Coverage-Based Metrics for Cloud Adaptation *

Yonit Magid
IBM Research – Israel

Rachel Tzoref-Brill
IBM Research – Israel

Marcel Zalmanovici
IBM Research – Israel

ABSTRACT

This work introduces novel combinatorial coverage based metrics for deciding upon automated Cloud infrastructure adaptation. Our approach utilizes a Combinatorial Testing engine, traditionally used for testing at the development phase, in order to measure the load behavior of a system in production. We determine how much the measured load behavior at runtime differs from the one observed during testing. We further estimate the involved risk of encountering untested behavior in the current configuration of the system as well as when transitioning to a new Cloud configuration using possible adaptation actions such as migration and scale-out. Based on our risk assessment, a Cloud adaptation engine may consequently decide on an adaptation action in order to transform the system to a configuration with a lesser associated risk.

Our work is part of a larger project that deals with automated Cloud infrastructure adaptation. We introduce the overall approach for automated adaptation, as well as our coverage-based metrics for risk assessment and the algorithms to calculate them. We demonstrate our metrics on an example setting consisting of two sub-components with multiple instances, comprising a typical installation of a telephony application.

CCS Concepts

•**Software development process management** → Risk management; •**Distributed architectures** → Cloud computing;

Keywords

Cloud Adaptation, Combinatorial Coverage, Load Coverage, Risk Metrics

*The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610802.

QUDOS'16, July 21, 2016, Saarbrücken, Germany
© 2016 ACM. 978-1-4503-4411-1/16/07...$15.00
http://dx.doi.org/10.1145/2945408.2945409

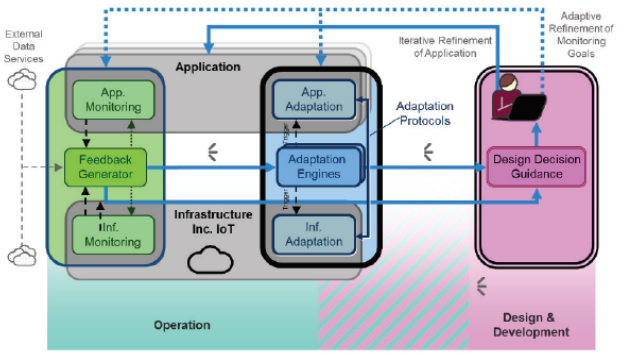

Figure 1: **High-level architecture of the CloudWave project. Our work fits into the middle blue part where the adaptation engine uses our metrics and other metrics to decide whether adaptation is needed and which adaptation action to take.**

1. INTRODUCTION

Adaptation of Cloud infrastructure is a major challenge that both stems from and contributes to the uncontrollability and unpredictability of the Cloud. It refers to actions such as migration (moving VMs from one host to another), scale-out (addition of VMs) and scale-up (addition of hardware resources) in order to adapt Cloud services to their environment based on changing needs and runtime behavior.

Adaptation is currently mostly done while considering only the status of the infrastructure, which ignores the needs of Cloud applications and their interactions with other applications. However, the vision of the CloudWave project[1] is to reach dynamic coordinated adaptation in order to improve service quality and optimized resource use.

The high-level architecture of the CloudWave dynamic adaptation is presented in Figure 1. It consists of three parts. The first is a monitoring component, shown on the left side in green. It collects both hardware metrics and application-specific information using any of a plethora of monitoring frameworks. The second part is an adaptation component, shown in the middle in blue. It consists of a number of sub-engines, each providing an evaluation of the current Cloud configuration and assessment of potential adaptation actions, based on the monitoring information fed to it. These are used by the central adaptation engine

[1]http://cloudwave-fp7.eu/

to trigger an adaptation action as well as to decide upon its nature. The third part is feedback-driven development, shown on the right side in purple. This component is used by the human-in-the-loop for agile development, i.e. manually improving the application based on feedback from the cloud infrastructure operations.

We implemented the work described in this paper as a sub-engine in the adaptation component. We present novel metrics that (1) assess the risk associated with the current Cloud configuration and (2) assist in reaching automated decisions about infrastructure adaptation actions. We base our metrics on the notion of combinatorial coverage, traditionally used at testing phase. Our metrics can be used in combination with metrics provided by other sub-engines, e.g., based on factors such as cost or energy, to reach a coordinated adaptation decision.

The reasoning behind the use of combinatorial coverage as a risk metric is as follows. Experience shows that in order to effectively reason about the overall load on the system, one needs to consider the loads that the different subcomponents experience. Certain system behaviors may be observed only when a specific combination of loads on the sub-modules is experienced. For example, a certain erroneous behavior may be exposed only when subcomponent A exhibits high CPU utilization *and* subcomponent B exhibits high memory utilization. Furthermore, we observe that load combinations experienced in production that were not seen during testing are more likely to cause the system to enter untested states, and hence increase the overall risk for failure. Based on this observation, we use Combinatorial Testing (CT), a well known technique for covering combinations of parameter-values in a test suite, to compute the load combination coverage achieved during testing and compare it with the coverage observed during production. When the two differ over a certain threshold, the risk imposed by the current Cloud configuration may require an infrastructure adaptation action. We then estimate for each possible adaptation action, what further combinations it would add to the space of load combinations not yet tested. Based on these measurements, we compute the overall coverage-based risk associated with different adaptation actions.

By performing the above, we extend the classical use of CT from test design phase to production phase, and from measuring combinatorial coverage to predicting missing coverage following potential adaptation actions.

We note that due to the uncontrollability and unpredictability of the Cloud, even when CT is used to design load tests, it is unlikely that the executed tests will manage to reach all load combinations introduced by the designed tests. Hence, we can assume that even before any adaptation action is taken, there are already untested load combinations that may be experienced in production.

We further note that we consider production load combinations unseen during testing as risky only for a certain window of time. If this window passes without a dramatic increase in the number of unseen combinations (which would have triggered an adaptation action), and in addition the system does not exhibit any problematic behavior, we assume that the new load combinations observed in this window are safe and add them to the set of tested combinations.

We implemented our metrics using a non-trivial efficient symbolic computation on top of our commercial, industrial-strength CT tool IBM Functional Coverage Unified Solution (IBM FOCUS) [8, 14]. We demonstrate the feasibility of our approach on an example setting consisting of two sub-components with multiple instances, comprising a typical installation of a telephony application. Our preliminary assessment showed that our risk and ranking metrics indeed behaved as expected, i.e., correlated to increasing load not previously experienced in the system, and preferred adaptation actions that introduced less previously unseen behavior.

Related Work. All Cloud providers - Azure, AWS, etc - monitor the state of used resources and perform adaptation either on behalf of a user request, e.g., when the user pays for more CPU power, or automatically within a predefined range, commonly based on load or cost considerations. Some of the existing methods of providing this kind of elasticity are surveyed in [7]. Our metrics can be used in combination with the existing ones to reach a coordinated adaptation decision. Our added value is that based on coverage information we can differentiate between possible adaptation actions when the existing methods do not, i.e., determine which adaptation action will result in the most previously tested/experienced workloads. We can also provide new triggers for adaptation in potentially risky situations that are not necessarily detected by the existing methods.

While there is a large body of work on the use of combinatorial coverage at testing phase [9, 13], we are unaware of work that utilizes it at production phase, for the purpose of Cloud adaptation or otherwise.

To conclude, our contributions are as follows: a novel approach for risk assessment of Cloud configurations based on combinatorial coverage, while extending its traditional use from testing to production; a novel approach for ranking of Cloud infrastructure adaptation actions based on the computation of coverage gaps that will be introduced following each action; and a scalable symbolic implementation in an industrial-strength commercial CT tool.

Our adaption of notions and practices from testing phase to production phase aligns with the DevOps paradigm, where the boundaries between testing and production environments for Cloud applications are no longer distinct.

2. BACKGROUND

In the following we provide background on the different technologies used in our solution – CT, which we use to define the load coverage; binary decision diagrams, which we use to compute it; and Cloud monitoring, which we use to collect the observed load during execution.

Combinatorial Testing. Combinatorial testing (CT) is a well-known test design technique, considered a testing best practice [2, 6, 16]. In CT, the test space is manually modeled by a set of parameters, their respective values, and constraints on the value combinations. The aggregate of parameters, values, and constraints is called a *combinatorial model*. A valid test in the test space is defined to be an assignment of one value to each parameter without violating the constraints. Traditionally, CT is used to design test cases, as follows. A subset of the test space, termed a *test plan*, is automatically constructed so that every valid combination of t values appears in some test in the test plan constructed by CT, where t is usually a user input. This systematic design of test plans is based on empirical data that shows that in most cases, the appearance of a bug depends on the interaction between a small number of features of the system under test [6, 10, 15]. In this work, the CT

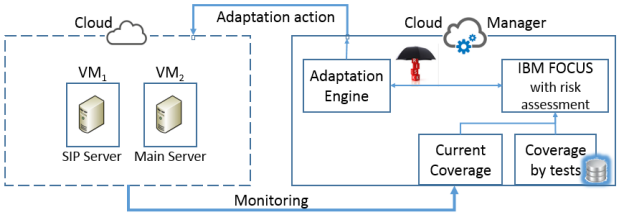

Figure 2: **A typical installation of a telephony application, along with our risk assessment sub-engine running on a Cloud manager.**

model represents different load parameters for the system under load, and is described in detail in Section 3. We use CT in a non-traditional way to compute the combinatorial coverage that was actually achieved during testing and during production. To this aim, we use the IBM FOCUS tool [8, 14], an industrial-strength commercial CT tool.

Using Binary Decision Diagrams to Represent Combinatorial Models. In [14], a compact representation of combinatorial models using Binary Decision Diagrams (BDDs) was presented. BDDs [1] are a compact data structure for representing and manipulating Boolean functions, commonly used in formal verification [4] and in logic synthesis [11]. [14] utilizes the efficient computation of Boolean operations on BDDs such as negation, conjunction and disjunction, to compute the BDD representing the set of valid tests from the user-specified constraints. Multi-valued parameters are handled using standard Boolean encoding and reduction techniques to BDDs [12]. The set of valid tests is represented by the negation of the BDD for the invalid tests, conjunct with a BDD that represents the legal multi-valued to Boolean encodings of the parameter values. We utilize this BDD-based representation of the combinatorial model as the basis for the computation of our coverage-based risk metrics. The computation is described in detail in Section 4.

Cloud Monitoring. Cloud application monitoring is an emerging topic among cloud technologies [17]. It was originally used for billing purposes, but recently interest has risen in exploiting it for other purposes, e.g. leveraging monitoring data for DevOps, including coordinated adaptation and feedback-driven development (FDD) [5]. Cloud platforms supply open source monitoring tools and APIs such as Ceilometer, Ganglia, Nagios, and Zabbix. For the work described in this paper we used Ceilometer [3], which provides all the basic information about the OpenStack components, such as Nova-Compute, Swift, Cinder, etc. It also provides support for the generation of monitoring events.

3. EXAMPLE

As a running example, consider an installation of a telephony application, as depicted in Figure 2. The installation consists of two main components, a Session Initiation Protocol (SIP) and a main server, each running on its own virtual machine. We will use this example to demonstrate the basic notions and information which our approach builds upon.

Commonly, triggers for adaptation of the infrastructure in which the application is running are based on user requests and/or resource utilization and cost considerations. We would like to add to these inputs also coverage considerations, i.e., considering the workload combinations which the application already experienced without failures, which can provide added value in adaptation-related decisions.

Table 1: **Example telephony load coverage model**

Parameter	Values
VM1 CPU	Low, Medium, High
VM1 Memory	Low, Medium, High
VM1 Net In	Low, Medium, High
VM1 Net Out	Low, Medium, High
VM1 Machine ID	Gr_1, Gr_2
VM2 CPU	Low, Medium, High
VM2 Memory	Low, Medium, High
VM2 Net In	Low, Medium, High
VM2 Net Out	Low, Medium, High
VM2 Machine ID	Gr_1, Gr_2

To that aim, we present in Table 1 the combinatorial model that captures all possible load combinations on the different resources in the VMs of the application. This model is used to compute the load combination coverage at testing phase and during production. In this example, we will focus on the CPU, memory and network hardware parameters (though this can be generalized to other types of resources). The possible load values are divided into low, medium, and high ranges. In addition, the model contains a machine ID for each VM to enable capturing changes in the load coverage space that will result from migration to a different physical machine. We enable abstracting similar type machines into group IDs, i.e., the ID may represent a group of machines that have the same physical attributes. The resulting model in our example consists of the 10 parameters (5 per VM) and their respective values as depicted in Table 1.

In the general case, the model will contain these 5 parameters for each application VM, and potentially additional values for the machine ID. It may also contain software-related parameters that capture application-specific load.

Table 2 presents the load combinations observed during testing of our application. The aim of CT is to design test cases that will achieve complete t-way coverage of the model combinations, where t is a user input. The most common application of CT is for pairwise testing, where $t = 2$. That is, every pair of values for every pair of parameters is covered during testing. Obviously, in Cloud settings and while using a combinatorial model that captures workloads it is extremely hard to direct the designed test cases to the required workloads, hence many of the resource utilization combinations are not covered by the test cases (i.e. exercised together in the same test). For example, the pair (`VM1 CPU=High, VM2 Memory=High`) was not observed during testing of our telephony application. Similarly, the pair (`VM1 Net In=Low, VM1 Net Out=High`) is also not observed during testing. In addition, since no adaptation action was performed during testing, all observed VM1 workloads are on machine ID Gr_1 per the initial configuration. Similarly, all VM2 workloads are on machine ID Gr_2.

During production, all untested combinations such as the ones above that are observed at runtime via the application monitoring contribute to the coverage gap between tested and observed load, and to its associated risk. The larger the coverage gap is in the current configuration, the greater the need may be for an adaptation action to take place.

We take the coverage metrics a step further in order to also rank between possible adaptation actions. For example, when migration is considered, if it is performed using

Table 2: **Combinatorial load coverage at test phase for the telephony load model**

VM1 CPU	VM1 Memory	VM1 Net In	VM1 Net Out	VM1 Machine ID	VM2 CPU	VM2 Memory	VM2 Net In	VM2 Net Out	VM2 Machine ID
Medium	Medium	High	Low	Gr_1	Medium	Low	Medium	High	Gr_2
Low	High	Low	Medium	Gr_1	Low	High	High	Medium	Gr_2
High	Low	Medium	High	Gr_1	High	Medium	Low	Low	Gr_2
Medium	Low	High	High	Gr_1	Low	High	High	High	Gr_2
High	High	Low	Low	Gr_1	High	Low	High	High	Gr_2
Medium	Low	Low	Low	Gr_1	High	High	Medium	Low	Gr_2
High	High	High	High	Gr_1	Low	Low	Low	Medium	Gr_2

a new physical machine, then a new machine ID value is added to the relevant parameter in the model, and consequently also many new untested combinations involving the new machine ID. Even when migration is performed to an existing machine ID, keeping the model definition as is, it can still cause many untested combinations to be executed, since the VM in question may or may not have been previously run on this machine ID. Our approach can give a different rank to different migration actions on different target machines, due to a different number of untested combinations in the resulting configuration. Similarly, scale-out will change the model definition (adding 5 new parameters for the duplicate VM) and will introduce new untested combinations between the original VM and the duplicate one, and between the duplicate VM and other elements of the model.

As depicted on the right side of Figure 2, the tested load combinations will be fed into our risk assessment engine on top of our CT tool IBM FOCUS, together with the monitored workloads. Our output risk assessment is fed into CloudWave's adaptation engine which reaches a collaborative decision based on various inputs on whether to perform adaptation. When adaptation is triggered, the adaptation engine sends a list of potential adaptation actions which we rank based on the coverage risk considerations described above. In Section 4 we describe in detail how we compute our metrics for adaptation triggering and action ranking.

4. THE COVERAGE-BASED APPROACH FOR CLOUD ADAPTATION

We present our two novel coverage-based metrics for risk assessment, one for assessing the risk associated with the current cloud configuration, and the other for evaluating the risk associated with each adaptation action. In the first mode, our engine periodically outputs a *Risk* metric based on the gap between the workloads combinations monitored during testing and those monitored in production. In the second mode, our engine prioritizes potential adaptation actions given by the central adaptation engine. In both modes, the workloads are modeled in a similar manner to that described in Section 3 (the same 5 parameters per VM). We use IBM FOCUS to compute pairwise coverage during testing and during production as will be described in the following. However our approach can be easily extended to any t-way coverage. As explained in Section 2, we achieve efficiency by using a symbolic BDD-based computation.

4.1 Triggering Adaptation Based on Coverage

In this mode, the input to our engine consists of the combinatorial model describing the existing VMs and their load parameters, the baseline workloads observed during testing, and live monitoring data representing the production time metrics relating to the parameters in the test model.

The output *Risk* metric is in the range of 0-1, representing the fraction of untested value pairs observed in production out of all untested pairs in the model. It is sent via Ceilometer to the CloudWave central adaptation engine. Upon initialization, our engine uses IBM FOCUS to calculate the pairwise coverage of the tests, i.e., the set of pairs of parameter values of the workloads monitored during testing. We denote this set by $Cov(Tests)$. The set $Valid(Model)$ denotes the set of all valid pairs of parameter values described by the model. Based on these two, the set of valid pairs in the model that were not observed during testing, *unseen_combs* is defined as:

$$\{(X,Y)|(X,Y) \in Valid(Model) \wedge (X,Y) \notin Cov(Tests)\}$$

The set $Valid(Model)$ is represented by a set of BDDs, each representing all valid parameters values for a certain pair of parameters. The size of this set is therefore calculated by summing the number of satisfying assignments to these BDDs. The set $Cov(Tests)$ is also represented by a set of BDDs, each representing all parameters values appearing in the tests for a certain pair of parameters. Again to calculate the size of this set we sum the number of satisfying assignments to each of these BDDs. Next, the size of the set of pairs uncovered by the tests, $|unseen\_combs|$ is given by

$$|Valid(Model)| - |Cov(Tests)|$$

unseen_combs is represented as a set of *coverage holes*. *Holes* are combinations of parameter values which are not covered by the given tests. Since we use pairwise coverage requirements, the computed holes will be either of size 1, i.e., single parameter values that were not tested, or of size 2, i.e., pairs of values that were not tested together, but each of the two values separately was tested. The coverage holes are represented as a set of BDDs: BDDs representing holes of size 1 for each parameter for which such holes exist, and BDDs representing holes of size 2 for each pair of parameters for which there are coverage holes in the given tests.

During production, our engine performs periodic calculations using the workloads monitored over a sliding time window (e.g., the past hour). Newly observed load combinations are considered risky only for this window of time. If it passes without the system exhibiting any problematic behavior, we assume that these combinations are safe and consider them as tested by adding them to the set of covered combinations $Cov(Tests)$. As a result, the coverage holes and $|unseen\_combs|$ are recalculated as described above.

The pairwise coverage of the workloads monitored in production during the current window of time $Cov(ProdWindow)$ is caluclated, and the current risk is given by:

$$risk_{metric} = \frac{|Cov(ProdWindow) \cap unseen\_combs|}{|unseen\_combs|}$$

The set $Cov(ProdWindow)$ is again represented by a set of BDDS, each representing all parameter value pairs appearing in the monitoring data over the production window for a certain pair of parameters. For each such BDD, $pairBDD$, we calculate the BDD $holeBDD$ representing the value pairs that are considered untested by constructing it from the size 1 and size 2 relevant BDDs in $unseen\_combs$. Finally we calculate a BDD $riskyPairCombinationsBDD$ by conjuncting $pairBDD$ and $holeBDD$. The resulting BDD represents the pairs of parameter values which were observed during the recent production window but had not been covered neither by tests nor during production prior to the recent time window. We sum the number of satisfying assignments for $riskyPairCombinationsBDD$ of each parameter pair to obtain the numerator in the formula above.

4.2 Ranking Adaptations Based on Coverage

In this mode, our engine is used to prioritize potential adaptation actions that have been given by the central adaptation engine. The prioritization is based on a calculation of the untested pairs that will be added due to each candidate adaptation action. We estimate for each possible adaptation action, what further combinations it would add to the space of load combinations not yet tested. Based on these measurements, we compute the overall risk associated with each adaptation action. This prioritization is derived from the ratio between the estimated number of combinations that will be added to the space of load combinations as a result of the adaptation action, and the total number of combinations that were not yet tested.

When invoked, our engine calculates the risk of each proposed adaptation action and then orders the proposed actions from the least risky to the most risky action.

- For migrating a certain VM vmx to destination target physical machine trg_pid, $risk_{migration}(vmx, trg\_pid)$ is given by:

$$\frac{|\{(X,Y)|Y_{vmx\_Machine\_id=trg\_pid} \wedge (X,Y) \in unseen\_combs\}|}{|unseen\_combs|}$$

  This means that all newly observed pairs of parameter values, one of which is trg_pid, are counted as risky since they indicate untested behavior. The computation above is again achieved symbolically as explained for the risk metric, while using a BDD constraining $vmx\_Machine\_id$ to trg_pid to consider the specific value pairs involving the migration action.

  If trg_pid has not been defined before as a potential machine for vmx, then as a result of this migration action, the model will be modified by adding trg_pid as a value for $vmx\_Machine\_id$, increasing the space of untested parameter value pairs $unseen\_combs$. This set will be recalculated to reflect this change, adding all the pairs for which one of the parameters is $vmx\_Machine\_id$ and its value is trg_pid:

$$unseen\_combs = unseen\_combs \cup \{(X,Y)|Y_{vmx\_Machine\_id=trg\_pid}\}$$

- For a scale-out action of a VM vmx:

  This action will add a new VM vmx_clone, and consequently new parameters to the model for vmx_clone. This naturally adds new pairs to the space of untested pairs of parameter values. We compute the number of these pairs as follows.

  - Pairs containing one vmx parameter and one vmx_clone parameter. To calculate the number of these pairs we simulate them on the current model by counting value pairs where both parameters belong to vmx parameters. We denote the result of this count by n_1.
  - Untested pairs of parameter values in the existing model, where at least one is a vmx parameter. These pairs will simulate in the new model pairs of parameters where at least one is a vmx_clone parameter. The number of such pairs, denoted by n_2, is calculated as before by iterating over the coverage hole BDDs. Note that we do not count tested pairs of which one is a vmx parameter, since we assume that vmx_clone will interact with other components similarly to the way vmx interacts with them, hence these pairs are considered tested also for vmx_clone.
  - Finally the risk for this action is given by:

$$risk_{scale-out}(vmx) = \frac{n_1 + n_2}{n_1 + n_2 + |unseen\_combs|}$$

5. PRELIMINARY RESULTS

We report on a preliminary feasibility assessment of our approach that we performed on the CloudWave infrastructure [5] on top of a commercial telephony application provided by one of the project partners. A more thorough evaluation is left for future work.

The telephony application consists of many components such as the SIP server, the media server, registration, authentication, authorization, and other components which may reside on separate VMs. The setup described in Figure 2 shows the most relevant component to our experiment; the SIP server VM. By the nature of the application, this component is likely to require a change in resources as the load of the application increases or decreases. The telephony application used may consist of hundreds of VMs running about 50 types of microservices; the setup we used began with 3 VMs plus a couple containing 'users' for load generation; additional VMs were added as load increased.

In the CloudWave setup each component received its own VM. The measured parameters were CPU, memory and network usage (in/out separately). The values were taken as the average during one minute, abstracted into low/med/high ranges, as described in the example in Section 3.

Since we have two different usages for the coverage-based risk engine, adaptation triggering, and ranking potential adaptation actions, we assess them separately.

5.1 Coverage-Based Adaptation Triggering

We begun the experiment by running existing tests for the telephony application, collecting and aggregating the pairs of parameter values covered by those tests. These pairs define the **Coverage by Tests** box in Figure 2.

We then ran the application using a call workload that changes over time. IBM FOCUS was used to generate a risk data-point every minute, based on the real-time measurements and on the historical test data. Figure 3 shows how

the risk changes over time. Initially it increases as new combinations, unseen during testing, are revealed. In our particular case, these combinations exist because testing was done under a relatively low load, while during deployment the load was much higher, pushing the system into situations not observed before.

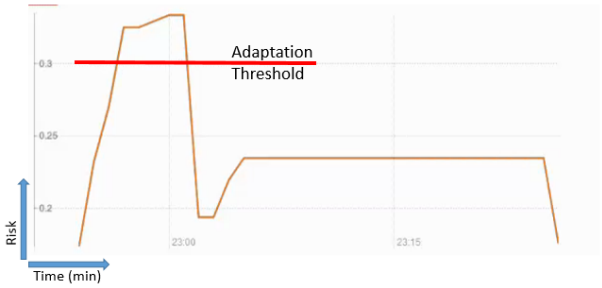

Figure 3: **Coverage-based risk over time**

Figure 3 further shows the risk fluctuating during the run of the application. The risk element introduced at the beginning of the run fades with time - as parameter combinations that caused the increase in risk exit the *time window* and are marked as tested. The risk increases once again as the load on the server peaks, introducing yet more unseen resource utilization. Finally, with time, as no new combinations are viewed, the risk decreases once again.

A CloudWave adaptation could have been triggered by a threshold on the coverage risk, for example when it crossed the 0.3 threshold, i.e. 30% of the untested value pairs of the model are seen at runtime, as depicted in Figure 3.

5.2 Coverage-Based Adaptation Ranking

As explained in the previous sections, coverage-based risk is one of the engines which the CloudWave adaptation component uses for deciding which change to perform on the cloud (if any) when a problem flag is raised.

A demonstration of the adaptation engine was devised which showed different outcomes based on the severity of the problem that triggered the adaptation. It consisted of the aforementioned telephony application, the focus being on two components: the SIP server and the Media server.

During this demo the adaptation engine decided to perform a migration of the SIP server. Our risk engine ranked the available physical machines based on the projected risk; one of them ranked higher (better) than the others since the SIP server resided on it during testing, as opposed to the others which were untested. Our ranking indicated a 4% addition of untested value pairs, as opposed to close to 10% using other physical machines. Following aggregation of our suggestion with the result of the other sub-engines, the migration was done to the machine suggested by our sub-engine. As a result, the load on the system was reduced with minimum increase in risk.

6. SUMMARY AND FUTURE WORK

In this work we presented a novel approach for cloud adaptation that is based on combinatorial coverage metrics, adopted from testing. We describe an efficient symbolic implementation of our metric computations, and demonstrate the feasibility of our approach on a commercial telephony application running with CloudWave adaptation management. We note that while the work described here was scoped to adaptation purposes, our metrics can be used for other Cloud management operations as well.

In the future, we plan to evaluate our approach, including its ability to detect situations that require adaptation which are undetected by traditional methods, and its success rate in differentiating and recommending adaptation actions that lead to more stable configurations. One of the challenges involved in such an evaluation is the need for a production system in a Cloud environment with managed adaptation.

We further plan to extend our metrics to ranking of scale-up operations, as well as normalizing the ranking of all potential adaptation actions. Finally, we plan to incorporate application-specific information into our load model and evaluate its contribution to the overall risk assessment.

Acknowledgements. We thank Orna Raz for her contribution to the ideas presented in this paper.

7. REFERENCES

[1] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Trans. on Comp.*, 1986.

[2] K. Burroughs, A. Jain, and R. Erickson. Improved quality of protocol testing through techniques of experimental design. In *SUPERCOMM/ICC*, 1994.

[3] Ceilometer. https://wiki.openstack.org/ceilometer.

[4] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[5] Cloudwave: Agile service engineering for the future internet. http://cloudwave-fp7.eu/.

[6] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-Based Testing in Practice. In *ICSE*, 1999.

[7] G. Galante and L. C. E. d. Bona. A survey on cloud computing elasticity. In *UCC*, 2012.

[8] IBM Functional Coverage Unified Solution (IBM FOCUS). http://researcher.watson.ibm.com/researcher/view_group.php?id=1871.

[9] D. R. Kuhn, R. N. Kacker, and Y. Lei. *Introduction to Combinatorial Testing*. Chapman & Hall/CRC, 2013.

[10] D. R. Kuhn, D. R. Wallace, and A. M. Gallo. Software Fault Interactions and Implications for Software Testing. *IEEE Trans. on Softw. Eng.*, 30, 2004.

[11] S. Malik, A. Wang, R. Brayton, and A. Sangiovanni-Vincentelli. Logic Verification Using Binary Decision Diagrams in a Logic Synthesis Environment. In *ICCAD*, 1988.

[12] S. Minato. Graph-Based Representations of Discrete Functions. *Representation of Discrete Functions*, 1996.

[13] C. Nie and H. Leung. A Survey of Combinatorial Testing. *ACM Comput. Surv.*, 2011.

[14] I. Segall, R. Tzoref-Brill, and E. Farchi. Using Binary Decision Diagrams for Combinatorial Test Design. In *ISSTA*, 2011.

[15] K. Tai and Y. Lie. A Test Generation Strategy for Pairwise Testing. *IEEE Trans. on Softw. Eng.*, 2002.

[16] P. Wojciak and R. Tzoref-Brill. System Level Combinatorial Testing in Practice – The Concurrent Maintenance Case Study. In *ICST*, 2014.

[17] Q. Zhang, L. Cheng, and R. Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 2010.