is that while input combinations might be required in order to reveal a bug, in practice, we can restrict the size of considered combinations and consider multiple "local" combinations in a test case.

Combinatorial testing formalizes this idea of considering a certain combination of inputs - in our case parameter subsets of size -, e.g., 2 or 3, where all possible value combinations are tried. Regarding the considered combination of inputs we distinguish the strength of combinatorial testing, e.g., strength 2 or 3. Each strength t (where $t \geq 2$) requires that each t-wise tuple of values of the different system parameters is covered at least once in the test suite, which reduces the necessary number of test cases substantially. Of course, the strength t could also be set to the maximum in order to do an exhaustive search. The natural question is then if this method is sufficient. In [12], for example, the authors report on an empirical study considering various programs from different domains and showed that it was enough to consider six-way interactions in order to detect all faults.

We now illustrate combinatorial testing in the context of our running example, where we restrict our focus purely on the testing methodology. Even more details are offered in the next section. For brevity let us consider the component modes as inputs:

input	values
$emot$	$standard, powerful$
ac	$none, manual, electronic$
dm	$leisure, race$
bat	$type1, type2, type3$

When searching for all two-way combinations, i.e., combinations of values for two particular inputs, we would obtain results similar or equivalent to the one depicted in Table 1. There for each combination of two inputs, all possible value combinations are given, which results in 9 test cases. For comparison reasons, we also depict the test cases for strength 3 in Table 2. It is worth noting that, when considering all combinations, we would finally obtain 36 test cases.

Table 1. All two-way interactions for the e-vehicle example

	$emot$	ac	dm	bat
1	$powerful$	$none$	$race$	$type1$
2	$standard$	$none$	$leisure$	$type2$
3	$powerful$	$none$	$leisure$	$type3$
4	$standard$	$manual$	$race$	$type1$
5	$powerful$	$manual$	$leisure$	$type2$
6	$standard$	$manual$	$race$	$type3$
7	$powerful$	$electronics$	$leisure$	$type1$
8	$standard$	$electronics$	$race$	$type2$
9	$powerful$	$electronics$	$race$	$type3$

The significant advantage of combinatorial testing is that the number of test cases can be reduced while still considering combinations of input values. In order to implement combinatorial testing as a test case generation method, the following steps are required:

1. First, someone has to write a model of the input space, comprising the inputs and their value domains.
2. The combinatorial design procedure takes this input space and generates an array where each row is simple a test case describing the value for each input considering the given strength t.
3. Every row is delivered back as a single test case describing potential input data (but not the expected output).

Another benefit of combinatorial testing is that Steps 2 and 3 can be automated completely. There are tools available for computing the test cases, e.g., the ACTS combinatorial test generation tool [16]. ACTS has been developed jointly by the US National Institute Standards and Technology (NIST) and the University of Texas at Arlington and currently has more than 1,400 individual and corporate users.

A drawback of combinatorial testing is that only test *input* data is generated. Hence, the oracle problem, i.e., classifying the computed output as being correct or not, still remains for combinatorial testing. However, at least, combinatorial testing offers a structured and well defined method for test input data generation that can be effectively used in practice.

Regarding an algorithm for computing test cases using combinatorial testing, we refer the reader to the available literature. The underlying data structure for computing the test is the mixed-level covering array which can be defined as follows (see [4]).

Definition 5 *A mixed-level covering array which we will denote as* $MCA(t, k, (g_1, \ldots, g_k))$ *is an* $k \times N$ *array in which the entries of*

Table 2. All three-way interactions for the e-vehicle example

	$emot$	ac	dm	bat
1	$standard$	$none$	$leisure$	$type1$
2	$powerful$	$none$	$race$	$type1$
3	$standard$	$none$	$race$	$type2$
4	$powerful$	$none$	$leisure$	$type2$
5	$standard$	$none$	$leisure$	$type3$
6	$powerful$	$none$	$race$	$type3$
7	$standard$	$manual$	$race$	$type1$
8	$powerful$	$manual$	$leisure$	$type1$
9	$standard$	$manual$	$leisure$	$type2$
10	$powerful$	$manual$	$race$	$type2$
11	$standard$	$manual$	$race$	$type3$
12	$powerful$	$manual$	$leisure$	$type3$
13	$standard$	$electronics$	$leisure$	$type1$
14	$powerful$	$electronics$	$race$	$type1$
15	$standard$	$electronics$	$race$	$type2$
16	$powerful$	$electronics$	$leisure$	$type2$
17	$standard$	$electronics$	$leisure$	$type3$
18	$powerful$	$electronics$	$race$	$type3$

the i-th row arise from an alphabet of size g_i. Let $\{i_1, \ldots, i_t\} \subseteq \{1, \ldots, k\}$ and consider the subarray of size $t \times N$ by selecting rows of the MCA. There are $\prod_{i=1}^{t} g_i$ possible t-tuples that could appear as columns, and an MCA requires that each appears at least once. The parameter t is also called the strength of the MCA.

The mixed level covering array defines all possible combinations of t inputs having a finite value domain of g_i for an input i. It is worth noting that in combinatorial testing we have to have finite domains (which is perfectly fine in case of configuration knowledge-bases). We might also remark that the technique for discretizing the parameter values is referred to as input parameter modeling in combinatorial testing [11]. After discussing combinatorial testing, we show how combinatorial testing can be effectively used for testing configuration knowledge-bases in the next section.

5 TESTING KNOWLEDGE-BASES

The obvious purpose of testing is to reveal a SUT's faults. To this end, the SUT is executed using certain input values, and the resulting behavior is logged. This behavior is compared with the expected one. In case of deviations, a fault is detected and we certainly get interested in the corresponding root causes. In his ACM Turing Lecture 1972, Edsger W. Dijkstra mentioned that "*program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence*". Hence, someone might be interested in efficiently detecting deviations, i.e., finding the right input that causes the misbehavior. Finding such an input might be like finding a needle in a haystack. Testing methods like combinational testing help in this respect.

For a more detailed view on testing, we recommend Myers book [14], where he - aside covering other issues - introduces 10 testing principles. In the 5th one, Myers mentions that "*test cases must be written for input conditions that are invalid and unexpected as well as for those that are valid and expected*". Hence, there is a requirement not only to test for expected results, but also to execute a SUT using input values for which the SUT was not designed. In case of a configuration knowledge base, this means that we have to use also queries where we expect no solution due to inconsistencies arising during resolution.

Testing is based on test cases. We formalize test cases in a simplified form appropriate for our purposes.

Definition 6 (Test case) *A test case for a SUT is a tuple (IN, OUT) where IN is a formalization of the input values, and OUT defines the expected output when executing the SUT using IN.*

We say that a test case (IN, OUT) is a passing test case for a SUT if the execution of SUT using *IN* returns an output that is not in contradiction with *OUT*. Otherwise, we say that the test case is a failing test case. A test suite is a set of test cases. In order to test a SUT, we are interested in having a test suite that comprises at least one failing test case. If there is no such test case, we assume the SUT to be correct with respect to the test suite.

After discussing some basic testing principles, the question remains of how to actually test configuration knowledge-bases. According to Definition 1, the formalized knowledge covers the system description *SD* and the requirements *REQ*. What we actually want to ensure is that when querying the knowledge-base using a certain request, we obtain the expected result. Hence, for testing purposes, we are interested mainly in testing *SD* and not *REQ*.

There are some additional aspects when discussing testing configuration knowledge-bases. For testing ordinary programs, the role of input and output variables is well known. For configuration problems, someone might, however, also consider *REQ* as input and the set of suitable configurations *SCONF* as output. It might also be desirable to ask for the requests to be obtained when assuming a certain configuration. In terms of configurations, most likely there are some valid configurations that are not suitable. Others are not even valid. According to Myers 5th testing principle, however, we also have to check the invalid and unexpected cases.

We now formalize these two testing problems. Let us assume a system description *SD* that describes configuration knowledge regarding *PARTS* and *MODES*. The first testing problem is for checking whether the derived suitable configurations are the correct ones.

Definition 7 (Testing configuration) *The* testing configuration *problem concerns testing the knowledge-base in its capabilities for deriving the expected configurations, and can be characterized as follows:*

Input: *SD, PARTS, and MODES*
Objective: *Finding test cases of the form $(REQ, SCONF)$, where REQ are requirements, and CONF is a set of expected configurations for the configuration problem $(SD \cup REQ, PARTS, MODES)$. Note that CONF might be empty in case of inconsistencies. Otherwise, CONF is expected to comprise suitable configurations only.*

The second testing problem is related to checking whether given configurations lead to the derivation of the correct requirements, if there are any.

Definition 8 (Testing requirement derivation) *The* testing requirement derivation *problem captures the case where we are interested in testing the capabilities of the knowledge base to derive requirements from conflicts. It can be characterized as follows:*

Input: *SD, PARTS, and MODES*
Objective: *Finding test cases, of the form (C, R) where C is a configuration and R is the expected result. Obviously, R might be $\bot$ in case the configuration itself lead to an inconsistency, i.e., $SD \cup C \models \bot$. R might comprises all requests for which C is a suitable configuration, or might be empty if there are no requests for which C is suitable.*

In order to solve both configuration specific testing problems, we need a method for computing input values, i.e., requirements respectively configurations, and the resulting values. For the first part, we can easily make use of combinatorial testing with the advantage of a reduced number of test cases to be computed while still retaining the capabilities for revealing a faulty behavior. Computing the expected outcome in an automated fashion, however, is not directly possible, because of a missing specification. Hence, we have to rely on the knowledge engineer to provide this information. In the testing community, this problem is referred to as the *oracle problem*. There are some related methods like model-based testing (e.g., see [22, 21]) or metamorphic testing (e.g., see [1, 2]). The latter uses symmetries in the functions or systems to be tested in order to gain information about the correct behavior. For example, when testing the sinus function implementation, we can make use of the property $\sin(x) = \sin(2\pi + x)$. If available, such techniques can be also used for testing configuration knowledge-bases. However, in the following we discuss the overall testing process ignoring metamorphic testing.

Algorithm 1 TEST_CONF(*SD*, *PARTS*, *MODES*, *CM*)

Input: A system description *SD*, its component set *PARTS*, their modes *MODES*, and a combinatorial testing model *CM* for requirements.
Output: A test suite *TS* where also the result of the test is stored for each test case

1: *TS* := ∅
2: $t := 2$
3: *flag* := *FALSE*
4: **repeat**
5: Call the combinatorial testing algorithm using *CM* and t and store the result in T.
6: **for all** $t \in T$ **do**
7: Convert t to its corresponding requirements representation *REQ*.
8: Call the configuration engine on (*SD* ∪ *REQ*, *PARTS*, *MODES*) and store the result in *SCONF*.
9: Present *REQ* and *SCONF* to the user for obtaining a classification *UC* ∈ {**PASS**, **FAIL**, ?}
10: **if** *UC* = **FAIL** **then**
11: Ask the user for *SCONF*
12: *flag* = *TRUE*
13: **end if**
14: *TS* := *TS* ∪ {(*REQ*, *SCONF*, *UC*)}
15: **end for**
16: $t := t + 1$
17: **until** $flag$ or no more t-way combinations are possible
18: **return** *TS*

In the proposed testing methodology for configuration knowledge-bases, we make use of combinatorial testing for generating the inputs for both problems, the *testing configuration* as well as the *testing requirement derivation* problem. We use these inputs, and a configuration engine (respectively a theorem prover) for generating the current output. The input and the corresponding output is given to the user (e.g., the knowledge engineer) for classifying the result as **FAIL** or **PASS**. Note that we also have to consider that the user has no clear understanding of the expected outcome. In this case, the classification inconclusive (i.e., ?) can be used. This test input generation and classification process that keeps the user in the loop, is started considering 2-way combinations. If no **FAIL** is obtained, the process can be continued for 3-way combinations or even stronger ones, of course re-using previously obtained classifications. The process can definitely stop when strength t in combinatorial testing (for obtaining t-way combinations) reaches the number of variables used. Experimental surveys suggest that it seems enough to consider 6-way combinations (see [12]).

Algorithm 1 summarizes the steps necessary for computing a test suite in order to solve the *testing configuration* problem. The algorithm for solving *testing requirement derivation* problem is very similar. Algorithm 2 shows the necessary steps. The only differences are in the for-loop of the algorithm, where we have to take care of the different situations. Both algorithms terminate assuming a finite set of requirements and configurations. When ignoring the time required for theorem prover, computing a configuration, and user interaction, the time required for executions is mainly bound by the time required for combinatorial testing.

Algorithm 2 TEST_REQ(*SD*, *PARTS*, *MODES*, *CM*)

Input: A system description *SD*, its component set *PARTS*, their modes *MODES*, and a combinatorial testing model *CM* for configurations.
Output: A test suite *TS* where also the result of the test is stored for each test case

1: *TS* := ∅
2: $t := 2$
3: *flag* := *FALSE*
4: **repeat**
5: Call the combinatorial testing algorithm using *CM* and t and store the result in T.
6: **for all** $t \in T$ **do**
7: Convert t to its corresponding configuration representation C.
8: **if** $SD \cup C \models \bot$ **then**
9: $R := \bot$.
10: **else**
11: Call the the theorem prover with input $SD \cup C$ and store the derivable requirements in R.
12: **end if**
13: Present C and R to the user for obtaining a classification *UC* ∈ {**PASS**, **FAIL**, ?}
14: **if** *UC* = **FAIL** **then**
15: Ask the user for R
16: *flag* = *TRUE*
17: **end if**
18: *TS* := *TS* ∪ {(C, R, *UC*)}
19: **end for**
20: $t := t + 1$
21: **until** *flag* or no more t-way combinations are possible
22: **return** *TS*

Finally, it is worth discussing the computation of combinatorial tests in Algorithm 1 and Algorithm 2. For Algorithm 2, we already computed the test cases in the previous section. See, for example, Table 1 for all two-way combinations. There, test case 4 would lead to an inconsistency when calling the theorem prover, because the *standard* motor would lead to *slowacc* which contradicts the rules $\neg(mode(dm, race) \wedge \neg fastacc)$ in combination with $\neg(slowacc \wedge fastacc)$. Hence, we would be able to detect the case

where a knowledge-base is missing some of the mentioned rules.

For obtaining the combinatorial tests for Algorithm 1, the situation is a little different (but not much). There, we are interested in requirement combinations. As discussed before, there might be cases where we do not want to specify all requirements. Hence, we have to find a model for the combinatorial testing algorithm where we are able to take not care on a certain requirement. For our e-vehicle example, we have three different requirement categories: cooling, driving distance, and acceleration, each of them with the following possible values:

input	values
cooling	*true*, *false*, -
driving distance	*city*, *interurban*, *max*, -
acceleration	*slowacc*, *fastacc*, -

Note that the value - is used to indicate that this requirement is currently not active. When using this model as input to the ACTS tool, we are able to obtain 12 combinatorial tests of strength 2 depicted in Table 3. Each row comprises requirements for our configuration model. Some of the requirements may lead to suitable configurations, some may not. This clarification has to be performed when considering the test cases in Algorithm 1.

Table 3. All two-way interactions for the requirements of the e-vehicle

	driving distance	acceleration	cooling
1	*city*	*slowacc*	*false*
2	*city*	*fastacc*	-
3	*city*	-	*true*
4	*interurban*	*slowacc*	-
5	*interurban*	*fastacc*	*true*
6	*interurban*	-	*false*
7	*max*	*slowacc*	*true*
8	*max*	*fastacc*	*false*
9	*max*	-	-
10	-	*slowacc*	*true*
11	-	*fastacc*	*false*
12	-	-	-

From the results obtained using our running example we are able to conclude that combinatorial testing – in principle – can be used to solve the two testing problems, which correspond to configuration knowledge-bases. These two problems correspond to the two different questions someone would ask during and after the development of configuration knowledge-bases. The first question, deals with the challenge of ensuring whether a knowledge-base is able to derive expected configurations. The second question is related to the evaluation whether a knowledge-base allows for deriving configurations that fulfill the given requirements. Both questions have to be addressed within the development of configurators and their knowledge-bases in order to gain trust in their correctness.

6 CONCLUSION

In this paper, we raised the question of how to test configuration knowledge-bases. We focused on model-based configuration and defined two testing problems. One for checking whether obtained configurations are in line with the requirements, and the other for testing whether the correct set of configurations is returned for given requirements. The proposed testing method relies on combinatorial testing for computing input data needed. We argued that combinatorial testing is very well suited for configuration testing, because of ensuring a good fault detection capability while still reducing the number of input combinations to consider. In practice, limited combinations, i.e., five- to six-way combinations have turned out to be sufficient for revealing faults that have not been found before. The question whether, e.g., six-way combinations are enough for configuration knowledge-base testing, will have to be addressed by future research and corresponding experiments.

In future research also the proposed approach has to be empirically evaluated. For such an evaluation, large configuration knowledge-bases should be used. Moreover, by introducing faults in the knowledge-bases someone would be able to check, whether the proposed approach is capable of detecting faults. Ideally, the fault detection capabilities should be compared with other approaches, e.g., random testing. Another interesting question is due to the testing capabilities of existing knowledge-based configuration tools. Do they support testing? Which testing strategies do they suggestion? These two questions among others can be answered, when carrying out a case study with the objective of evaluating existing configuration solutions. It is worth noting that we focussed more on the principles of testing configuration knowledge-bases in this paper and provided a solution. We leave a detailed empirical analysis of the proposed approach for future research.

ACKNOWLEDGEMENTS

The research presented in this paper has been carried as part of the eDAS project funded by the European Commission FP-7 grant agreement number: 608770.

REFERENCES

[1] T.Y. Chen, S.C. Cheung, and S.M. Yiu, 'Metamorphic testing: a new approach for generating next test cases', Technical report, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, (1998). Technical Report HKUST-CS98-01.

[2] T.Y. Chen, J. Feng, and T.H. Tse, 'Metamorphic testing of programs on partial differential equations: a case study', in *Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC '02)*, pp. 327–333, Los Alamitos, CA, (2002). IEEE Computer Society.

[3] David M. Cohen, Siddhartha R. Dalal, Michael L. Fredman, and Gardner C. Patton, 'The AETG system: An approach to testing based on combinatorial design', *IEEE Trans. Softw. Eng.*, **23**(7), 437–444, (1997).

[4] Charles J. Colbourn, 'Covering arrays', in *Handbook of Combinatorial Designs*, eds., Charles J. Colbourn and Jeffrey H. Dinitz, Discrete Mathematics and Its Applications, 361–365, CRC Press, Boca Raton, Fla., 2nd edn., (2006).

[5] Abeer El-Korany, Ahmed Rafea, Hoda Baraka, and Saad Eid, 'A structured testing methodology for knowledge-based systems', in *11th International Conference on Database and Expert Systems Applications (DEXA)*, pp. 427–436. Springer, (2000).

[6] A. Felfernig, G. Friedrich, D. Jannach, and M. Stumptner, 'Consistency-based diagnosis of configuration knowledge bases', *Artificial Intelligence*, **152**(2), 213–234, (2004).

[7] A. Felfernig, G. Friedrich, D. Jannach, and M. Zanker, 'An integrated environment for the development of knowledge-based recommender applications', *International Journal of Electronic Commerce (IJEC)*, **11**(2), 11–34, (2006).

[8] A. Felfernig, K. Isak, and T. Kruggel, 'Testing knowledge-based recommender systems', *OEGAI Journal*, **4**, 12–18, (2005).

[9] Ronald Hartung and Anne Håkansson, 'Automated testing for knowledge based systems', in *Knowledge-Based Intelligent Information and Engineering Systems*, eds., Bruno Apolloni, RobertJ. Howlett, and Lakhmi Jain, volume 4692 of *Lecture Notes in Computer Science*, 270–278, Springer Berlin Heidelberg, (2007).

[10] Caroline C. Hayes and Michael I. Parzen, 'Quem: An achievement test for knowledge-based systems', *IEEE Transactions on Knowledge and Data Engineering*, **9**(6), 838–847, (November/December 1997).

[11] D.R. Kuhn, R.N. Kacker, and Y. Lei, *Introduction to Combinatorial Testing*, Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series, Taylor & Francis, 2013.

[12] D.R. Kuhn, R.N. Kacker, Y. Lei, and J. Hunter, 'Combinatorial software testing', *Computer*, 94–96, (August 2009).

[13] Stephen Murrell and Robert T. Plant, 'A survey of tools for the validation and verification of knowledge-based systems: 1985-1995', *Decision Support Systems*, **21**(4), 307–323, (1997).

[14] Glenford J. Myers, *The Art of Software Testing*, John Wiley & Sons, Inc., 2 edn., 2004.

[15] Iulia Nica and Franz Wotawa, '(re-)configuration of communication networks in the context of m2m applications', in *Proceedings of the 15th Workshop on Configuration*, Vienna, Austria, (2013).

[16] NIST, *User Guide for ACTS*. which is online available at csrc.nist.gov/groups/SNS/acts/documents/acts_user_guide_v2_r1.1.pdf; last visited on June 20th, 2014.

[17] Robert Plant, 'Rigorous approach to the development of knowledge-based systems', *Knowl.-Based Syst.*, **4**(4), 186–196, (1991).

[18] Robert T. Plant, 'Expert system development and testing: A knowledge engineer's perspective', *Journal of Systems and Software*, **19**(2), 141–146, (1992).

[19] Markus Stumptner, Gerhard Friedrich, and Alois Haselböck, 'Generative constraint-based configuration of large technical systems', *AI EDAM*, **12**, 307–320, (9 1998).

[20] Juha Tiihonen, Timo Soininen, Ilkka Niemelä, and Reijo Sulonen, 'Empirical testing of a weight constraint rule based configurator', in *ECAI 2002 Configuration Workshop*, pp. 17–22, (2002).

[21] J. Tretmans, 'Model-based testing and some steps towards test-based modelling', in *Proceedings of the 11th International School on Formal Methods for Eternal Networked Software Systems (SFM 2011)*, (2011).

[22] M. Utting and B. Legeard, *Practical Model-Based Testing - A Tools Approach*, Morgan Kaufmann Publishers Inc., 2006.

[23] Cemal Yilmaz, Myra B Cohen, and Adam A Porter, 'Covering arrays for efficient fault characterization in complex configuration spaces', *Software Engineering, IEEE Transactions on*, **32**(1), 20–34, (2006).

[24] Linbin Yu, Yu Lei, R.N. Kacker, and D.R. Kuhn, 'Acts: A combinatorial test generation tool', in *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, pp. 370–375, (2013).