Budapest University of Technology and Economics
Dept. of Measurement and Information Systems

Automated Evaluation of the Test Traces of Autonomous Systems

TECHNICAL REPORT

Gergő Horányi
István Majzik Dr.

2012.12.20.

Executive summary

Autonomous systems can make and execute decisions to achieve their goals without direct human control [Connelly 2006]. A significant part of these systems, for example autonomous robots used in the household or manufacturing, operate in real, uncontrolled environment, thus they must properly react to unexpected combinations of environmental objects and events: they shall be *robust* to be capable of correctly handling unforeseen situations and *safe* to avoid harmful effects with respect to humans. This way the evaluation of their robustness (precisely, the degree to which they can function correctly in the presence of invalid inputs or stressful environmental conditions) and functional safety forms an important part of their verification and validation.

To support the testing of the robustness and safety of the context-aware behaviour of autonomous robots, in the R3-COP project [R3-COP] we developed a model based testing concept [Micskei 2012]. It is characterized by three main components: (1) context and requirements modelling to represent test requirements, (2) a search based test generation method to derive stressful contexts for robustness testing, and (3) an automated off-line test evaluation approach. We focused on the systematic generation of the stressful contexts that can be derived from the context model and the behavioural requirements, in order to satisfy robustness related test goals and related coverage criteria.

Having these test contexts generated, the tests can be executed in a real or simulator based environment: each generated test context is set as initial context of execution, then the autonomous system is started to perform its mission, and the events, actions and context changes are recorded in a *test trace* until the mission is ended.

According to this testing concept, the evaluation of the test traces has several challenges. Each requirement shall be mapped to a test oracle that is able to compare the sequence of contexts, events and actions recorded in a test trace to the ones allowed by the requirement. Since the test contexts represent complex situations that may match several requirements, each test trace shall be compared to each requirement. Moreover, this comparison shall be started in each step of the test trace (even in an overlapped way) since it may happen that during a test execution an initial context of another requirement evolves. The comparison of context objects and their properties shall take into account the hierarchy of object types and the abstract relations (like “tooClose” and “near” relations mapped to physical distances) that are included in the context model and in the requirements.

To address these challenges, we propose solutions (algorithms and tools) to solve the test evaluation problem in an efficient way: for context matching, we use a graph based algorithm that is optimized for matching multiple graphs at the same time, and adapt it to hierarchical context models with multiple valuations.

1 Construction of test oracles

According to our test strategy, the test traces captured during the test execution shall be evaluated against each of the scenarios (requirements) to identify whether a scenario is triggered and violation of any requirement is detected. The test evaluation is performed using so-called *test oracles*. The high-level inputs and outputs of the oracles are illustrated on Figure 1.

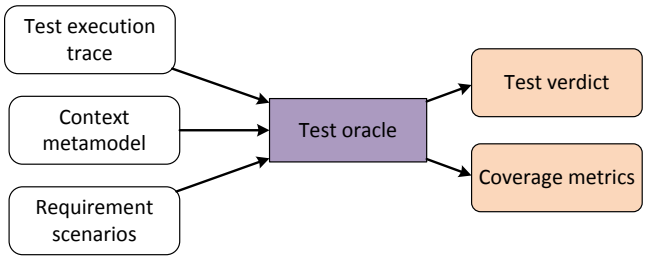


Figure 1 The high-level inputs and outputs of the test oracle

- The test execution trace is a precise log of the events, actions and context related changes recorded by the test execution environment.
- The test oracle also requires the context metamodel. This metamodel is used to evaluate a matching between the objects in the test execution trace and the objects in the requirement scenarios, taking into account the type hierarchy.
- The third input of the oracle is the set of requirement scenarios. The definition of the requirement scenario language can be found in deliverable D4.2.1v2. The requirements refer to events and actions defined in the action model and context fragments according to the context metamodel.

As our requirement specification language contains two parts, the test oracle has two tasks:

- Matching the configurations (contexts) recorded in the test trace to the *context fragments* specified in the scenarios. As described in D4.2.1, each context fragment is an instance of the context metamodel and specifies those parts of the SUT's context that are relevant from the point of view of the given scenario. Context fragments are not only used to specify initial conditions (in the form of initial contexts), but also to specify interim conditions (in the form of interim contexts) and the expected final conditions (final contexts, respectively). All these context fragments together define a *context sequence*. Accordingly, the configuration recorded in the trace shall be matched to the context sequences defined in each scenario.
- Matching of the events and actions in the trace to the ones in the scenario. This matching problem can be traced back to the theoretical problem of generating a so-called *observer automaton* for each scenario that processes the trace and signals when the events and actions are matched with the ones specified in the scenario, taking into account the specific modalities used in the scenarios.

The solution, which we present in this deliverable, solves the two tasks simultaneously. Each scenario is mapped to an observer automaton that represents not only the sequence of events and actions, but also the sequence of context changes relevant from the point of view of the scenario. The mapping algorithm, which is based on the formal semantics of the scenario language, is presented in deliverable D4.2.1.

The test oracle will start observer automaton instances (at least one instance for each scenario), which will evaluate whether a given test trace triggers, satisfies or violates the requirement formalized by the scenario and represented by the corresponding observer automaton.

A scenario is triggered if both the context fragments and the event/actions in its “trigger” part (pre-chart) are matched with the ones found in a segment of the recorded test trace. In case of a triggered scenario, its “assert” part is examined. If both the context fragments and the events/actions are matched with the ones in the subsequent postfix of the trace (that follows the segment matching the “trigger” part) then the scenario is passed, otherwise it is violated. It is also possible, that a test is inconclusive, when the preconditions in the “trigger” part are not met. The test oracle collects the results of each scenario matching, and generates a coverage report.

In the following subsections we present the algorithms responsible for test evaluation. First, we present an algorithm which performs the context fragment matching tasks. Then we present the algorithm responsible for running the observer automata and the integration of the context matching algorithm.

The main challenges of the test evaluation task can be summarized as follows:

- *Matching all scenarios*: All requirements shall be examined for matching and potential violation of the required behaviour. It may happen that a trace targeting a given scenario will also match another scenario(s) that shall be checked for violation. For example, the trace used to check a scenario which specifies that the robot shall avoid collision with a human (scenario 1) will also trigger the scenario which specifies that the robot shall issue a warning in the presence of a human (scenario 2).
- *Matching from multiple steps of the trace*: The matching of a scenario shall be examined not only from the first step of the trace but also from each step of the trace. This is motivated by the fact that the trigger part of a scenario may occur during (e.g., in the middle of) the test execution represented by the trace. For example, let us consider a trace which is recorded when a robot is tested to check whether a warning is issued in presence of a human. It may happen that during this trace the robot comes near to an object, this way the trigger part of a scenario specifying the behaviour of the robot close to an object can be matched, and this second scenario shall also be checked. Moreover, it may also happen that a scenario can be matched from multiple steps of a trace (e.g., when the robot moves close to several objects), even in an interleaving way.
- *Non-deterministic behaviour during testing*: Test data is generated to match the initial context fragments of one or more targeted scenarios, this way it is expected that the trace recorded using the given test data will match these scenarios. However, due to the non-deterministic nature of the behaviour of the SUT, it may also happen that the trace does not match the trigger part of a targeted scenario (but matches another scenario). Accordingly, targeted scenarios cannot be distinguished and left out of the detailed matching.
- *Abstraction and hierarchy of concepts*: The hierarchy of the types of context objects (e.g., when a subtype can be used in place of its ancestor) and the mapping between the abstract concepts (in the scenarios) and the concrete concepts (in the trace) shall be considered. For example, the configuration when a Robot is in a relation *closeTo* with a piece of Furniture is matched when the distance of the given robot is less than 10 cm from a stool, and also when its distance is less than 5 cm from a table (assuming that the *closeTo* relation can be mapped to a distance less than 25 cm). To resolve the abstract relations, we perform a pre-processing step on the test trace which derives the valid and relevant abstract relations on the basis of the concrete attributes of objects.
- *Dynamic changes*. In the scenarios, the changes in the environment (e.g., when an object appears or disappears) are represented by events and/or (sequence of) interim contexts. The changes of configurations in the trace shall be matched with these representations. The requirements can contain a sequence of events, actions, and interim contexts that not necessarily include the precise timing of their occurrence, only their ordering. However, there are also dynamic objects specified in the initial context

fragment that could appear/disappear with given time constraints, and the relation of these time constraints with the occurrence of the specified actions/events in the scenario is not known in advance. Therefore the precise sequence of interim contexts resulting from the appearance/disappearance of these objects can't be determined before the evaluation. Thus the test oracle shall insert an interim context when the timing of a dynamic object necessitates it.

- *Nondeterministic observer automaton:* A requirement scenario may contain alternatives in the behaviour, this way an observer automaton generated from a scenario may be nondeterministic, i.e., one state in the observer automaton may have multiple subsequent states. The evaluation shall consider all possible runs simultaneously.

To address these challenges, in our approach we first created a core algorithm for matching context fragments. This core algorithm formulates the problem as searching subgraph isomorphism between the requirement contexts and the contexts of the execution trace. Our core subgraph matching algorithm was developed bearing in mind that one context fragment from the test execution trace (called in the following *configuration graph*) shall be matched simultaneously for multiple context fragments from the requirement scenarios (called in the following *requirement graphs*).

1.1 Formulation of the problem

Context fragments, as well as configurations in a test trace are instances of the context metamodel. Instances of a metamodel can be commonly represented as labelled graphs, this way the mapping of static context fragments and configurations to labelled graphs is a straightforward idea: objects are mapped to graph vertices (where vertex labels determine the type of the object), and the relations among them are mapped to graph edges (where edge labels determine the type of the relation). The changes represented by dynamic events, however, need special consideration.

Our previously introduced context metamodel contains dynamic events. In our modelling language the dynamic events must be specified in the initial context fragment (e.g., an object will appear). Interim and final context fragments can describe only static configurations (i.e., without dynamic events). There is a given set of dynamic events that can be described in the initial context fragment. Here we consider the following dynamic events: AppearEvent, DisappearEvent, and MoveEvent. These events are associated with the objects that are concerned.

To match the sequence of static configurations recorded in the test execution trace to the requirement scenarios, we assume that a sequence of static contexts fragments can be derived from each scenario, and these precisely specify the occurrence of the dynamic events given in the initial context fragment of the scenario.

In general, interim context fragments can be used to specify the contexts when the occurrences of dynamic events are relevant from the point of view of the requirement. For example, if the initial context fragment contains an object with a MoveEvent attached to it, then we may add to the scenario the interim context fragments, when the relevant contexts (from the point of view of the requirement) resulting from the moving object appear. In Figure 2 we depict an example initial context fragment with dynamic events, and an interim context fragment that defines the relevant context (resulting from the dynamic events specified in the initial context fragment). Here the initial context fragment specifies that a robot is moving and a human will appear in the room, while the interim context fragment specifies that the moving robot is near to a piece of furniture (i.e., it is the relevant context). The sequence of context fragments presented in Figure 3 is a sequence of *static context fragments* that can be matched to contexts recorded during test execution. It contains the static part of the initial context fragment (i.e., without the human that will appear later as defined by the AppearEvent), then an interim context fragment giving that the human appears, and finally a second interim context fragment giving that the moving robot is near the furniture.

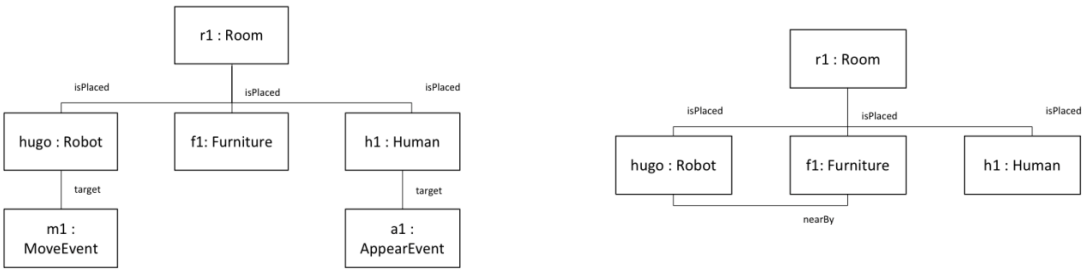


Figure 2. Initial context fragment with dynamic events and a resulting interim context fragment

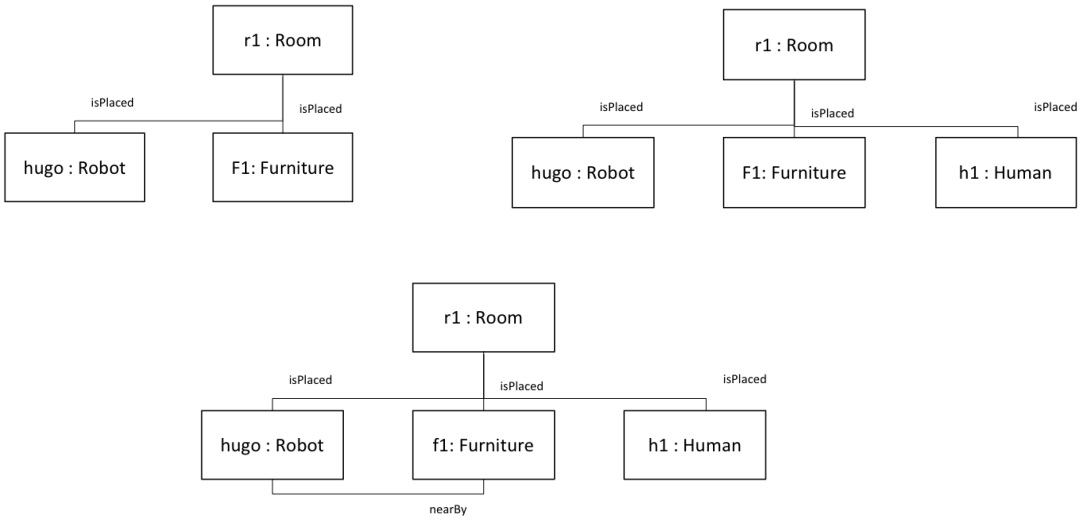


Figure 3. Context sequence with static parts

To resolve the dynamic events specified in the initial context fragment, two approaches can be followed. First, the creator of the scenario can manually add interim contexts, specifying when the relevant context occurs (with respect of the other events). This makes the specification of the requirements more complicated, moreover, it is also possible to create inconsistent scenarios. (Note that inconsistent scenarios do not cause false positive test results, but they can result in scenarios that cannot be matched with any test trace.) The other approach is the (automated) derivation of the corresponding static interim context fragments on the basis of the parameters of the event (e.g., its timeliness). It is relatively easy to insert interim context fragments in case of AppearEvents and DisappearEvents. The only challenge in inserting the necessary interim contexts is that the events and the other interim contexts contains only ordering information and no exact timing, while the dynamic events in the initial context have exact time information. Therefore we can't generate the new interim contexts before the evaluation process, but the algorithm shall generate and insert them on-the-fly, when the matching process reaches the dynamic event's time in the test trace.

In case of MoveEvents it is more practicable to manually insert interim context fragments that specify the relevant contexts as the automatic insertion of the interim context resulted by MoveEvents should require a precisely formalized representation of the path of the movement.

Having assumed that, we can formulate the context matching problem in a more abstract way as follows. The requirement scenarios are considered as sequences of static context fragments. Each context fragment is represented as a graph, and context fragment sequences are represented as graph sequences (where each graph represents a context

fragment). The vertices of the graphs represent the objects, while the edges between the vertices represent relations between the objects of the context fragment.

These graphs have also two labelling functions derived from the corresponding context fragments: one for the vertices and one for the edges. The following labels are defined:

- Labels for vertices describe the type of the object, which is represented by that vertex (e.g., Human or Furniture).
- Labels for edges describe the relations between the represented objects (e.g., *isPlaced* or *nearBy*).

The graph sequences derived from the requirement scenarios are called *requirement graph sequences*.

The sequence of configurations recorded in a test trace is mapped to graph sequence in a similar way. These graph sequences are called *configuration graph sequences*.

As we stated previously, it is not possible to check the matching of the two sequences separately from the matching of events/actions, due to the dynamic events that are specified in the initial contexts.

To address the challenges described in the previous subsection, we apply specific algorithms:

- *Matching all scenarios*: We use a graph matching algorithm that is optimized for matching multiple graphs. Requirement graphs to be checked for matching a configuration graph in the same step are represented together in a so-called *decomposition structure*. In a decomposition structure the isomorph subgraphs (even from multiple graphs) are represented only once, and this way the re-use of partial matching is supported. Re-use is efficient when the requirement graphs contain similar patterns, which is expected when we specify the behaviour of a robot in a given environment, e.g., in a kitchen, where the same (abstract) setup of objects appear in case of several requirements. In Figure 4 two requirement graphs (on the left) and their decomposition structure (on the right) is presented, here the sub-graph consisting of the nodes labelled with 1 and 2 is represented only once, thus its matching detected in the first requirement graph shall not be checked again when the second requirement graph is checked.

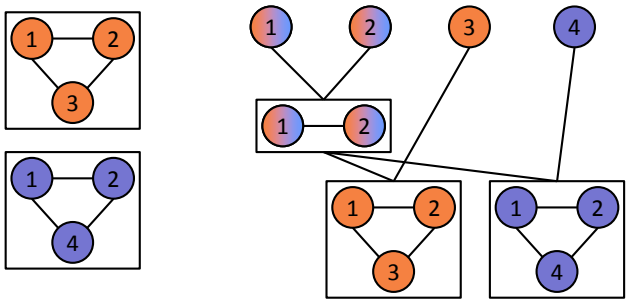


Figure 4. Two requirement graphs and their decomposition structure

- *Matching from multiple steps of the trace*: The oracle runs observer automata to check the behaviour in the execution trace. The oracle checks all possible runs of the automata (e.g. if two transitions are enabled, the oracle checks both of them) to consider the nondeterministic semantics of the observer automata. The generated observer automata have a loop transition (self-loop) in the initial location. This way the real matching can be started at any step of the trace, as there will be a run of each automaton, where the automaton remains in its initial location (as seen in Figure 5), thus skipping any potential prefix of the test trace. This approach also solves the problem of the matching one requirement overlapping with itself (e.g. on Figure 6).

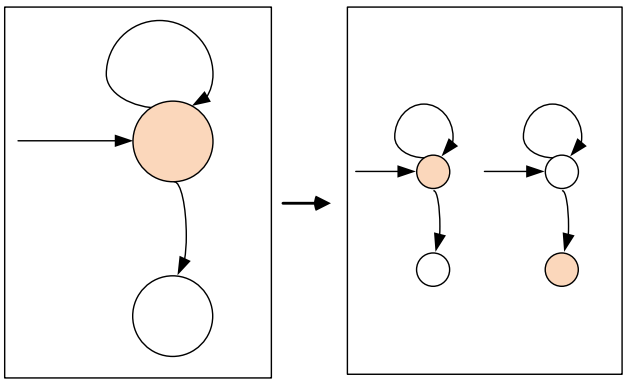


Figure 5 The nondeterministic semantics of the observer automaton: an active initial state and two potential subsequent active states

- *Non-deterministic behaviour during testing*: All requirement graphs are handled in a uniform way and checked continuously for every trace.

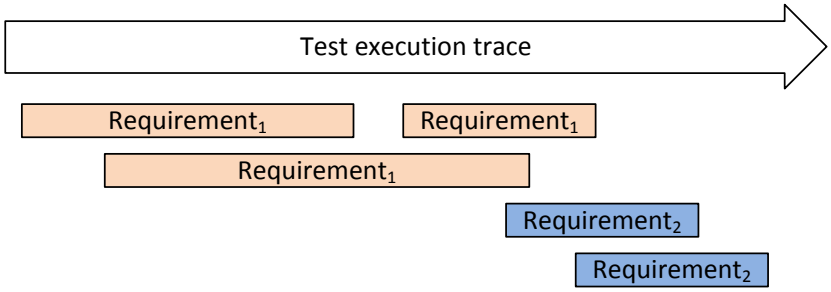


Figure 6 Interleaving of the matching of requirements

- *Abstraction and hierarchy of concepts:* To match the labels of vertices and edges (i.e., to provide so-called valuations of graph elements), the so-called *compatibility relation* is introduced (instead of the direct equivalence of labels), that conforms to the type hierarchy (and instantiations) defined in the context metamodel. Moreover, the mapping from abstract relations (in the requirement graphs) to concrete attributes (in the configuration graphs) is also considered in the matching by the pre-processing step mentioned above.

  The matching of graphs may consider several potential valuations at the same time. To keep track of these valuations, these are represented in a separate data structure. As illustrated in Figure 7, the potential valuations of the same graph in the decomposition structure (on the top of the figure) are handled together (as presented on the bottom of the figure).

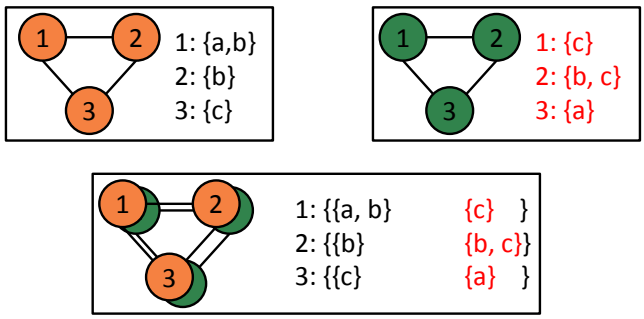


Figure 7 Handling valuations of the same graph structure

- *Dynamic changes:* When the oracle runs an observer automaton corresponding to a requirement scenario, it inserts the representation of the necessary interim contexts

(resulting from the dynamic events that were specified in the initial context fragment) into the observer automaton, in the form of new states with a context switch. The point of insertion is determined by the timing of the dynamic event. The oracle also updates the subsequent context fragments to represent the effects of the dynamic event defined in the initial context fragment.

- *Nondeterministic observer automaton:* As seen previously in Figure 5, the oracle checks all possible runs of an observer automaton. Actually, the oracle deals with even more possibilities in case of context changes. If the automaton contains a context change transition, then it is possible that there are a lot of matches (with different valuations) to the test execution trace. In this case, the oracle creates an automaton instance for each possible valuation, and runs all of them.

The matching of individual graphs, as a core algorithm, is based on the work of Messmer et al. [Messmer 2000]. On the top of this algorithm for matching individual graphs, we developed an algorithm, which executes and evaluates the observer automata.

In the following subsections we introduce the algorithms for matching efficiently multiple requirement graph sequences on a configuration graph sequence. For a high-level overview, Figure 8 shows the call hierarchy between the algorithms:

- The Observer automaton execution context is responsible for operating the observer automata with the defined automaton semantics. The execution context evaluates each run of each automaton and stores the results.
- The *Decompose* algorithm is used to construct the decomposition structure from the requirement graphs (for multiple matching).
- The *Search* algorithm is the core algorithm for matching individual graphs, in our case it searches for matching of a configuration graph on a decomposition structure representing the requirement graphs.
- The *Vertex test* is a helper algorithm for searching all possible matches of a vertex in a graph, taking into account the compatibility relation of the labels of vertices.
- The *Combine* algorithm is another helper algorithm, used for merging two valuation sets together, taking into account the compatibility of the labels of edges.

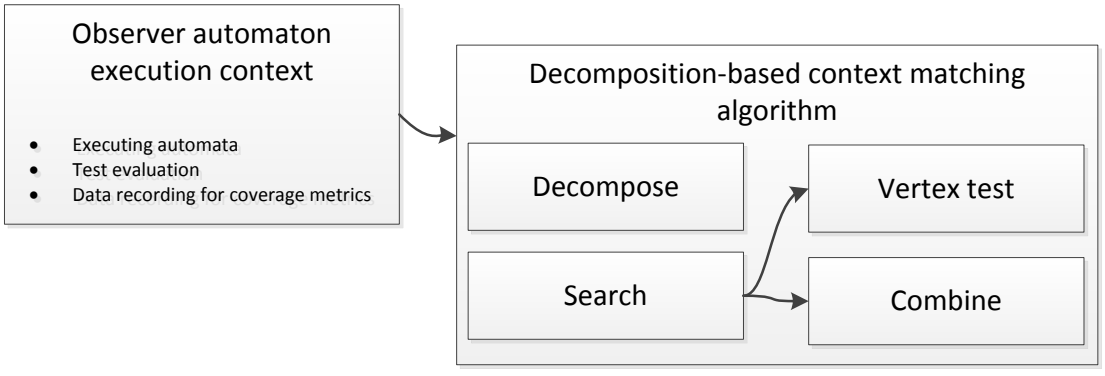


Figure 8. Overview of the algorithms

1.2 Matching context graphs using a decomposition approach

As a core algorithm, we developed an efficient subgraph isomorphism algorithm mainly based on the work of Messmer et al. [Messmer 2000]. The main focus of the algorithm is the possibility to check one configuration graph for multiple requirement graphs simultaneously.

The algorithm returns valuations, which are bijective functions between the configuration graph and the requirement graphs. An illustration of the valuations is presented on Figure 9.

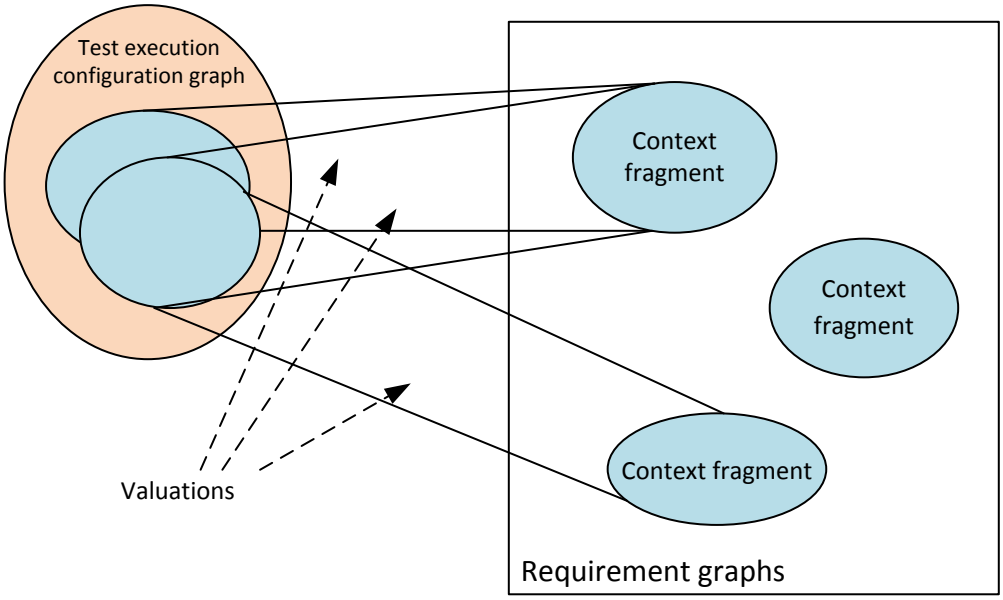


Figure 9 The graph matching algorithm's task to find the valuations

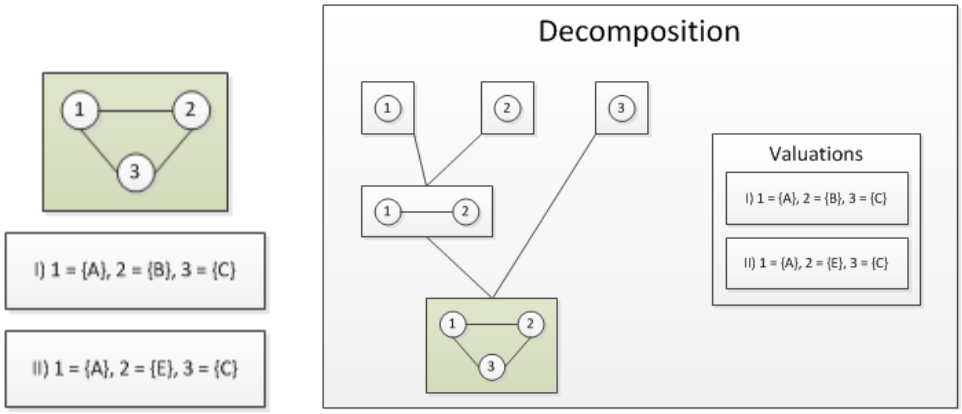
To achieve the required efficiency, the algorithm creates a decomposition structure (as introduced previously on Figure 4). This structure represents all the requirement graphs in a compacted form. The decomposition stores each isomorph subgraph only once, therefore during the search of valuations the isomorph subgraphs have to be processed only once. In case of similar graphs, this algorithm increases the efficiency of the evaluation.

The following parts of this section introduce the algorithm for creating the decomposition structure (section 1.2.1) and the search algorithm, which collects the valuations from the decomposition structure (section 1.2.2).

1.2.1 The decomposition

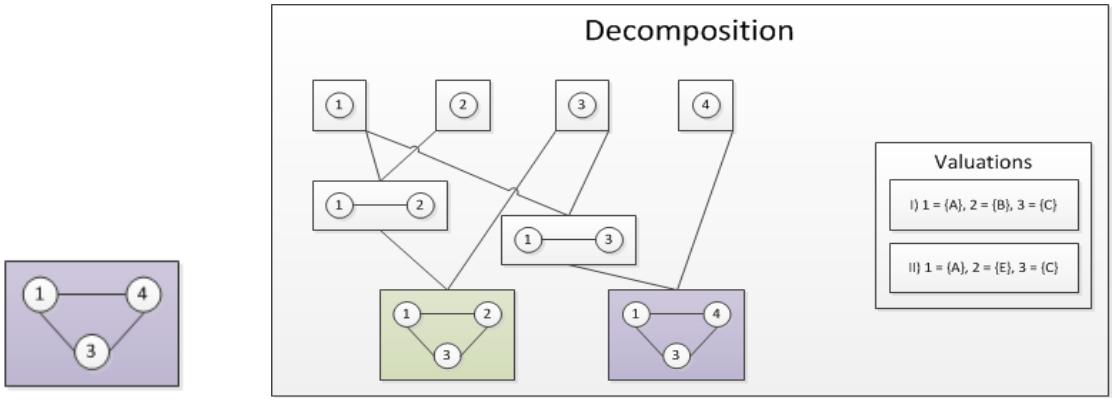
The decomposition part of the algorithm has three inputs. As the first input, it needs the result of a previous decomposition step, which will be iteratively extended (it can be an empty decomposition structure at the beginning). The second input of the algorithm is a (requirement) graph, which will be added to the structure. The last input is a set of valuations, which represent restrictions on the graph. It is possible that a graph has multiple, separate valuation sets, as well as no valuations at all.

Figure 10 (a) shows an input requirement graph with its valuation sets. The result of the decomposition step, when it is added to an empty decomposition structure, is depicted on Figure 10 (b). The decomposition groups vertices together randomly (as stated in [Messmer 2000], the order of the processing of nodes has no significant effect on execution time). Figure 10 (a) shows a next graph input of the algorithm, which is added to the previous structure, resulting in the decomposition structure presented in Figure 10 (b).



(a) Input (b) Output

Figure 10 First step: input and output for the decomposition algorithm



(a) Input (b) Output

Figure 11 Second step: input and output for the decomposition algorithm

As these structures show, the algorithm can handle multiple graph valuations together. This approach significantly reduces the size of the decomposition structure, and also makes the algorithm more efficient.

DecomposeMore

input : G graph, V set of valuations, D decomposition
output : the updated D decomposition

$S_{max} \leftarrow \emptyset$
if $|V(G)| = 1$ **then**
  exit
end
foreach $(G_i, G'_i, G''_i, E_i) \in D$ **do**
  if G_i *is subgraph of* G *and* $|V(S_{max})| < |V(G_i)|$ **then**
    $S_{max} \leftarrow G_i$
  end
end
if S_{max} *is isomorphic to* G **then**
  exit
end
if S_{max} *is empty and* $V(G) > 1$ **then**
  $S_{max} \leftarrow$ random subgraph of G
  decompose(S_{max})
end
decompose($G - S_{max}$)
$E \leftarrow$ set of edges between S_{max} and $G - S_{max}$ in G
add $(G, S_{max}, G - S_{max}, E)$ to D and associate valuations

Algorithm 1 The algorithm for building the decomposition structure

Algorithm 1 shows the pseudo-code of the decomposition. It processes graphs with more than one vertex. The algorithm creates a D decomposition structure, which contains the sub-graphs as (G, G', G'', E) 4-tuples, where G' and G'' are two disjoint sub-graphs of the G graph. The edges between G' and G'' in G are the edges in E. Formally it can be defined as follows:

$$V(G) = V(G') \cup V(G''), \qquad V(G') \cap V(G'') = \emptyset, \qquad E(G) = E(G') \cup E(G'') \cup E$$

The algorithm first searches for the largest S_{max} subgraph, which has already been processed in D. For this purpose, the algorithm uses the Search algorithm (presented in the next subsection). If S_{max} is isomorphic to the input graph, then the input graph is already stored in the decomposition structure, so the algorithm exits. If there is no sub-graph of the G input graph in the D decomposition, then the algorithm creates S_{max} as a random sub-graph of G, and decomposes S_{max} recursively. After that, the algorithm decomposes the previously not decomposed parts of G (i.e., $G - S_{max}$), and finds the edges between the two subgraphs. Lastly, the algorithm stores the decomposition as a 4-tuple, and also stores the valuations in the D decomposition.

1.2.2 The search algorithm

The task of the search algorithm is to search and return all possible valuations of a configuration graph in a previously created decomposition structure. Algorithm 2 shows the pseudo-code of the search algorithm. In comparison with the algorithm proposed by Messmer et al., the decomposition structure has been modified: our decomposition structure contains valuations separately (rather than substituting the valuations and storing concrete vertices), which makes the structure more compact, as the similarities of the sub-graphs are searched in the *structure* of the graph and not in the concrete vertices.

SearchConfigurationGraph

input : D decomposition, I configuration graph
output : R set of subgraph isomorphism matches

let P the set of all graphs in D: $P \leftarrow \bigcup_{i=1}^{n} \{S_i, S'_i, S''_i\}$
foreach $S \in P$ **do**
  mark S as unsolved;
end
foreach $S = (V_S, E_S, \mu_S, \nu_S) \in P$ *with* $|V_S| = 1$ **do**
  mark S as solved
  $F_S \leftarrow \text{VertexTest}(V_S, \mu_S(V_S), G_I)$
  if $F_S = \emptyset$ **then**
    mark S dead for all valuations
  else
    foreach m *valuation* $\in D$ **do**
      merge F_S and m for V_S as $F_{m,S}$
      if $F_{m,s} \neq \emptyset$ **then**
        mark S alive for m
      else
        mark S dead for m
      end
      associate $F_{m,S}$ subgraph isomorphism to S for m
    end
  end
end
while *there are* $S \in P$*, where* S *is unsolved* **do**
  $X \leftarrow$ randomly one of $(S, S_1, S_2, E) \in D(B)$, where S is unsolved and S_1, S_2 is solved and has at least one alive m valuation
  if X *is empty* **then**
    exit while
  end
  $N \leftarrow$ set of m which are alive in S_1 and S_2
  mark S as solved
  foreach $m \in N$ **do**
    $F_{m,1} \leftarrow$ subgraph isomorphisms of S_1
    $F_{m,2} \leftarrow$ subgraph isomorphisms of S_2
    $F_{m,S} \leftarrow \text{Combine}(S_1, F_{m,1}, S_2, F_{m,2}, E, G_I)$
    if $F_{m,S} = \emptyset$ **then**
      mark S dead for m
    else
      mark S alive for m
      associate $F_{m,S}$ subgraph isomorphism to S for m
    end
  end
end
$R \leftarrow \emptyset$
foreach G_i *model graph which is alive with any* m **do**
  foreach *alive* $m \in G_i$ **do**
    add subgraphs isomorphism of $G_{m,i}$ to R
  end
end
return R

Algorithm 2. Algorithm for searching subgraph isomorphisms on a decomposition structure

The algorithm first collects all possible graphs in the decomposition into the P graph set. All the graphs are marked as “unsolved”. The first part of the algorithm works with graphs that contain only one vertex. First, it marks the graph as “solved”, as it has been processed. Then the algorithm searches for all possible valuations of the vertex in the configuration graph (this

search is implemented in the VertexTest algorithm, see below). If there is no possible valuations (F_s is empty), then this vertex cannot be matched to the configuration graph, so it has to be marked as “dead” for any valuation. Otherwise the algorithm looks up all the valuations in the decomposition and merges them with the valuation found previously with VertexTest. If the two valuations have contradictions, then the vertex has to be marked “dead” for that valuation. Otherwise it will be marked as “alive”.

The second part of the algorithm processes the graphs with more vertices. Each graph appears in the decomposition structure as a sub-graph node. The algorithm searches for nodes in the decomposition structure, where the node is “unsolved”, but the sub-nodes are “solved” and “alive” for at least one valuation. If there is no node like that, then the algorithm exits. The algorithm marks the found graph as “solved”, and collects the valuations which are alive in both sub-nodes. The algorithm merges the two sub-nodes’ valuations using the Combine algorithm (see below). If the result is an empty set, as the valuations have contradictions, then the node is marked as “dead” for that valuation. Otherwise it can be marked as “alive”. This algorithm runs until there are no more unprocessed (“unsolved”) nodes in the decomposition.

After that the algorithm collects together all valuations marked as “alive” and returns them as a result.

The VertexTest algorithm (Algorithm 3) is the algorithm for searching all possible matches of a vertex in a graph. The algorithm returns a vertex as a match, if the labelling function for that vertex returns a label compatible with the input vertex. A label is compatible with a vertex if the object of the context represented by that vertex has the label as its type, or has the label as one of its supertypes.

VertexTest

input : v vertex with label l and a configuration graph G_I
output : F set of vertex matches

```
1  G_I ← (v_I, E_I, μ_I, ν_I)
2  F ← ∅
3  l = ν(v)
4  foreach v_I ∈ V_I do
5  |  if l is compatible with μ(v_I) then
6  |  |  f(v) ← v_I
7  |  |  F ← F ⋃ {f}
8  |  end
9  end
10 return F
```

Algorithm 3. The vertex test algorithm

The Combine algorithm (Algorithm 4) is used for merging two valuation sets together. This algorithm takes as input two graphs (S_1, S_2), a configuration graph (G_I), labelled edges E and two sets of valuations (F_1, F_2). These valuations are bijective functions, mapping the S_1 and S_2 graphs to the G_I graph. The edges of E are the edges between S_1 and S_2 graphs. To produce the combination of the two valuation sets, the algorithm checks two conditions for all possible combinations. First, the images of the two graphs must be disjoint. Second, it must be ensured that each edge that is specified in the set E is mapped correctly onto edges in G_I and vice versa. It means that for each $e = (v_1, v_2)$ edge in E, there is an $e' = (v'_1, v'_2)$ edge in G_I, where v_1 is mapped to v'_1 by the valuations, and v_2 is mapped to v'_2. If both conditions are satisfied, then the valuations can be combined together, giving us the subgraph isomorphism of the two graphs together to the configuration graph.

Combine

input : S_1, S_2 graphs, G_I configuration graph, set of E edges with label, F_1, F_2 sets of valuations for S_1, S_2 to G_I

output : F set of merged valuations

$S_1 \leftarrow (V_1, E_1, \mu_1, \nu_1)$
$S_2 \leftarrow (V_2, E_2, \mu_2, \nu_2)$
$F \leftarrow \emptyset$
foreach f_1, f_2 *pairs, where* $f_1 \in F_1, f_2 \in F_2$ **do**
  if $f_1(V_1) \bigcap f_2(V_2) = \emptyset$ *and for each edge* $e \leftarrow (v_1, v_2) \in E$ *exists an edge* $e_I = (f_1(v_1), f_2(v_2)) \in E_I$, *with* $\nu(e)$ *compatible with* $\nu_I(e_I)$ *and for each edge* $e_I = (f_1(v_I), f_2(v'_I)) \in E_I$ *between* $f_1(V_1)$ *and* $f_2(V_2)$, *there exists an edge* $e = (f_1^{-1}(v_I), f_2^{-1}(v'_I)) \in E$ *with* $\nu_I(e_I)$ *compatible with* $\nu(e)$ **then**
    $f : (V_1 \bigcup V_2 \rightarrow V_I)$ from $S_1 \bigcup_E S_2$ to G_I where $f(v) = f_1(v)$ if $v \in V_1$ and $f(v) = f_2(v)$ otherwise
    $F \leftarrow F \bigcup \{f\}$
  end
end
return F

Algorithm 4. The combine algorithm

This algorithm was originally published in [Messmer 2000]. It is applied here with a small modification: in our version an edge can be mapped from the E labelled edge set to an edge in the configuration graph, if the labels are compatible.

The compatibility of two edges is defined by the mapping between the test requirements and the test trace, since the edges in the requirement graphs (derived from requirement context fragments) may represent abstract relations (e.g., *nearby*), while the edges in the configuration graph (derived from the test trace) represent concrete values (e.g., the distance is 4 units). Note that the mapping from concrete values to abstract relations is performed by the pre-processing of the test execution trace (which restores the abstract relations that are relevant in the requirement scenarios) this way the compatibility relation can be easily checked.

1.3 Matching scenarios using observer automata

As presented earlier, each requirement scenario is mapped to an observer automaton. The test oracle's task is to run these automata to match the provided test trace and return an evaluation for each scenario (whether it is passed, violated or inconclusive).

1.3.1 Observer automaton definition

The detailed definition of the observer automaton can be found in D4.2.1. The automaton is a variant of a nondeterministic finite automaton. The main difference is the presence of two transition classes:

- *Normal transition*, which can have (1) a guard condition, which enables the transition; (2) an update, which can assign values to variables; and (3) the description of event/action processing.
- *Context change transition*, which can have (1) a guard condition; (2) an update associated with the transition, and (3) context switch information. This transition represents a switch to a new context fragment in the requirement scenario.

As the automaton is nondeterministic, one state may have multiple subsequent states. There are three different types of states defined:

- *Trivial accepting state:* A trivial accepting state represents a state, which is generated for the "trigger" (precondition) part of the requirement. If an automaton ends in a trivial accepting state then that run results in an inconclusive verdict for the requirement

represented by the automaton (i.e., the trigger part of the requirement does not match the test execution trace).

- *Stringent accepting state:* Stringent accepting states belong to that part of the requirement scenario, where the behaviour described by the trigger part (precondition) is already met and the assertion part is also matched. If an automaton ends in a stringent accepting state, then the run passed the requirement.
- *Rejecting state:* Rejecting states belong to that part of the requirement scenario, where the behaviour described by the trigger part (precondition) is already met but the assertion part is not matched. If an automaton ends in a rejecting state, then a requirement is violated.

1.3.2 Execution

The test oracle's task is the execution of the observer automata generated from the requirement scenarios considering the operational semantics of the automata. Each automaton shall be executed simultaneously and each possible run shall be considered. For this purpose, we defined an *automaton execution context*, which uses the context matching algorithm presented in section 1.2. The core algorithm of the execution context is presented as Algorithm 5.

Automaton execution context

input : A set of automata to be executed, C referenced context models, T trace to be validated
output : Trace matches

```
1  I ← instantiate all automata from A with initial location
2  set TP trace pointer to the beginning of T trace
3  R ← empty set of trace matches
4  while TP is not after the last entry of T trace do
5      foreach automaton ∈ I do
6          foreach p transition path ∈ enabled context change transition paths in automaton from
           current location with TP and C do
7              S_automaton,p ← create new automaton instance from automaton
8              foreach t transition ∈ p do
9                  S_automaton,p ← operate t on S_automaton,p
10             end
11             add S_automaton,p to I
12         end
13     end
14     foreach automaton ∈ I do
15         remove automaton from I
16         if no normal transition is enabled in automaton then
17             store results in R
18         else
19             foreach t transition ∈ enabled normal transitions of automaton do
20                 if TimeStamp_TP fires dynamic events then
21                     C_TP,automaton ← handle dynamic event of TP on C
22                     if C_TP,automaton can't be matched to T_TP then
23                         go to next automaton of I
24                     end
25                 end
26                 modify C_automaton with TimeStamp_TP and C
27                 if t can be matched to T_TP then
28                     S_automaton,t ← operate transition on S_automaton
29                     add S_automaton,t to I
30                 else
31                     store results in R
32                 end
33             end
34         end
35     end
36     TP ← advance TP
37 end
38 return R
```

Algorithm 5. The core algorithm for automaton execution context

The operations referenced in the algorithm (operating t, storing the results, handling dynamic events) are detailed in the following paragraphs.

The execution context – EC for short – receives as inputs all generated observer automata from the requirement scenarios, the referenced context model and the trace to be checked. The EC then executes all automata in a parallel way and collects the information related to the run of these automata as the results of matching. These results will be used for the calculation of coverage metrics. The results are calculated at the end of each run, with the following algorithm (Algorithm 6).

Storing the results of one automaton execution

1 **if** $Loc_{automaton}(Source(t))$ *is rejecting* **then**
2 | $result \leftarrow violated$
3 **else if** $Loc_{automaton}(Source(t))$ *is trivial accepting* **then**
4 | $result \leftarrow inconclusive$
5 **else if** $Loc_{automaton}(Source(t))$ *is stringent accepting* **then**
6 | $result \leftarrow passed$
7 **end**
8 store $result$ in R

Algorithm 6. Storing the results of one automaton execution

The algorithm's main cycle can be considered as a step with two separate parts:

- The first part collects context change transition paths (with satisfied guard conditions) from the current location (let us note it here with $CCP(L)$, where L is the current location of the automaton under investigation). The definition of the context change paths is the following:

  $CCP_0(L) = \{(L)\}$
  $CCP_1(L) = CCP_0(L)\ \tilde{\cup}\ \{paths\ with\ 1\ additional\ ctx.change\ to\ each\ path\ in\ CCP_0(L)\}$
  $\vdots$
  $CCP_i(L) = CCP_{i-1}(L)\ \ \tilde{\cup}\ \{paths\ with\ 1\ additional\ ctx.change\ to\ paths\ in\ CCP_{i-1}(L)\}$
  $CCP(L) = CCP_k(L), if\ |CCP_k(L)| = |CCP_{k+1}(L)|$

  $Where\ \tilde{\cup}\ is\ an\ union\ operator, which\ merges\ all\ paths\ with\ the\ same\ destination.$

  The $CCP(L)$ represents all the paths, which can be traversed without the need of processing any action, event or context change from the trace. A path is possible, if all context change transitions in the path can be matched (after each other) to the current context in the test trace. For all possible paths, a new automaton instance shall be added to the set of automata. The matches are calculated with the decomposition based context matching algorithm presented earlier. As the matching of one path can result in several valuations from the contexts in the requirements to the context in the trace, it is possible to add more automaton instances to the set of automata, than the number of paths in $CCP(L)$.

- After the context change transitions are processed, the trace shall be considered. The trace can contain context change information – when the context of the test execution is changed – or event/action information – when the SUT received an event or sent an action. The EC checks whether the subsequent information on the trace can be matched with the next enabled transition of the automaton. For this matching, the EC uses the previously presented context matching algorithm in case of context changes, or a simple compatibility check in case of actions and events.

After each step, the algorithm advances the trace pointer, until it reaches the end of the trace. After the last element of the trace is processed, the trace is evaluated, and the results are computed.

Before the EC checks whether the next trace element can be matched to the transitions in all observer automata, it also searches for dynamic events from the initial context. Each dynamic event has an exact timestamp that specifies the occurrence of the event. If the

timestamp of the next trace element is larger than the timestamp of an unprocessed dynamic event, then the object associated with the dynamic event shall be added to (or removed from) the context in case of an Appear action (or Disappear action, respectively). The timestamp of the dynamic events is evaluated by considering the elapsed time since the beginning of matching the requirement (and not since the beginning of the trace).
The approach to define “local” timestamps with respect to a requirement makes the scenario language easier to use with the search based test generation, but makes the evaluation be a more complex task, since the context fragments can be different for each automaton (as these contexts depend also on the point of time when the automaton was started, this way on the occurrence of dynamic events that were specified in the initial context fragment). To handle the problem, the EC contains a Context Fragment Manager component, which calculates the context fragment for each automaton by keeping track of the effective dynamic events and the related objects.

1.3.3 Example run of the algorithm

In this section we present a short example for the algorithm of the execution context that matches a scenario using an observer automaton.

In this example we take one requirement, from D4.2.1. The requirement specifies that if the robot is near to a human or an animal, then a nearby alert shall be emitted (Figure 12).

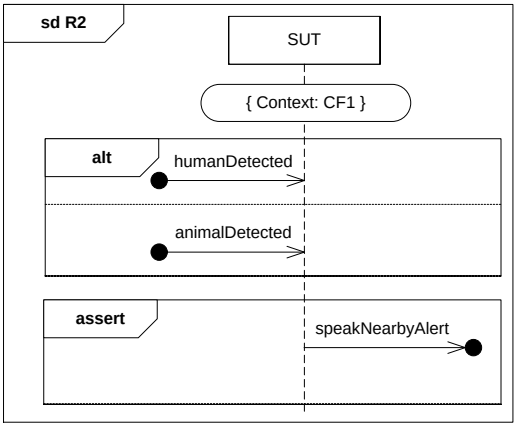


Figure 12 Example requirement from D4.2.1.

The unwinding algorithm (described in D4.2.1) generates the observer automaton presented on Figure 13.

The test oracle needs a context metamodel, which is the metamodel of the referenced context fragments in the requirements. In case of our requirement, the only referenced context fragment is CF1, which is presented on Figure 14.

For this example, the trace of a test run is presented on Figure 15, It includes context changes (the two configuration contexts are presented on Figure 16 and Figure 17), events and actions.

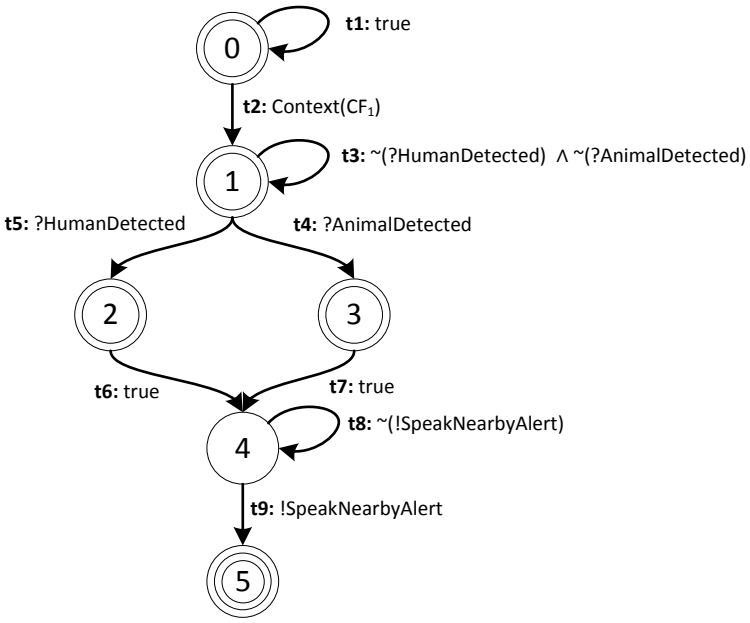


Figure 13 Observer automaton for the requirement on Figure 12.

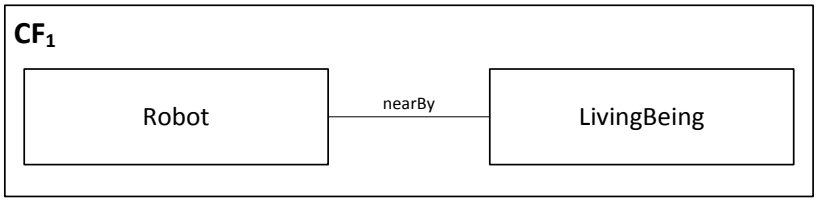


Figure 14 Context fragment 1 for the example requirement

Figure 15 Example test trace

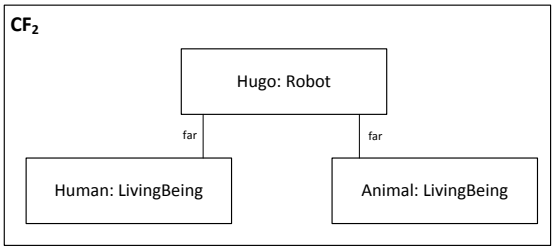


Figure 16 Context fragment 1 for the example trace

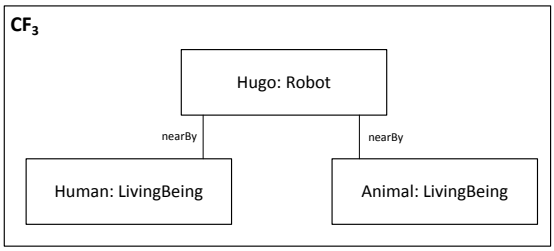


Figure 17 Context fragment 2 for the example trace

The steps of matching the requirement scenario (represented by the observer automaton) to the test trace are presented on Figure 18 and Figure 19. The algorithm checks all possible automaton states with all possible valuations as explained below.

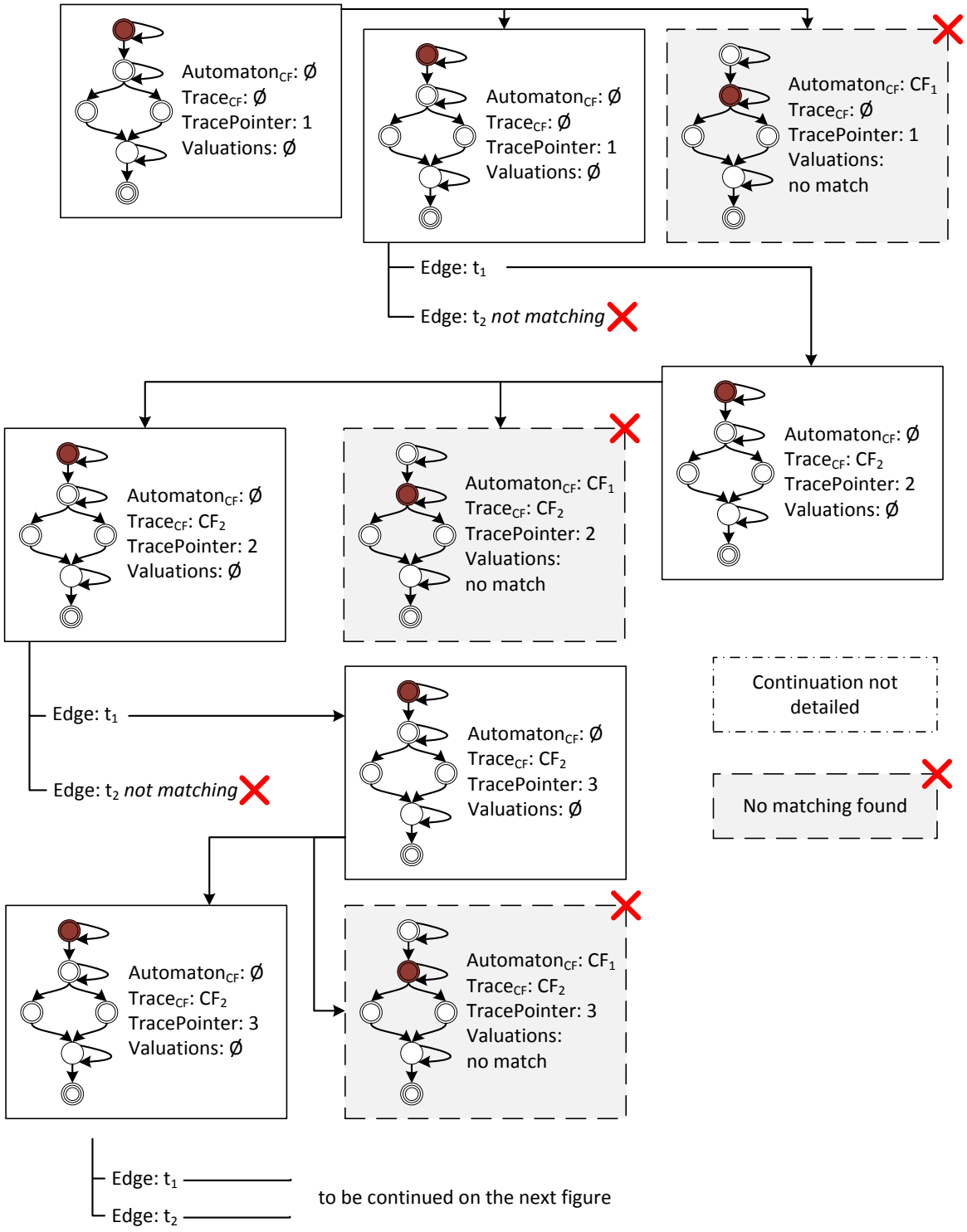


Figure 18 First part of the example run

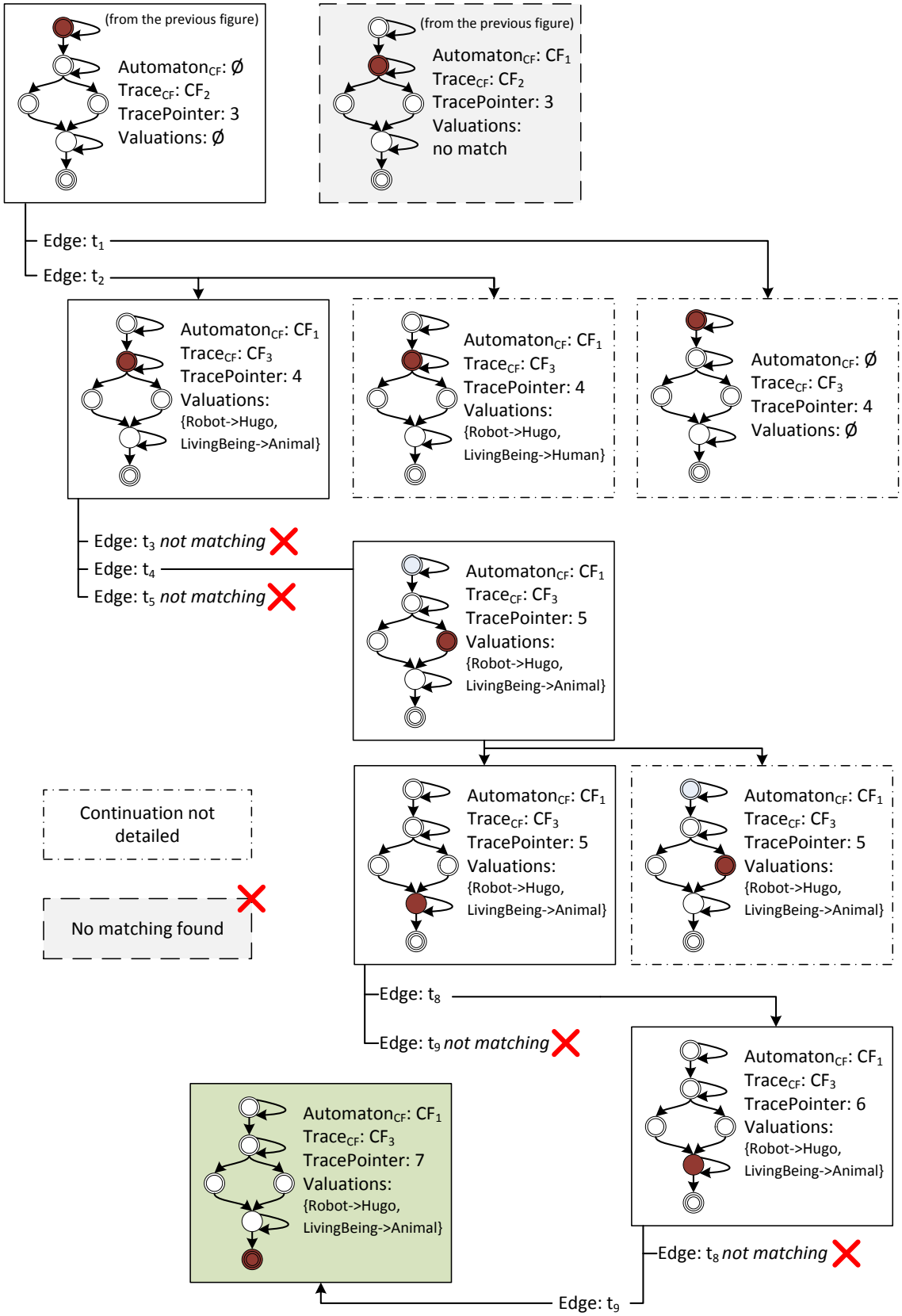


Figure 19 Second part of the example run

The algorithm starts with a clean instance of the observer automaton, which means that the automaton is in its initial location with an empty context fragment, it has no valuations and

the trace pointer points to the initial location of the trace where the context fragment of the trace is empty.

As a first step, the algorithm calculates the CCP set, which contains the first and second locations of the observer automaton. The second location cannot be reached, as the automaton context (CF_1) is not compatible with the empty context of the trace. Thus, the algorithm keeps the automaton instance in the initial location, and checks the outgoing edges (t_1, t_2). Only t_1 is enabled (as its guard condition is true), so the automaton stays in the initial location, the context fragments are unchanged, but the trace pointer is advanced to 2.

Then the algorithm repeats this step (firing edge t_1) twice, until it finds a context fragment in the trace which is compatible with CF_1. As the calculation reaches CF_3 (after the trace pointer advances to 4), it finds two possible valuations ({Robot → Hugo, LivingBeing → Human} and {Robot → Hugo, LivingBeing → Animal}), this way it creates two observer automaton instances. These instances follow similar steps, so the figures above present only one of them.

As the CCP is an empty set at location 1, the algorithm checks the edges. Only t_4 is enabled, so the automaton steps to location 3. As it has an edge with a true guard, the CCP contains both location 3 and location 4. At location 4, the automaton skips the HumanDetected message by firing t_8 and then fires t_9 matching the SpeakNearByAlert message.

At that point (location 5) the algorithm finds that in this run the test trace passed the requirement .The other runs are inconclusive.

References

[Connelly 2006] Connelly, J., et. al. Hong, W., Mahoney, R., Sparrow, D.: Challenges in Autonomous System Development. In: Proc. of Performance Metrics for Intelligent Systems, 2006

[Messmer 2000] B. T. Messmer and H. Bunke. “Efficient Subgraph Isomorphism Detection: A Decomposition Approach”. IEEE Transactions on Knowledge and Data Engineering, Vol. 12 No. 2, pp 307-323, March 2000. DOI: 10.1109/69.842269

[Micskei 2012] Micskei, Z., Szatmári, Z., Oláh, J., Majzik, I.: A Concept for Testing Robustness and Safety of the Context-Aware Behaviour of Autonomous Systems. In: Proc. 1st Int. Workshop on Trustworthy Multi-Agent Systems, Springer LNCS 7327, pp. 504–513, 201

[R3-COP] R3-COP: Resilient Reasoning Robotic Cooperating Systems. ARTEMIS-2009-1 Project No. 100233, http://www.r3-cop.eu/