Automated Generation of Initial Configurations for Testing Component Systems

Frédéric Dadeau, Jean-Philippe Gros, and Olga Kouchnarenko

Univ. Bourgogne Franche-Comté, CNRS, FEMTO-ST Institute,
15B avenue des Montboucons, 25030 Besançon, Cedex, France
`firstname.lastname@femto-st.fr`

Abstract. In the context of component-based systems, this paper presents the automated generation of initial states, from which an adaptive system starts to receive sequences of events that aim to provoke reconfigurations. For generating these states, also called configurations, we present a combinatorial algorithm supporting various architectural elements and relationships among them, while satisfying consistency constraints expressed by invariants. Moreover, this algorithm deals with the system-dependant instantiations of the primitive and composite components, parameters and relations, in order to produce meaningful structured configurations. While testing adaptation policies for component-based systems, this algorithm allows us to improve the capability of fulfilling coverage criteria by using different initial configurations. To illustrate the approach, the paper reports on experiments on a simulation with platoons of autonomous vehicles.

1 Introduction

Even if models of component-based systems are very heterogeneous, most of them consider software components either as black boxes, or as grey boxes if some of their inner features are visible, having fully-described interfaces. Systems' behaviour is then specified using components' definitions. In general, the system state, also called a configuration, is the specific definition of the elements that define what a system is composed of, while a reconfiguration can be seen as a transition from a configuration to another. In this context, adaptation rules or policies can be used to guide dynamic reconfigurations of component-based systems [6,18,10] by using either architectural constraints on configurations, or events, or temporal constraints over sequences of events and reconfigurations.

Overall, our goal is to validate that the adaptation policy rules are faithfully implemented by the system. In [7], the system execution has been validated w.r.t. the adaptation policy by checking that the reconfigurations that are triggered during the execution correspond to those authorized in the adaptation policy. In addition, [8] addresses the issue of validating that the system execution, which starts from a particular configuration, respects the utilities of the reconfiguration rules of the adaptation policy. It is easy to see that the testing process depends on initial configurations. For example, in the context of autonomous systems,

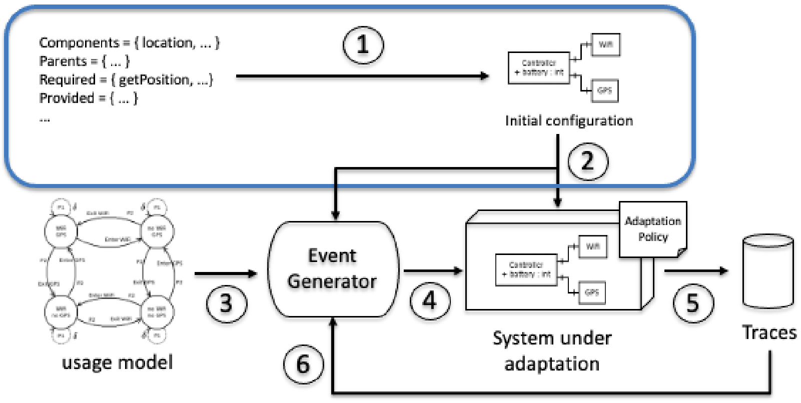

Fig. 1. The process of online test generation for adaptive systems

when the execution starts from a configuration with the full battery, only a few reconfigurations are expected and, consequently, the test cases to cover aimed behaviour with reconfigurations may be long enough. On the contrary, starting from a configuration where the battery level is low, provokes reconfigurations to save energy, and the test cases are expected to be shorter. So, to go further, the present paper presents the automated generation of initial configurations, from which the adaptive system starts to receive sequences of events that aim to provoke reconfigurations.

This process is summarized in Fig. 1. The adaptive system's architecture is described by a component-based model, from which initial configurations are automatically generated while instantiating the model (1). These initial configurations are necessary inputs to initialize testing (2) of the system, as the tests are executed starting from them, as well as the system itself. To take into account the environment in which the system is executed, usage models [25] for components are provided as inputs as well. Starting from initial configurations, test cases are composed of the events, which are extracted from components usage models (3). These events are sent by the test generator to the system (4), in an online testing manner: as the system's behavior depends on the environment, test outputs are observed on the reconfiguration trace (5), and analyzed by the test generator (6) to both guide the next event to be sent to the system under test (4), and verify that the system behaves as expected w.r.t. the various artifacts that are available, namely adaptation policies with temporal properties (not shown in Fig. 1). This last point has been described in [7], where as the present paper focuses on points (1) and (2).

In order to generate initial configurations, we first considered a random data generator. However, due to the complexity of the structures to be generated for component-based systems, such a random generator could possibly not terminate, or hardly converge to a relevant configuration, which is realistic in terms of architecture. Indeed, the relationships that are defined by the component model, regarding parenting relationships, delegations and bindings, define a constraint satisfaction problem (CSP), that cannot be effectively solved by a random process. While constraint solvers exist, such as CLPS [5], using them is problematic in our context. First issue is a large number of solutions computed that will be structurally similar, due to symmetries in the solution space. Second, solvers are usually meant to determine if a CSP has a solution, but a fine tuning of the solver is required to obtain some variety in the proposed solutions.

To overcome these issues, the contributions of this work are to propose a dedicated combinatorial algorithm that is used to enumerate all possible symmetry-free solutions of the CSP defined by the component model, in order to produce initial configurations. This algorithm integrates symmetry elimination patterns which reduce the combinations to be considered. While this algorithm can be used to perform bounded exhaustive testing as in [22], the resulting configuration set can be sampled to select a subset of configurations, that reduce the number of test data to consider. A sampling method, aiming to amplify the variety of configurations, is presented as a second contribution.

Outline. The paper is organized as follows. After a brief overview of the component-based systems under adaptation policies, some basic notions on coverage criteria for their testing are presented in Sect. 2. Section 3 presents the component-based model that is used to represent systems configurations. The configuration generation process, based on bounded exhaustive computation of the possible configurations is presented in Sect. 4. The algorithm is described along with optimisations that aim to reduce the combinatorial explosion, and data selection criteria that make it possible to sample the solutions to a small but significant subset. Section 5 reports on experimental results w.r.t. the research questions. Related works are presented in Sect. 6 before concluding in Sect. 7.

2 Background

On component-based systems under adaptation policies. In this paper, only the basic and generic concepts of component-based systems are considered to allow their application to various hierarchical models: components as entities of several types, required and provided interfaces as interaction points between components, bindings to link component interfaces. Components are either primitive components providing data or services, or composite compound components delegating their interfaces. Components can have some attributes used as configuration parameters. This section presents a running example of such a system, whereas Section 3 provides the reader with needed formal notions.

Example 1. Let us start with an example of a Vehicle Platooning Application (VPA for short) inspired from [4]. This complex system is composed of vehicles which are either in solo mode, or organized in some platoons, as displayed in Fig. 2. This figure also provides a component-based architecture corresponding to the displayed VPA situation. In VPA, each platoon is led by a leader vehicle. Any vehicle in solo mode can ask to join a platoon or decide to create a new platoon with another vehicle in solo mode. Each vehicle in a platoon can ask to quit it either because the vehicle reached his destination, or because it needs to refill its energy. The platoon leader may change either because another vehicle has more autonomy or a further destination. Some external events happen in the system environment, e.g., a new vehicle can arrive on the road, or a driver may decide to quit the platoon on his way to a new destination. These changes on system's architecture level are considered as dynamic reconfigurations.

For dynamic reconfigurations to occur only in suitable circumstance, adaptation rules indicate, for a given set of configurations, which reconfiguration operations can be triggered, with a utility level associated. Following [6], they are of the form **when** b **if** g **then utility of** *ope* **is** f. As introduced in [18], reconfiguration operations in adaptation policies are guarded by temporal logic properties that may either make use of propositional formulae over configurations, or involve sequences of events and/or reconfigurations.

when after *Join normal* until *Quit normal* **and** *VehicleId.battery* $<$ *33*
if *state* $=$ *leader* **then utility of** *PassRelay* **is** *high*

when after *Join normal* until *Quit normal* **and** *VehicleId.battery* $>$ *Leader.battery*
if *state* $=$ *platooned* **then utility of** *GetRelay* **is** *medium*

Example 2. Let us consider 2 adaptation rules involving the *PassRelay* and *GetRelay* reconfigurations. Intuitively, the above rules apply to all vehicles and are used to determine when it is possible to have a relay between the leader and another vehicle of the platoon. In the first case, the *PassRelay* reconfiguration of *high* utility can be triggered when the leader has not enough autonomy.In the second case, the *GetRelay* reconfiguration of medium utility may trigger when the autonomy of a vehicle is greater than the autonomy of the leader.

Notice that a reconfiguration is suggested with a utility value (e.g. from $Ft = \{$ high, medium, low$\}$). For the formal definition, the reader can refer to [7].

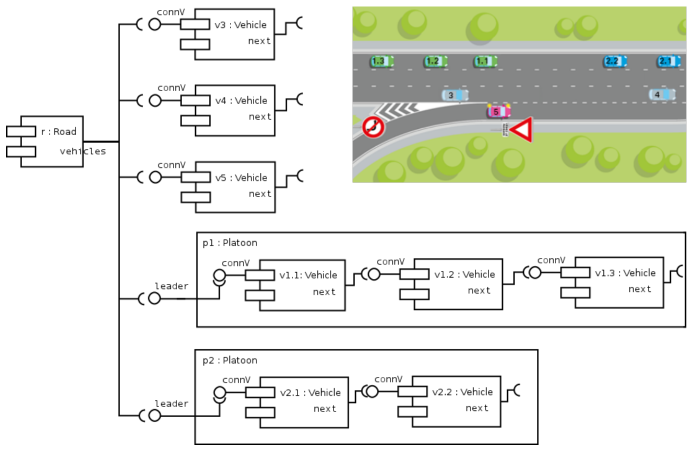

Fig. 2. Component architecture for the considered VPA (top-right side)

On coverage criteria for adaptation policies. In a previous work [7], we have proposed a test generation technique which aims to generate sequences of external events, from usage models of system's components, in order to exercise the reconfiguration rules described in the adaptation policy. The dedicated coverage criteria have been designed for adaptation rules with temporal patterns by exploiting coverage criteria for temporal patterns described in [23].

These coverage criteria can be used as a means to handle the input data, to evaluate a test suite, by measuring how much of the considered artifacts–e.g., temporal properties and adaptation rules with temporal properties–the test suite covers, and to decide when to stop testing. In [23], a temporal property is considered as covered by a test suite TS if each transition of the property test automaton is covered by at least one test case tc from TS. Having the same temporal patterns allows us to consider coverage criteria for adaptation rules and thus for adaptation policies. In [7], the adaptation rule is covered by a test case tc if the rule is eligible–there is a configuration that tc reaches, where b scope and g guard predicates are evaluated to true,–and *ope* is actually triggered from such a configuration. Coverage criteria for policies are obtained by lifting this notion to sets of rules.

As a consequence, the introduced coverage criteria for adaptation rules allow the user to evaluate if (i) the triggered reconfigurations were among eligible ones, in order to detect undesirable reconfigurations, and (ii) a generated test suite execution has triggered all the reconfiguration rules that were described in the adaptation policy, so as to detect specified but never triggered reconfigurations, even for long test cases generated.

3 Component-based Model

This section gives means for specifying component-based systems. Their architectural model is defined as a triplet $\langle Elem, Rel, Inst\rangle$, where $Elem$ is a set of the *component elements*, Rel describes the *architectural relationships* between these elements, and $Inst$ is an instantiation of $Elem$ and Rel in terms of actual components and relations.

Components. Components are entities that can be assembled to create an application. As usual, interfaces are used for interactions between components. A provided interface is an interface that the component realizes, whereas a required interface is an interface that the component needs to be able to run. Composite components may delegate their interfaces to inner components. Formally, $Elem = \{CTypes, IProvided, IRequired, Params, ITypes, PTypes, Contings\}$, where $CTypes$ is the non-empty set of components types, $IProvided$ (resp. $IRequired$) is the set of interfaces, which are provided (resp. required) by the components, $Params$ is the set of components' parameters, $ITypes$ (resp. $PTypes$) is a finite set of the interfaces (resp. parameters) types, $Contings$ is the set of contingencies that represent the cardinality of required interfaces (single or multiple connections, optional or mandatory).

Example 3 (Components of the VPA example). The architectural elements of the VPA configuration depicted in Fig. 2 are as follows:

$$\begin{array}{ll}
CTypes & = \{Road, Platoon, Vehicle\} \\
IProvided & = \{connV, leader\} \\
IRequired & = \{vehicles, next\} \\
Parameters & = \{battery, position, speed, goal\} \\
ITypes & = \{VInfo\} \\
PTypes & = \{int, float\} \\
Contings & = \{singleopt, singlemandatory, multiopt, multimandatory\}
\end{array}$$

Relationships. The architectural relationships among components are defined by a tuple $Rel = \{IPType, IRType, Provider, Requirer, Contingency, ParamType, Definer, ParentTypes, DelegProv, DelegReq\}$ in which $IPType$ (resp. $IRType$) is a total function that maps a provided interface in $IProvided$ (resp. a required interface in $IRequired$) to its type in $ITypes$, $Provider$ (resp. $Requirer$) is a total function that maps a provided interface in $IProvided$ (resp. a required interface in $IRequired$) to its component type in $CTypes$. $Contingency$ associates each required interface in $IRequired$ with its contingency. $ParamType$ is a total function that associates with each parameter in $Params$ its type in $PTypes$, $Definer$ is a total function to define the component type in $CTypes$ for each parameter, $ParentTypes$ associates each component type with the component types of its parent components[1], and $DelegProv$ (resp. $DelegReq$) describes pairs of provided interfaces (resp. required interfaces) that are linked by a delegation from a parent component to one of its subcomponents.

Example 4 (3 relationships of the VPA example). For the VPA component model, one has $Provider = \{connV \mapsto Vehicle, leader \mapsto Platoon\}$, $Requirer = \{vehicles \mapsto Road, next \mapsto Vehicle\}$, and $DelegateProv = \{connV \mapsto leader\}$.

Instantiation. An instantiation provides the main entities of the component-based system and thus defines its particular configuration, which consists of the components that are present and put together thanks to their relationships. The instantiation is a 6-tuple $Inst = \{Comps, CT, Parents, Binds, DelProv, DelReq, Value\}$ in which $Comps$ is the set of component *instances*; a total function, called CT, associates with each component in $Comps$ its type in $CTypes$; $Parents$ associates with each component in $Comps$ the set of its parent components; $Binds$ is a relation to bind provided and required interfaces of components; $DelProv$ (resp. $DelReq$) describes the delegated interface of a sub-component in relation with the delegating interface of the parent component; and $Value$ provides the value of each component parameter.

Example 5 (Instantiation of the VPA example). A component can be instantiated several times, as e.g. the components of type $Vehicle$. The configuration in

[1] each component type is mapped to a set of component types, as we assume that components can be shared by composite components.

Fig. 2 is given by the following instantiation:
$Comps = \{v1.1, v1.2, v1.3, v2.1, v2.2, v3, v4, v5, r, p1, p2\}$
$CT = \{\{v1.1 \mapsto Vehicle, ..., p1 \mapsto Platoon, ..., r \mapsto Road\}$
$Parents = \{v2.1 \mapsto \{p1\}, ..., v3 \mapsto \emptyset, ..., p \mapsto \emptyset, r \mapsto \emptyset\}$
$Binds = \{((v3, connV), (r, vehicles)), ..., ((v2.2, connV), (v2.1, next)), ...\}$
$DelProv = \{((v1.1, connV), (p1, leader)), ..., ((v2.1, connV), (p2, leader))\}$
$DelReq = \{\}$
$Value = \{v1.1 \mapsto \{battery \mapsto 31, position \mapsto 253.3, ...\}, ...\}$

In addition, following [20], set-theoretical constraints on this architectural model are provided so as to express: (*i*) the consistent typing of components, (*ii*) the consistent binding of interfaces, and finally (*iii*) the consistent parent relationship. For example, only components having a common parent can be bound; mandatory contingencies are fulfilled; a delegated interface of parent component is bound to an appropriate interface of a child component. Finally, some constraints inherent to the considered system are expressed as *system-dependant* invariant properties, like in [20]. They are also needed for the system configurations to be consistent.

We refer to [9,18] for the definition of components, interfaces, bindings, etc., and their consistent assembly obeying invariants. We call a state or a configuration of a component-based system a set of instantiated above-mentioned architectural elements together with their types and relations to link them.

4 Generation of Initial Configurations

Given the component-based model $\langle Elem, Rel, Inst\rangle$, this section describes a configuration generation algorithm that is used to enumerate all possible symmetry-free solutions of the CSP defined by the component model, in order to produce initial configurations. The aim of this combinatorial algorithm is to build a set of configurations that are correct-by-construction, especially regarding the architectural and consistency constraints that guarantee the correct parenting, delegations and bindings. Nevertheless, the execution of the test cases from all the computed configurations can be a tedious task, especially the setup of the test environment for a given configuration. Hence, some of the solutions can be sampled to produce a reduced set of configurations that are different from each other. These configurations can then be used as initial configurations for the online testing process described in Sect. 1.

4.1 Combinatorial Algorithm

The test generation algorithm is summarized below as Algorithm 1. It takes as an input component model parts *Elem* and *Rel*, and aims to produce all instantiations $SInst$ up to a given size (expressed as the number of components) that fulfill the description. In order to eliminate irrelevant configurations w.r.t. the system-dependant invariant (notably to restrict possible bindings, e.g. in order to prevent vehicles to be connected to each other outside a platoon), an invariant function can be provided in order to define valid configurations.

Algorithm 1 is parameterized by: the minimal and maximal number of components of each type, the total number of components, a parameter instantiation function, which aims to determine how component parameters are supposed to be valued, and an invariant function which is supposed to provide additional constraints on the configurations, which complement the description of the component model and rule out irrelevant configurations.

The combinatorial algorithm proceeds by successive steps. Each step consists in identifying the possible solutions before considering the valid one, one-by-one to proceed to the next step. Once a given step has explored all the possibilities,

```
1: **Inputs**
2:    $Elem$
3:    $Rel$
4:    $N : int$
5:    invariant: $Inst \to \mathbb{B}$
6:    genParameters: $Comp$, $CT$, $Definer \to Values$
7: **Output**
8:    $SInst$ // the set of possible instantiations
9: **Begin**
10: SInst $\leftarrow \emptyset$
11: for all $Comp$, $CT$ from genComponents(N) do
12:    for all $Parents$ from genParenting($Comp$, $CT$) do
13:       if not isFresh($Parents$) then
14:          proceed to the next value of $Parents$
15:       end if
16:       for all $Delegations$ from genDelegations($Comp$, $Parents$) do
17:          if not isFresh($Delegations$) then
18:             proceed to the next value of $Delegations$
19:          end if
20:          for all $Binds$ from genBindings($Comp$, $Parents$, $Delegations$) do
21:             if not isFresh($Binds$) then
22:                proceed to the next value of $Binds$
23:             end if
24:             $Values$ = genParameters($Comp$, $CT$, $Definer$)
25:             $Inst = \langle$ $Comp$, $CT$, $Parents$, $Delegations$, $Binds$, $Values$ $\rangle$
26:             if invariant($Inst$) then
27:                $SInst \leftarrow SInst \cup Inst$
28:             end if
29:          end for
30:       end for
31:    end for
32: end for
33: End
```

Algorithm 1: Initial configurations generation

the algorithm backtracks to the previous step to consider the next solution before moving onto the next step. The different steps are the following.

Step 1. The algorithm starts (line l.11) by considering all possible partitions of the components according to their types, bound by a maximal cardinality size. This step relies on the $CTypes$ description in $Elem$. Each pair from $Comps \times CT$ (of components and types in $Inst$) is considered for the subsequent step.

Step 2. The second step (l.12) aims to produce, for a given set of instances, a parenting relationship that fulfills the following constraints: (i) each composite component has at least two children, and (ii) no loop may appear in the parenting relationship. At this step, the isFresh function (l.13) is used to detect if the solution that is computed has been already encountered modulo permutation in a previous iteration of the current loop, as illustrated by Fig. 3. If so, the solution is not considered for the subsequent step, and the algorithm proceeds the next parenting solution.

Step 3. The third step (l.16) consists in delegating required or provided interfaces of the composite components to one of its children. To save space $Delegations$ represents both $DelProv$ and $DelReq$ described previously. All interfaces of the composites have to be delegated to their inner components. Similarly to the previous step, symmetrical solutions are ruled out, as illustrated by Fig. 4.

Step 4. The fourth step (l.20) consists in computing, based on the current parenting and delegations, a binding of compatible required and provided interfaces, that satisfies architectural constraints (e.g. only components with the same parent can be bound) and contingency constraints (single/multiple, mandatory/optional). Here again, symmetrical solutions, as illustrated by Fig. 5, are not considered.

Step 5. Once the structure of component system is generated, the algorithm eventually computes data values for the component parameters, according to their type, based on a given valuation function that can be user-defined (l.24-25). For both discrete and continuous domains of the considered parameters, this function makes use of the *Beta*-distribution method [12] with parameterized probabilistic distributions allowing the user to vary the density of random draw from these domains, and to automate the process. In the end, the configuration that has been computed is checked against the invariant (l. 26), before being stored (l. 27).
Finally, only valid and consistent configurations up to size N are computed[2].

Proposition 1. *Given number N of components' instances to be generated, Algorithm 1 terminates either by providing a set $SInst$ of consistent configurations with up to N components (without those obtained by permutations of architectural elements), or by returning the empty set if none of the configurations satisfies consistency constraints or system-dependant invariant.*

[2] The interested reader can find an implementation of this algorithm at `https://fdadeau.github.io/CSConfigGen/`

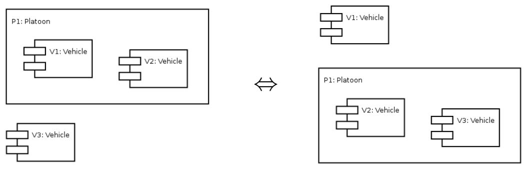

Fig. 3. Symmetries in parenting relationships

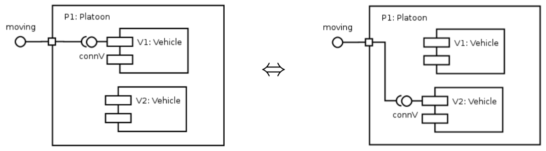

Fig. 4. Symmetries in delegations

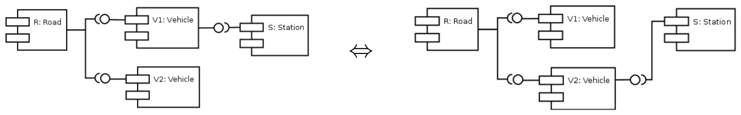

Fig. 5. Symmetries in bindings

4.2 Initial Configuration Sampling

Once initial configurations generated, there is a need to select some of them for testing the system under adaptation policies. Intuitively, while using coverage criteria for handling test inputs, the greater the difference between the configurations, the higher the coverage rate will be.

Comparing configurations. The difference between the configurations is computed by processing them two by two. This computation can be divided into 2 parts: on the overall architecture, and on particular system-dependant features. First, we compare the overall structure of the configurations by counting the difference Δ_{comps} between the numbers of the components instances (primitive and compound ones), and the difference $\Delta_{hierachy}$ between the numbers of all the ancestors of the involved components. This way, a better coefficient will be given to a complex configuration with nested composite components, compared with a flat configuration. Depending of the systems under consideration, the number of bindings may differentiate the configurations as well, hence $\Delta_{bindings}$.

In [1], a negative inverse exponential function is used to limit the complexity score, together with a coefficient to scale it between two values. In the same spirit, we have chosen a logarithmic function to limit our coefficient values. So, in the end, a coefficient on the overall architecture difference is computed by the following formulae:

$$k = log_{10}(\Delta_{comps}^{\Delta_{hierarchy}+\Delta_{bindings}})$$

Example 6. Let us consider again the configuration in Fig. 2 with 8 vehicles, 2 platoons, 1 road, and 3 bindings (configuration A). Let us compare it to configuration B composed of 5 solo vehicles on the road. One has $\Delta_{comps} = 5$ for primitive and compound components, $\Delta_{hierarchy} = 5$, and $\Delta_{bindings} = 3$, so $k = log_{10}(5^{5+3}) = 5.59$.

Second, while testing adaptation rules, the test case generation may be impacted by the values of the component parameters, that are involved in the rules. So, the validation engineer should be able to examine the parameters that are worthy of attention. To compare two configurations with a different number of the components of the same type, for each parameter of interest of the configuration with more instances, the closest values are two by two selected for merging, until the same number of values is obtained. So, given number n of instances of the same type considered, l_1 and l_2 sorted lists of n values of parameter Par, the proportional difference is computed by the following formulae:

$$score_{Par} = \frac{100}{n \times (max_{Par} - min_{Par})} \times \sum_{i=1}^{n} |l_{1_i} - l_{2_i}|$$

where max_{Par} and min_{Par} represent resp. maximal and minimal values of parameter Par. The scores of all parameters from $Pars \subseteq Params$ that impact the adaptation rule, are aggregated in a final score, where k is the coefficient on the overall architecture difference between two configurations:

$$difScore = k \times \sum_{Par \in Pars} score_{Par}$$

Example 7. Let us consider again configurations A and B from Example 6, with a focus on *battery* parameter. Let us suppose that the battery level values for the vehicles in A are $(20, 22, 34, 54, 62, 72, 80, 99)$. As there are 3 more vehicles in A, the 3 pairs of the closest values are: $(20, 22)$, $(55, 62)$ and $(72, 80)$. After the merging step (here by averaging), one has $l_1 = (21, 34, 58, 76, 99)$. For B, let us take $l_2 = \{26, 55, 62, 74, 89\}$. Applying $score_{Par}$ to *battery* gives:

$$score_{Bat} = \frac{100}{5 \times (100 - 20)} \times \sum_{i=1}^{5} |l_{1_i} - l_{2_i}| = 0.25 \times 42 = 8.4$$

Assuming *distance* parameter score being $score_{distance} = 5.1$, the aggregated difference score is then: $difScore = 5.59 \times (8.4 + 5.1) = 75.46$.

Configuration sampling. Sampling consists in reducing the set of possible configurations to a subset of size NS, in which the difference scores between the configurations are maximized. Such an optimization problem can be solved by various kind of approaches, such as SAT-solving, linear programming, clustering [13], genetic programming [19], etc. For this work, we choose to implement a greedy algorithm, shown below as Algorithm 2. Based on the set of m generated initial configurations computed, an $m \times m$ score matrix (named $Scores$ line 2) is built, where $a_{i,j}$ element represents $difScore$ between configuration i and configuration j [3]. NS (line 2) denotes the number of configurations to select, which is the cardinality of $Configs$ set of indexes of selected configurations (line 6).

[3] In this matrix $a_{i,i} = 0$ and $a_{i,j} = a_{j,i}$. The complexity of the computation is quadratic in the number m of configurations.

The algorithm starts by selecting the biggest score in $Scores$ matrix with function selectHigestScore (line 6), and marks the corresponding element as selected (line 7). $Configs$ set is then updated (lines 8, 9). Here selectHigestScore($Scores$) function browses the matrix given parameters, and returns the indexes corresponding to the biggest difference score between i-th and j-th configurations. Then, in the corresponding row i and column j the biggest remaining scores are chosen (lines 11 and 14), and the corresponding configurations indexes are added to $Configs$ (lines 13 and 16). Function selectHighestScoreInLine(i) (resp. selectHighestScoreInColumn(j)) browses the i-th row (resp. the j-th column) of $Scores$ matrix, and returns the index of the column (j') (resp. row i') corresponding to their respective biggest score. In order to prepare the next iteration step, the indexes are updated (lines 17, 18). The steps in lines 11 to 18 are repeated, until $Configs$ size reaches NS.

By construction, only configurations with big difference scores are selected.

```
 1: Inputs
 2:    $Scores$, $NS$
 3: Output
 4:    $Configs$
 5: Begin
 6: $i, j \leftarrow$ selectHigestScore($Scores$)
 7: mark $a_{i,j}$ as selected
 8: Configs.add($i$)
 9: Configs.add($j$)
10: repeat
11:    $j' \leftarrow$ selectHighestScoreInLine(i)
12:    mark $a_{i,j'}$ as selected
13:    Configs.add($j'$)
14:    $i' \leftarrow$ selectHighestScoreInColumn(j)
15:    mark $a_{i',j}$ as selected
16:    Configs.add($i'$)
17:    $i \leftarrow i'$
18:    $j \leftarrow j'$
19: until Configs.size() $< NS$
20: return $Configs$
21: End
```

Algorithm 2: Initial config. sampling

Proposition 2. *Given NS, number of configurations to select from $SInst$ set of size m, Algorithm 2 terminates by providing $Configs$ set of size $NS \leq m$ of configuration indexes from $SInst$ that have the most significant difference scores.*

4.3 Integration into the Online Test Generation Process

The online testing process relies on the usage models, one per component type, which are probabilistic automata. They capture the behavior of components and determine, for a given state, which external events can be sent to the component, at a given rate. As an example, Figure 6 represents the usage model of Vehicle type components. In this figure, edges with solid lines represent external events that may be used for stimulating (i.e., testing) the component, whereas dotted lines represent internal events that may occur and change the state of the automaton. Edges may also be labelled by δ, which represents a quiescence, meaning that no event will be sent to the system. Finally, the number in parentheses represents the probability for the considered event to be selected.

The usage models are specific to each component type, and each state of the automaton represents a given configuration for the component. As a consequence, we assume that there is, for each component type, a function to determine the initial state of usage model for a component of this type. We denote $init_{CType}$ this function.

Example 8. Assuming that a component v of type Vehicle appears in the instantiation $Inst$ that is generated by the process described in 4.1 and selected by the process described in 4.2, this component's automaton initial state will be given by the following function:

```
function $init\_Vehicle(v,\ Inst)$
   if (($v,\ connV$) $\notin$ $Inst.Binds$) return 0
   else if ($Inst.Parents(v)$ $\neq$ $\emptyset$) return 1
   else return 2
```

5 Experimentation

This section describes experiments to assess the testing approach described in Sect. 1 and displayed in Fig. 1 while using Algorithms 1 and 2 for initial configuration automatic generation and sampling. The goal of the experiments is to answer the following research questions.
[RQ1] *To what extent the use of different initial configurations improves the generated tests? (shorter? improve coverage? find more faults?)*
[RQ2] *To what extent the symmetry-breaking in Algorithm 1 reduces the number of generated configurations?*

On the experiments. To experiment, a simulator of the VPA example has been developed as a Java program (almost 6000 lines of code). It can be modified at will, e.g. to set up initial configurations and sequences of events. It is also possible for the validation engineer to modify the implementation of the adaptation policies that guide system's reconfigurations. Actually, the implementation may depend on the reconfigurations utilities and on strategies for handling priorities of the reconfigurations with the same utility level. The validation of implementation choice has been described in [8].

For the VPA system under test, 8 adaptation rules have been designed, that integrate 5 temporal properties of interest.

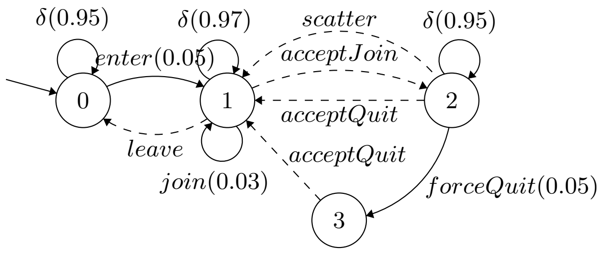

Fig. 6. Usage model for the Vehicle components

Example 9. In addition to 2 adaptation rules for vehicles from Example 2 involving *battery* parameter, we consider the following rule with *distance* parameter:
when after *Join normal* until *Quit normal* **and** *VehicleId.distance* < 10
if *state* $!=$ *leader* **then utility of** *QuitPlatoon* **is** *high*

On the experimental protocol. Let us now describe the experimental protocol. Once the set of initial configurations $Inst$ of cardinality 1200 generated by applying Algorithm 1, matrix $Scores$ of difference scores is computed. Afterwards, 10 closest configurations with small difference scores and 10 farthest ones with big difference scores are selected by applying twice Algorithm 2 to $Score$ matrix.

As the test generator performs Markovian walk over components usage models, the experiment is replayed 170 times for each set of configurations, one by one ($2 \times 10 \times 170$), to provide a confidence in the experimental results. This allows observing produced traces with actually triggered reconfigurations in order to compute the coverage criteria rate [7] as recalled in Sect. 2.

For the VPA example simulation, as displayed in Fig. 1, running an experiment consists then in starting from a selected initial configuration and letting the test generator deal with usage models of components to send the events at a given rate to the system under adaptation policies. During 3000-step experiments, the reconfigurations occur (with traces produced) and make system's architecture evolve.

On the results. The coverage rate is separately aggregated for the properties and for the adaptation rules by applying coverage criteria described in [7] and reminded in Sect. 2. Given the set of initial configurations, for each experiment the coverage rate for the rules is the ratio of the number of adaptation rules, that are covered by at least one test case starting from a configuration from this set, to the number of rules under consideration.

Table 1 below reports on the experimental results [4], where the lines correspond to the coverage rate reached for the properties and rules, depending on the configuration set chosen with either small $difScore$ values, or big $difScore$ values. The columns represent the running experiment number (from 1 to 170), with an extract of 9 experiments below. For example, for the run in column $n+1$, the first line (Small dif. score) indicates 94% of coverage for properties and 75% for rules, where as the second line (Big dif. score) indicates 100% coverage for properties and 75% for adaptation rules. The Av. column indicates the average coverage rate of a sub-line. For example, in the first line (Small dif. score) the first sub-line indicates 86.5% properties coverage on average for 170 performed experiments. Also, for the 8 adaptation rules, 0% coverage rate indicates that no rule has been triggered during the 3000-step experiment with about 300 external events sent to the SUT [5], whereas 100% coverage says that all the adaptation rules have been triggered. The column M.freq. indicates the most often seen

[4] More results are in Table at `https://fdadeau.github.io/CSConfigGen/table.html`

[5] Let us note that for each experiment, on the given clock tick, on average 10% of steps correspond to the events from the usage models sent to the SUT (cf. (4) in Fig. 1), whereas δ occurs for the remaining 90% of steps.

rate value over 170 experiments performed for each set of configurations. So, 75 indicates that 75% is the most frequent coverage.

Let us note that the most frequent 75%-rate for adaptation rules is due to the rules, whose *QuitDistance* reconfiguration is not triggered because of *distance* parameter involved. For these rules, 3000-step experiments are not long enough for decreasing *distance* values, and the scope and guard predicates of the concerned rules remain false.

Run number			23	24	25	26	27	28	29	30	31		Av.	M.fr.
Small dif.	Prop.Cov.(%)	...	56	94	17	94	56	61	94	100	94	...	86.5	94
score	Rule Cov.(%)	...	0	75	0	75	0	13	75	75	75	...	58.5	75
Big dif.	Prop.Cov.(%)	...	100	100	100	100	100	100	94	100	100	...	98.1	100
score	Rule Cov.(%)	...	94	75	75	75	75	100	75	75	75	...	80.2	75

Table 1. Extract of experimental results and coverage rates

On the symmetry elimination. To address **RQ2**, we ran an experiment, which consists in counting the number of configurations that are generated with or without the symmetry detections that we have considered.

For the VPA example we designed 5 setups that differ in the minimal and maximal number of components of each type that are generated. The invariant specifies that vehicles that are not in a platoon are not connected together. The setups are as follows: Setup#1: 1 Road, 1 to 5 Vehicles, 0 to 2 Platoons, 0 to 1 Station. Setup#2: 1 Road, 5 Vehicles, 0 to 2 Platoons, 0 to 1 Station. Setup#3: 1 Road, 6 Vehicles, 0 to 2 Platoons, 0 to 1 Station. Setup#4: 1 Road, 1 to 5 Vehicles, 1 Platoon, 0 to 1 Station. Setup#5: 1 Road, 1 to 5 Vehicles, 0 to 2 Platoons, 1 Station.

Setup	#1	#2	#3	#4	#5
P, D, B, I	62	26	39	30	44
P, D, B, $\overline{I}$	181	93	166	54	149
$\overline{P}$, D, B, I	325	244	1098	158	222
$\overline{P}$, D, B, $\overline{I}$	640	482	2398	378	491
$\overline{P}$, $\overline{D}$, B, I	1083	897	5270	410	700
$\overline{P}$, $\overline{D}$, $\overline{B}$, I	2249	1953	17495	1294	1466
$\overline{P}$, $\overline{D}$, $\overline{B}$, $\overline{I}$	22971	21213	337625	4174	21218

Fig. 7. Number of configurations for each setup

By turning on or off (denoted with a line over the corresponding symbol) some of the symmetry eliminations (parenting P, delegations D, bindings B) and invariant filtering (I), we obtain the results shown in Table 7. Setup #1, #3 and #5 took about 25 seconds on a standard laptop (Dual-core i5 1.6GHz with 8Go RAM) to generate 21.000–23.000 configurations. Due to the exponential blow up of the unfolding, it takes about 20 minutes to generate the 337.625 configurations of Setup #3, which was reduced to 16 minutes when enabling symmetry reductions.

All symmetry reductions are clearly relevant in order to master the combinatorial explosion, showing that only a tractable number of configurations is generated even for highly combinatorial setups. On this case study, the experimentation also shows that a large set of irrelevant configurations can be generated, based only on the description of the component model (without considering the invariant).

Notice that these symmetrical configurations have 0 for the difference score, and thus, only one of them will at most be kept by the selection process. As a consequence, removing them at the soonest prevents useless computations from being performed.

The obtained experimental results allow assessing the use of automatically generated initial configurations and their sampling. Indeed, they show that the set of the generated configurations with significant difference scores gives better coverage rates, thus answering our research questions.

6 Related Work

System's configurations are required for online testing approaches to work. In our approach, these configurations are automatically generated from a component model by using boundary testing [3,21,24] to generate system's values. In [21], the authors present a test case generation based on boundary goals derived from a formal model, with a feedback to refine boundary criteria if the boundary goals are not reachable. In our approach, as in [14], the boundaries of the parameters are defined in the model; in this respect our contribution consists in generating a wide diversity of combinations.

Our symmetry filtering approach for generating initial configurations is close to the TACO tool [11], which applies SAT-based techniques instrumented with a symmetry-breaking predicate to JML-annotated sequential Java programs in order to eliminate some isomorphic models. Our approach goes further, as we also consider hierarchical objects.

In the field of software product lines (SPLs), the configuration spaces are often determined by features and constraints over them, modeled as feature models (FMs). A feature oriented testing (FOT) described in [16] applies FMs to test-case designs for black-box testing. In this approach, SAT-based automated test-suite generation and correctness checking of test-case designs are performed. In [15] the authors present a comparative study of combinatorial testing (CT) and random testing (RT) algorithms for testing SPLs. On the chosen benchmarks, this study shows that the diversity of configurations sampled by CT is 2 to 3 times higher than those sampled by RT.

As reported in [17], in fuzz testing, the performance substantially varies depending on input configuration files, or seeds, used to start fuzzing with. However, most papers (among 32) have treated their choice casually, apparently assuming that any seed would work equally well, without providing particulars. Our experimentations with initial configurations also show that testing process heavily depends on them.

7 Conclusion and Future Works

In this paper, we have presented an approach to automatically generate initial configurations for testing component-based systems. The presented algorithm allows generating structured data composed of architectural elements and relationships to link them, while satisfying general consistency constraints expressed by invariants. System-dependant instantiation of component parameters is integrated at appropriate algorithm steps, in order to generate meaningful inputs for testing. The provided experimental results on a simulation of platoons of autonomous vehicles show that this approach allows us to improve the capability of fulfilling coverage criteria by using different initial configurations. Thus, the present work usefully extends the approach for testing component-based systems in [7]. This approach is generic for any component-based framework, and can be extended to adapt to component models, such as BIP [2].

One of the future work directions consists in providing the user with a refinement method in order to enlarge or reduce defined boundaries. Also, we intend to improve detection of dubious reconfigurations, and to provide the user with the means to validate that an adaptation policy, that is correctly implemented, fulfills extra-functional properties, such as optimized resource-consumption, etc.

References

1. B. Alkan and R. Harrison. A virtual engineering based approach to verify structural complexity of component-based automation systems in early design phase. *Journal of Manufacturing Systems*, 53:18–31, 2019.
2. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in BIP. In *Proc. of the Fourth IEEE Int. Conf. on Software Engineering and Formal Methods*, SEFM'06, pages 3–12, Washington, DC, USA, 2006. IEEE Computer Society.
3. B. Beizer and J. Wiley. Black box testing: Techniques for functional testing of software and systems. *IEEE Software*, 13(5):98–, 1996.
4. C. Bergenhem. Approaches for facilities layer protocols for platooning. In *IEEE 18th Int. Conf. on Intelligent Transportation Systems, ITSC 2015*, pages 1989–1994. IEEE, 2015.
5. F. Bouquet, B. Legeard, and F. Peureux. CLPS-B - A constraint solver to animate a B specification. *Int. J. Softw. Tools Technol. Transf.*, 6(2):143–157, 2004.
6. F. Chauvel, O. Barais, I. Borne, and J.-M. Jézéquel. Composition of qualitative adaptation policies. In *23rd IEEE/ACM Int. Conf. on Automated Software Engineering (ASE 2008)*, pages 455–458. IEEE Computer Society, 2008.
7. F. Dadeau, J.-P. Gros, and O. Kouchnarenko. Testing adaptation policies for software components. *Softw. Qual. J.*, 28(3):1347–1378, 2020.
8. F. Dadeau, J.-P. Gros, and O. Kouchnarenko. Online testing of dynamic reconfigurations w.r.t. adaptation policies. *Modeling and Analysis of Information Systems*, 28(1):52–73, 2021. (In Russian).
9. J. Dormoy, O. Kouchnarenko, and A. Lanoix. Using temporal logic for dynamic reconfigurations of components. In *FACS, Int. Symp. on Formal Aspects of Component Software*, volume 6921 of *LNCS*, pages 200–217. Springer Berlin Heidelberg, 2010.

10. R. El Ballouli, S. Bensalem, M. Bozga, and J. Sifakis. Programming dynamic reconfigurable systems. *Int. J. Software Tools for Technololy Transfer*, 2021.
11. J. P. Galeotti, N. Rosner, C. G. López Pombo, and M. F. Frias. Taco: Efficient sat-based bounded verification using symmetry breaking and tight bounds. *IEEE Transactions on Software Engineering*, 39(9):1283–1307, 2013.
12. A. Gupta and S. Nadarajah. *Handbook of beta distribution and its applications*. CRC press, 2004.
13. J. A. Hartigan and M. A. Wong. A k-means clustering algorithm. *JSTOR: Applied Statistics*, 28(1):100–108, 1979.
14. A. Hussain, S. Tiwari, J. Suryadevara, and E. Enoiu. From modeling to test case generation in the industrial embedded system domain. In *STAF'18 Collocated Workshops, Revised Selected Papers*.
15. H. Jin, T. Kitamura, E.-H. Choi, and T. Tsuchiya. A comparative study on combinatorial and random testing for highly configurable systems. In *Testing Software and Systems - 32nd IFIP WG 6.1 International Conference, ICTSS 2020, Naples, Italy, December 9-11, 2020, Proceedings*, volume 12543 of *Lecture Notes in Computer Science*, pages 302–309. Springer, 2020.
16. T. Kitamura, T. B. N. Do, H. Ohsaki, L. Fang, and S. Yatabe. Test-case design by feature trees. In *Proc. 5th International Symposium, ISoLA 2012*, volume 7609 of *LNCS*, pages 458–473. Springer, 2012.
17. G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 2123–2138, 2018.
18. O. Kouchnarenko and J.-F. Weber. Adapting component-based systems at runtime via policies with temporal patterns. In J. L. Fiadeiro, Z. Liu, and J. Xue, editors, *FACS, 10th Int. Symp. on Formal Aspects of Component Software*, volume 8348 of *LNCS*, pages 234–253. Springer, 2014.
19. J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.
20. A. Lanoix, J. Dormoy, and O. Kouchnarenko. Combining proof and model-checking to validate reconfigurable architectures. *Electron. Notes Theor. Comput. Sci.*, 279(2):43–57, 2011.
21. B. Legeard, F. Peureux, and M. Utting. Automated boundary testing from Z and B. In *FME 2002: Formal Methods - Getting IT Right*.
22. K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. ISSTA'04, page 133–142, New York, NY, USA, 2004. Association for Computing Machinery.
23. S. Taha, J. Julliand, F. Dadeau, K. Cabrera Castillos, and B. Kanso. A compositional automata-based semantics and preserving transformation rules for testing property patterns. *Formal Aspects of Computing*, 27(4):641–664, dec 2015.
24. M. Utting, A Pretschner, and B Legeard. A taxonomy of model-based testing approaches. *Softw. Test. Verification Reliab.*, 22(5):297–312, 2012.
25. G. H. Walton, J. H. Poore, and C. J. Trammell. Statistical testing of software based on a usage model. *Software: Practice and Experience*, 25(1):97–108, 1995.