Evaluating Reconfiguration Impact in Self-adaptive Systems

An Approach based on Combinatorial Interaction Testing

Sagar Sen
Certus Software V&V Centre
Simula Research Laboratory
sagar@simula.no

Stefano Di Alesio
Certus Software V&V Centre
Simula Research Laboratory
stefano@simula.no

Dusica Marijan
Certus Software V&V Centre
Simula Research Laboratory
dusica@simula.no

Arnab Sarkar
Certus Software V&V Centre
Simula Research Laboratory
arnab@simula.no

***Abstract*—Self-adaptive software adapts its behavior to the operational context via automatic run-time reconfiguration of software components. Particular reconfigurations may negatively affect the system Quality of Service (QoS), and therefore their impact over the system performance needs to be thoroughly evaluated. In this paper, we present an approach, based on Combinatorial Interaction Testing (CIT), that generates a sequence of configurations aimed at evaluating the extent to which reconfigurations affect the system QoS. Specifically, we transform a Classification Tree Models (CTM) of the configurations domain to a Constraint Satisfaction Problem (CSP) in ALLOY, whose solution is a sequence of reconfigurations achieving T-wise coverage between system features, and R-wise coverage between configurations in the sequence. The resolution of the CSP is performed by an incremental growth algorithm that divides the generation of the sequence into sub-problems, and merges the results into a final set of test configurations. Preliminary validation in a self-adaptive vision system shows that our methodology effectively identifies critical reconfigurations exhibiting a high variation in QoS. This result encourages the use of CIT as a strategy to evaluate the performance of self-adaptive systems.**

I. Introduction and Related Work

Self-adaptive systems modify their behavior at runtime due to contextual changes from the environment, and feedback from internal components. These systems often use *reconfiguration* (or *adaptation*) mechanisms in order to adapt to the environment without the need of user intervention [1]. Consider for instance a self-adaptive vision system for object detection that automatically identifies unknown objects in a video sequence. The system autonomously adapts its behavior upon changes in the environmental context, such as decrease in light intensity [2]. Indeed, when low-light conditions are detected, the system *reconfigures* to increase the gamma and select night-vision object detection features [3]. Self-adaptive systems are usually designed with hundreds of configurations to cope with a large variety of possible environmental conditions. Furthermore, the unpredictability of the operating environment entails that a large number of *reconfigurations*, i.e., adaptations between configurations, can take place at runtime. Depending on the system design, particular reconfigurations may put a severe strain on computational resources, affecting the system Quality of Service (QoS). These *critical* reconfigurations need to be thoughtfully investigated in order to ensure that the system meets its expected performance at runtime. However, in complex systems, testing every possible reconfiguration is practically infeasible due to the large number of configurations and potential adaptations between them.

Related Work. Ensuring that self-adaptive systems meet their expected QoS levels at runtime is one of the biggest challenges when designing such systems [4]. For this reason, most established approaches for designing self-adaptive systems focus on quality assurance, considering adaptation properties such as throughput, setting time, and accuracy [5]. Even though more and more approaches also include guidelines for testing, verification and validation are among the least focused phases while engineering self-adaptive software. This is because it is hard to throughly explore a large domain of configurations and adaptations between them, which lead to several execution paths at runtime [6]. In particular, traditional verification and validation techniques such as testing, model checking, static analysis, and program synthesis, have all been proposed for assessing self-adaptive software quality. However, the main challenge of existing approaches is their integration into the self-adaptation life cycle, which has to cope with the unpredictability of reconfigurations at runtime [7]. In the domain of software product lines, Feature Modeling (FM) [8] and Combinatorial Interaction Testing (CIT) [9] have successfully addressed the problem of sampling the configuration space and hence generate effective test configurations in large systems [10]. This is because FM and CIT are based on the concept of *test coverage*, which ensures a thoughtful exploration of the system behavior. Therefore, the key idea behind our work is to apply the principles of FM and CIT in the domain of self-adaptive systems, in order to adequately cover the space of reconfigurations that at runtime define adaptive behavior.

Contribution of the Paper. In this paper, we present an approach, based on FM and CIT, to evaluate the reconfigurations impact over a self-adaptive system QoS. Our approach takes as input a Classification Tree Model (CTM) [11] that captures the domain of the system configurations. CTMs are modeling formalisms widely used in FM, which structure the system features in a hierarchical tree by modeling abstract components as *classifications*, and concrete implementations as *classes*. The approach generates a sequence of (re)configurations achieving both (1) T-wise coverage of interactions between components, and (2) R-wise coverage of interactions between reconfigurations. This is because, according to the principles of CIT, covering interactions between components and reconfigurations allows us to thoroughly exercise the system under a variety of

conditions. Our methodology consists of three steps.

1) First, we derive the set of system configurations by transforming the input CTM to a Constraint Satisfaction Problem (CSP) in ALLOY [12], a declarative specification language for expressing structural constraints in software systems. Each solution of the CSP is a *configuration* of the self-adaptive system, i.e., a set of concrete classes implementing the system abstract components. However, the total number of configurations is often too large to be thoroughly tested, and therefore a strategy is needed to select a small, yet representative, set of configurations.
2) Therefore, we generate a set of configurations that covers all the T-wise interactions between components and classes of the self-adaptive system. We achieve this by solving the CSP in conjunction with predicates enforcing the T-wise coverage criteria between the classes of the CTM. Note that, for complex systems, such CSP may become too large to be solved within convenient time. Therefore, we augmented the CSP solving process with a *divide-and-merge* strategy inspired by previous work on generating configurations that cover T-wise interactions for feature models [13]. The strategy divides the satisfaction of large sets of predicates representing T-wise interactions between classes and components into smaller problems, merges the partial solutions of the subproblems, and generates a final set of test configurations.
3) Finally, this set of is used to generate a *test sequence* of reconfigurations covering all the ordered R-wise interactions between configurations. The test sequence can then be executed in the system in order to identify the reconfigurations yielding the highest variation in QoS.

The approach is supported by VINYASA[1], a tool that takes as input a CTM specified in the test design and modeling tool TESTONA[2], and outputs a test sequence of reconfigurations.

II. METHODOLOGY

In this Section, we present a methodology to generate test sequences of reconfigurations in self-adaptive systems starting from a Classification Tree Model (CTM) of the system variability (Section II-A). Our methodology consists of the three steps shown in Figure 1a: (1) transformation of a CTM representing the target system variability to a Constraint Satisfaction Problem (CSP) specified in ALLOY (Section II-B), (2) generation of a set of test configurations covering all the T-wise interactions between classes and compositions in a CTM (Section II-C), (3) generation of a test sequence that covers all the R-wise interactions between each of the configurations generated in the previous step (Section II-D). The methodology is supported by VINYASA, a tool implemented in Java which takes as input a CTM specified in the commercial tool TESTONA, previously known as CTE XL. TESTONA is a proven industry-strength tool for modeling classification trees, which includes numerous add-ons for integration with external testing suites. Therefore, it represents a natural choice for any approach aiming for industrial use [11]. The source code of VINYASA is available on-line[3].

[1]VINYASA is a Sanskrit word that refers to the creation of a sequence. «Nyasa» means *to place*, and «vi» denotes *in a special way*.

[2]http://www.testona.net

[3]http://sites.google.com/a/simula.no/vinyasa/

A. Modeling Variability through a Classification Tree Model

We first introduce variability modeling through Classification Tree Models using as example TEKIO, a self-adaptive vision system for *scene understanding* that includes features for objects detection, identification, and tracking [3].

Compositions, Classifications, and Classes. We model the variability in a self-adaptive system using the CTM formalism, which hierarchically structures the system features in a tree. The root composition of the tree represents the target system, and the children, referred to as *compositions*, represent its components. Non-decomposable components are defined as *classifications*, and are structured in alternative *classes* realizing the concrete implementation of classifications. The CTM of TEKIO is shown in the upper part of Figure 1b. The variability in TEKIO is in the selection of its video processing components, namely *ImageAcquisition*, *CameraConfiguration*, *ImageSegmentation*, and *ObjectDetection*. These components are modeled as classifications, i.e., as *abstract* components whose behavior is realized by a set of alternative classes. The classes represent in turn *concrete* components with fixed parameter settings. For example, the behavior of the *ObjectDetection* abstract component can be alternatively realized by the *HAAR* and *LocalBinaryPattern* (*LBP*) concrete components.

Configurations. Particular selections of classes give rise to system configurations. Specifically, a system *configuration* is a set C of classes that satisfies the following two constraints: (c1) each atomic component in the CTM is realized by a class in C, and (c2) each class in C realizes the behavior of a different atomic component. We refer to these constraints as *implicit* in the CTM, because they directly follow from the semantics of classes and classifications. Ten configurations are shown in a grid at the bottom of the CTM in Figure 1b, where the classes selected are shown by solid gray dots. For example, *Configuration_14* represents a configuration where *ImageAcquisition*, *CameraConfiguration*, *ImageSegmentation*, and *ObjectDetection* are realized by *VideoFileStream*, *Default-Light*, *MeanShiftSegmentation*, and *LBP*, respectively.

Explicit Constraints between Classes in a Configuration. Note that the definition we provide entails that every set of classes satisfying the constraints (c1) and (c2) is a *valid* configuration. However, CTM also support the definition of *explicit* cross-tree constraints specifying rules on the selection of classes. For instance, a constraint may state that *Default-Light* and *LBP* cannot be selected at the same time, and hence a configuration containing both of them is said to be *invalid*. Even though in TEKIO there are no such cross-tree constraints, we consider them in our analysis as they are commonplace in self-adaptive systems.

Interactions of Classes in a Configuration. The presence or absence of classes in a configuration gives rise to class *interactions*. Specifically, a tuple of cardinality T specifying presence or absence of T classes in a configuration C represents a *T-wise interaction* in C. For example, in *Configuration_14*, the pairwise interaction between the classes *Smoothing* and *LBP* specifies that the former is absent, while the latter is present.

Sequence of Reconfigurations. As discussed before, self-adaptive systems adapt their behavior to the environment by dynamically switching from a configuration to another. This means that, at runtime, the system runs a *sequence of reconfigurations*, i.e., an ordered list of transitions between configurations. For instance, the folder *Test Group* at the

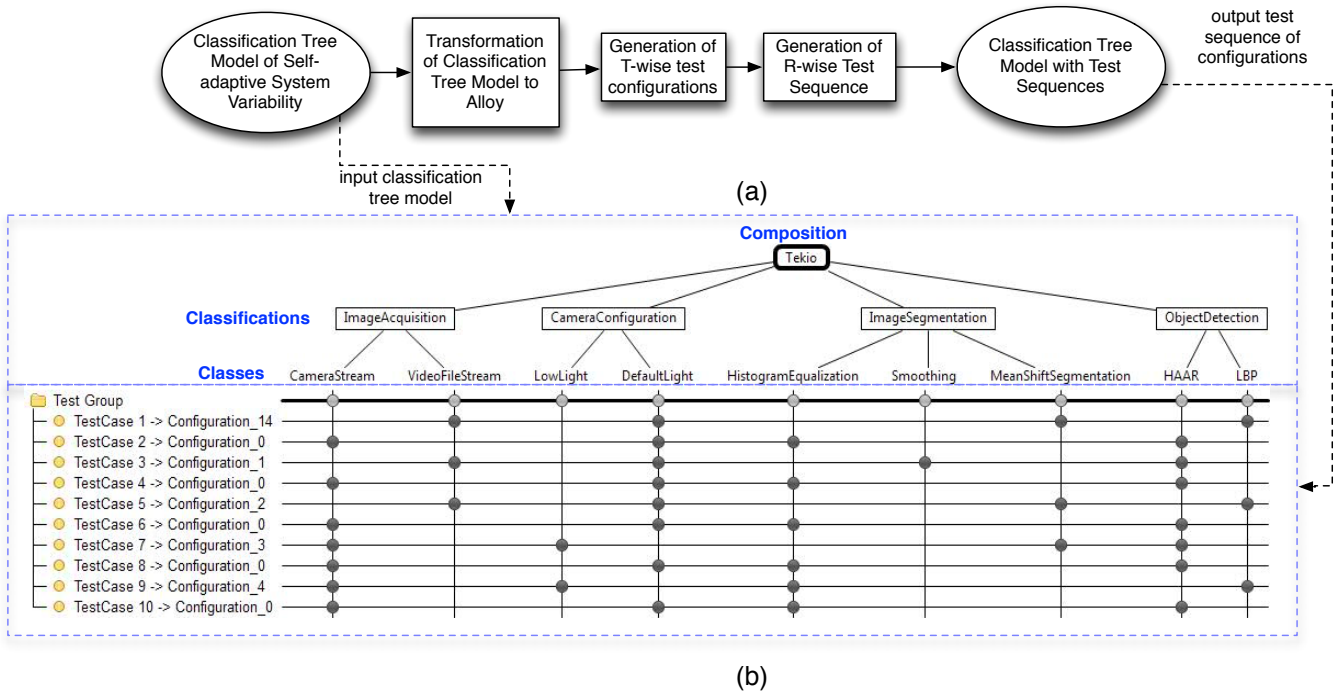

Fig. 1: Our methodology for generating a test sequence to evaluate the reconfiguration impact in self-adaptive systems

bottom of Figure 1b represents a sequence of 10 configurations and 9 reconfigurations between them. Specifically, *Test Group* starts with the initial configuration *Configuration_14*, and initially reconfigures to *Configuration_0*. Then, it reconfigures from *Configuration_0* to *Configuration_1*, and so on. Note that, during operation, a self-adaptive system may repeat configurations in a sequence.

B. Transforming a Classification Tree Model to a Constraint Satisfaction Problem in ALLOY

In order to generate valid test configurations from a CTM, we capture the implicit constraints between compositions and classes in a Constraint Satisfaction Problem (CSP) implemented in ALLOY. This is done in three steps: (1) generating an ALLOY model representing compositions, classifications, and classes of a CTM (Section II-B1), (2) generating predicates representing T-wise interactions between classes and compositions (Section II-B2), and (3) detecting the T-wise interaction predicates representing valid configurations (Section II-B3).

```
sig Tekio{}
sig ImageAcquisition{}
sig CameraConfiguration{}
sig ImageSegmentation{}
sig ObjectDetection{}
sig CameraStream{}
sig VideoFileStream{}
sig LowLight{}
sig DefaultLight{}
sig HistogramEqualization{}
sig Smoothing{}
sig MeanShiftSegmentation{}
sig HAAR{}
sig LBP{}
```

Listing 1: Signatures from the Classification Tree Model

```
sig Configuration {
  f1: one Tekio,
  f2: one ImageAcquisition,
  f3: one CameraConfiguration,
  f4: one ImagesSegmentation,
  f5: one ObjectDetection,
  f6: lone CameraStream
  f7: lone VideoFileStream,
  f8: lone LowLight,
  f9: lone NormalLight,
  f10: lone HistogramEqualization,
  f11: lone Smoothing,
  f12: lone MeanShiftSegmentation,
  f13: lone HAAR,
  f14: lone LBP }

one sig ConfigurationSet {
  configurations : set Configuration }
```

Listing 2: *Configuration* and *ConfigurationSet* signatures

*1) **Generating an ALLOY Model from a Classification Tree.*** Starting from a CTM CT, we generate an ALLOY model A_{CT} capturing classes, compositions, and constraints between them in ALLOY, a modeling language based on relational first-order logic. An ALLOY model consists of signatures, fields, facts, and predicates. Each *signature* denotes a set of non-decomposable entities called *atoms*, represented as signature *fields*. Specifically, each field belongs to a signature, and represents a relation between two or more signatures. Such relations are interpreted as sets of tuples of atoms. Note that each atom represents a variable, while a complete assignment of values to atoms is said to be an *instance*, i.e., a solution for the model. *Facts* represent constraints between signatures and atoms that must hold at any time, while *predicates* represent parametrized constraints which can be included in other predicates or facts. Consider the TEKIO CTM (top of Figure 1b), which has one composition, four classifications, and nine classes. These nodes are denoted in A_{CT} as 14 ALLOY signatures (Listing 1). A_{CT} also declares the two signatures *Configuration* and *ConfigurationSet* shown in Listing 2. *Configuration* contains 14 fields $f_1, f_2, \ldots, f_{14}$ representing the signatures of system compositions, classifications, and classes defined in Listing 1. Note that the *Configuration* fields representing compositions and classifications, i.e., $f_1 \ldots f_5$, must be present in all the signature instances, and hence are qualified by the keyword *one*. On the other hand, the fields $f_6 \ldots f_{14}$ that represent classes can be either present or absent in a signature instance, and hence are qualified by the keyword *lone*. This is because configurations always include at least one class realizing each of the classifications, but a class may be either present or absent in a configuration. *ConfigurationSet* specifies that a sequence of configurations is a single set of *Configuration* signatures. A_{CT} specifies mutual exclusion between classes of the same classification as ALLOY facts, as shown in Listing 3.

Each fact expresses a constraint related to a classification, specifying that, for each configuration, the number of classes realizing the classification is equal to one. For example, the *ObjectDetection* component is realized by either *HAAR* or *LBP*, entailing that, for each configuration, the sum of cardinalities of the fields f_{13} and f_{14} must be equal to one. Note that implicit constraints in the CTM are captured by the *one* and *lone* keywords in the *Configuration* signature (Listing 2), and by the four facts in Listing 3. Cross-tree explicit constraints are specified as additional facts.

```
fact Fact_ImageAcquisition { all c:Configuration | #c.f6 + #c.f7 = 1 }
fact Fact_CameraConfiguration { all c:Configuration | #c.f8 + #c.f9 = 1 }
fact Fact_ImageSegmentation { all c:Configuration | #c.f10 + #c.f11 + #c.f12 = 1 }
fact Fact_ObjectDetection { all c:Configuration | #c.f13 + #c.f14 = 1 }
```

Listing 3: Facts specifying the mutual exclusion of classes realizing the behavior of the same atomic component

2) ***Generating Interaction Predicates Achieving T-wise Coverage***. In our approach, we generate a sequence of test reconfigurations that achieves T-wise coverage of the interactions between classes and components represented by the atoms $f_1, \ldots, f_{14}$ of the *Configuration* signature. We compute the set I of predicates that enumerate the T-wise interactions between fields in A_{CT}. In ALLOY, a predicate is a constraint that can solved by specifying a *scope*, i.e., a maximum number of atoms in each signature. The scope determines the size of the Constraint Satisfaction Problem, which is then transformed by the ALLOY Analyzer to Conjunctive Normal Form (CNF) formulae. Finally, the CNF formulae are solved by a satisfiability solver such as MiniSAT [14] in order to obtain an instance. Note that, if an ALLOY model consists of N elements related to the nodes of the CTM, the number of predicates in I is equal to the combinations of compositions and classes taken T at a time: $\mid I \mid = 2^T \cdot \binom{N}{T}$. For example, a 3-wise interaction can specify that each *Configuration* instance has to be such that f_8 has cardinality (#) zero, i.e., is absent, while f_{10} and f_{13} have cardinality one, i.e., are present (Listing 4). The qualifier *some* specifies that at least one *Configuration* has to be such that *LowLight* (f_8) is not selected, and *HistogramEqualization* (f_{10}) and *HAAR* (f_{13}) are selected.

```
pred t { some c: Configuration | #c.f8 = 0 and #c.f10 = 1 and #c.f13 = 1 }
```

Listing 4: Example of a 3-wise Interaction Predicate

3) ***Detecting Valid Interaction Predicates***. Note that not every predicate $p \in I$ is *valid*, i.e., it is such that the conjunction $A_{CT} \cap p$ has a solution. For example, an interaction specifying the presence of f_{13} and f_{14} is not valid, because *HAAR* and *LBP* both realize the behavior of the *ObjectDetection*. Hence, we identify the valid predicates in I. For each $p \in I$, we solve the model $A_{CT} \cap p$ for a *scope* equal to 1, in order to obtain *exactly one* test configuration for which the interaction defined by p holds. This step generates the subset $V \subseteq I$ of valid predicates extracted from I, i.e., predicates p for which $A_{CT} \cap p$ has at least one instance.

C. Generating Configs. with T-wise Interaction Coverage

After generating V, we generate a set of configurations satisfying all predicates in V. This step consists of solving the conjunction $\bigcap_{p \in V} A_{CT} \cap p$ to find an instance for *ConfigurationSet*, i.e., a set of Configuration instances that satisfy all the predicates in V. In practice, we aim at finding a *ConfigurationSet* with the least number of configurations. This is because a small set of configurations allows to have a short sequence that can be quickly executed, rendering the evaluation of the system QoS less time consuming. Note that, the larger the size of V, the higher the chance that V contains mutually inconsistent predicates, which hence must be present in different *Configuration* signatures of *ConfigurationSet*. However, our practical experience suggests that, for large CTMs, SAT solvers cannot solve the conjunction of all predicates in V within convenient time. This is because ALLOY transforms large conjunctions of predicates in several thousands of CNF formulae, which are hard to solve all at once. Therefore, we propose a *divide-and-compose* strategy that breaks down the resolution of a large conjunction of predicates into smaller CSPs. This strategy is implemented by an algorithm named *Incremental Growth*, which is adapted from previous work on test case generation approach for software product lines [13].

Algorithm 1 incGrow($A_F, V, mnScp, mxScp$)

```
Pool ← {}
repeat
    CT ← {}
    repeat
        CT.add(V.selectRandom), {result, CT.solution} ← solve(A_F, CT, mnSc, mxSc)
        if result == False then
            CT.remove(predicate), V.add(predicate)
        else
            V.remove(predicate)
    until result == False
    Pool.add(CT)
until V.isEmpty
repeat
    mergedSet = mergedSet.add(Pool.iterator().next().CT.solution)
until Pool.iterator().hasNext()
return mergedSet
```

The algorithm, shown in Algorithm 1, takes as input an ALLOY model A_F, a set of valid predicates V, a minimum scope $mnScp$, and maximum scope $mxScp$. Recall that, in ALLOY, the scope is the maximum number of atoms in each signature. When solving a model, a small scope might lead to no solution, while a large scope might lead to a *ConfigurationSet* containing too many *Configuration* instances. Since we are interested in finding the minimal set configurations satisfying all the predicates in V, the algorithm instructs ALLOY to find solutions with $mnScp$ first, progressively increasing the scope up to $mxScp$ when no solution can be found. Usually, $mnScp$ is set to 1, while $mxScp$ is set to the highest scope for which the SAT solver used can terminate within convenient time. The algorithm begins by initializing an empty $Pool$ of predicates whose conjunction can be solved within convenient time. Then, the algorithm selects a $predicate$ from V, adds it to the set CT, and generates a Configuration by solving $A_F \cap CT$ within the scope range given by $mnSc$ and $mxSc$. The algorithm continues to incrementally add predicates to CT until no solution can be found within the scope rage. In this case, the algorithm joins the set CT to the $Pool$, and repeats the process by creating an empty CT in order to find a solution for all the remaining predicates in V. In this way, the algorithm stores a set of conjunction of predicates in $Pool$, and finally merges the solutions for all the conjunctions in CT, returning a $mergedSet$ of configurations. By construction, the configurations in $mergedSet$ satisfy all predicates in V.

D. Generating a Sequence with R-wise Configs. Coverage

Finally, we generate a test sequence of reconfigurations achieving R-wise coverage within the configurations generated in the previous step. Doing so is important in order to ensure

that every sequence of R transitions between configurations is tested by the sequence. Indeed, the residual memory in a configuration can negatively affect the next configuration. For example, transitioning from *LowLight* to *DefaultLight* may require a re-computation of the initial parameters in *ImageSegmentation*, or a clean up of buffer variables that slow down the adaptation. Generating the test sequence with R-wise coverage consists of enumerating the permutations between the configurations generated in the previous step. Note that, for K configurations, the length of the test sequence is $\frac{K!}{(K-R)!}$.

III. Applicability and Preliminary Validation

We ran a preliminary validation of our methodology in TEKIO, the self-adaptive vision system introduced in Section II-A, in order to investigate the *critical* reconfigurations that have a negative impact over the system performance. In particular, we generated a test sequence achieving pairwise coverage of both feature interactions within configurations ($T = 2$), and configurations within the sequence ($R = 2$). Recall from Section II-B1 that the TEKIO CTM contains 14 nodes, i.e., compositions, classifications, and classes. From those, we generated the set I containing $2^2 \cdot \binom{14}{2} = 364$ predicates enumerating the T-wise interactions between features and classes, 100 of which were valid ($\mid V \mid = 100$). Starting from V, the *Incremental Growth* algorithm generated 12 configurations, which we used to finally generate a test sequence of $\frac{12!}{(12-2)!} = 132$ reconfigurations. We run this sequence of reconfigurations for 100 times, in order to analyze the adaptive behavior of TEKIO under a variety of different inputs coming from the system image acquisition source. Note that, during normal operation, the adaptation planner of TEKIO decides the sequence of reconfigurations based on input coming from the environment. Therefore, we instruct TEKIO to execute our fixed sequence, so that we are able to explore the behavior of the adaptation planner without having to generate inputs that trigger a large variety of reconfigurations. Furthermore, executing the sequence in 100 runs allows us to average out the effect that a particular contextual change in the environment might have had. We measured the variation in *frame rate* and *CPU usage* of TEKIO during each reconfiguration, as these are among the most important factors when assessing the QoS of vision systems [15]. In particular, we identified the critical reconfigurations in our sequence, i.e., the reconfigurations exhibiting high variation in frame rate and CPU usage. Experimental results showed that the critical reconfigurations in our sequence exhibited a variation up to twice as high as that of critical reconfigurations in randomly generated sequences. These preliminary results show that our approach effectively identifies reconfigurations with high QoS variation, and encourage a more thoughtful validation.

IV. Conclusion

Evaluating the impact reconfigurations have over self-adaptive systems performance is important in order to ensure such systems meet their expected Quality of Service (QoS) levels at runtime. However, evaluating the performance of self-adaptive systems is a task whose challenges have not been entirely addressed, mainly because of the unpredictability and the large domain of adaptive behavior. In this paper, we present a methodology to automatically generate test sequences of configurations that satisfy the combinatorial interaction coverage test adequacy criteria. In particular, we generate a set of test configurations covering all T-wise interactions between features of a self-adaptive system, and from this set a test sequence covering all the R-wise interactions between the configurations. Our methodology is supported by VINYASA, a tool that takes as input a Classification Tree Model of the domain of system configurations, and casts the generation of a test sequence of reconfigurations as Constraint Satisfaction Problem in ALLOY. The CSP is then solved using a scalable *Incremental Growth* algorithm that divides the generation of the sequence into subproblems. Preliminary validation in a self-adaptive vision system shows that our methodology effectively reveals critical reconfigurations that exhibit a high variation in QoS. This result encourages future work on the use of Combinatorial Interaction Testing for evaluating the impact reconfigurations have over the system performance.

References

[1] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, and A. Wolf, "An Architecture-Based Approach to Self-Adaptive Software," *Intelligent Systems and Their Applications, IEEE*, vol. 14, no. 3, pp. 54–62, 1999.

[2] L. M. Rocha, S. Moisan, J.-P. Rigault, and S. Sagar, "Girgit: A Dynamically Adaptive Vision System for Scene Understanding," in *International Conference on Computer Vision Systems*, J. L. Crowley, B. A. Draper, and M. Thonnat, Eds., vol. 6962, 2011.

[3] S. Hurtado, S. Sen, and R. Casallas, "Reusing legacy software in a self-adaptive middleware framework," in *Adaptive and Reflective Middleware on Proceedings of the International Workshop*, 2011.

[4] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009.

[5] B. Cheng, R. de Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, and Others, "Software engineering for self-adaptive systems: A research roadmap," *Software Engineering for Self-Adaptive Systems*, pp. 1–26, 2009.

[6] R. De Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, M. Litoiu, B. Schmerl, G. Tamura, N. M. Villegas, T. Vogel *et al.*, "Software engineering for self-adaptive systems: A second research roadmap," in *Software Engineering for Self-Adaptive Systems II*.

[7] N. M. Villegas, H. A. Müller, G. Tamura, L. Duchien, and R. Casallas, "A framework for evaluating quality-driven self-adaptive software systems," in *Proceedings of the 6th international symposium on Software engineering for adaptive and self-managing systems*.

[8] A. Hervieu, B. Baudry, and A. Gotlieb, "PACOGEN : Automatic Generation of Pairwise Test Configurations from Feature Models," in *Proc. of Int. Symp. on Soft. Reliability Engineering (ISSRE'11)*, 2011.

[9] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, p. 11, 2011.

[10] M. F. Johansen, O. Haugen, and F. Fleurey, "An algorithm for generating t-wise covering arrays from large feature models," in *Proceedings of the 16th International Software Product Line Conference - Volume 1*, ser. SPLC '12, 2012, pp. 46–55.

[11] E. Lehmann and J. Wegener, "Test case design by means of the cte xl," in *Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000)*, 2000, pp. 1–10.

[12] D. Jackson, *Software Abstractions: Logic, Language, and Analysis*. MIT Press, March 2006.

[13] 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 *International Conference on Software Testing*, 2010.

[14] Niklas Een and Niklas Sorensson, "MiniSat: A SAT Solver with Conflict-Clause Minimization, Poster," in *SAT 2005*, 2005.

[15] G. R. Bradski, "Real time face and object tracking as a component of a perceptual user interface," in *Applications of Computer Vision, 1998. WACV'98. Proceedings., Fourth IEEE Workshop on*.