GraphSeq: a Graph Matching tool for the extraction of mobility patterns

Minh Duc Nguyen1,2, Hélène Waeselynck1,2, Nicolas Rivière1,2
1 CNRS; LAAS; 7 avenue du Colonel Roche, F-31077 Toulouse, France
2 Université de Toulouse; UPS, INSA, INP, ISAE; LAAS; F-31077 Toulouse, France
e-mail: {mdnguyen, waeselyn, nriviere}@laas.fr

Abstract—Mobile computing systems provide new challenges for verification. One of them is the dynamicity of the system structure, with mobility-induced connections and disconnections, dynamic creation and shutdown of nodes. Interaction scenarios have then to consider the spatial configuration of the nodes as a first class concept. This paper presents GraphSeq, a graph matching tool for sequences of configurations developed in the framework of testing research. It aims to analyze test traces to identify occurrences of the successive spatial configurations described in an abstract scenario. We present the GraphSeq algorithm, as well as first experiments using randomly generated graphs, outputs from a mobility simulator, and test traces from a case study in ad hoc networks.

Keywords-Algorithm; graph matching; testing; mobile computing systems; scenarios

I. INTRODUCTION

Mobile computing systems involve devices (handset, PDA, laptop, intelligent car...) that move within some physical areas, while being connected to networks by means of wireless links (Blue-tooth, IEEE 802.11, GPRS...). Applications in such systems differ from ones in "traditional" distributed systems in many aspects: frequent connections and disconnections of mobile nodes, communication with unknown partners in a local vicinity, context awareness. These novelties raise new challenges for the testing technology.

One issue is how to express interaction scenarios in mobile settings. Scenario descriptions are quite useful to support various test-related activities, such as the representation of requirements [1], of test purposes [2], of test cases [3], or of execution traces [4]. A number of scenario languages have been proposed in the framework of distributed systems, such as UML Sequence Diagrams [5] or Message Sequence Charts (MSCs) [6]. However, they are not sufficient to account for mobile settings. We, therefore, proposed extensions to fill these gaps [7].

An extension is to consider the spatial relations between nodes as a first class concept. Labeled graphs are used to depict the spatial configurations of a scenario. As a result, the analysis of test traces to identify occurrences of a scenario involves graph matching problems. It led us to develop GraphSeq, a graph matching tool for sequences of configurations.

This paper is organized as follows. Section II explains the background to the development of GraphSeq. Section III introduces the principle of GraphSeq. Based on graph homomorphism building, our tool reasons on sequences of graphs (i.e., sequences of spatial configurations). Then, Section IV presents the algorithms implemented in GraphSeq. First experiments with GraphSeq are reported in Section V, using randomly generated graphs, outputs from a mobility simulator, and test traces from a case study. Section VI concludes the paper.

II. BACKGROUND OF GRAPHS

Scenario languages typically consider interactions in terms of the (partial) orders of communication events. The events are represented by drawing lifelines for the involved entities, and by graphically displaying communication between entities. Some events are ordered while others may interleave in a non-deterministic way, hence yielding partial orders. As argued in [7], when it comes to represent scenarios for mobile computing systems, the focus on event ordering fails to capture other important aspects, such as the spatial configuration of entities.

To illustrate this, let us discuss a case study we performed, a partitionable Group Membership Protocol (GMP) for ad hoc networks [8]. A GMP is a fundamental service lying at the heart of fault-tolerant systems. It aims to maintain a consistent view of who is in the group, in spite of the faults that may affect some nodes. The goal of this GMP is to offer a membership service that accommodates the disconnections induced by mobility (when mobile nodes get out of range). Each group is managed by a leader that may decide to merge with other groups, or to split its group. Decision is based on the notion of safe distance, where the safe distance is strictly lower than communication range. If two nodes are “close enough”, this will prevent motion-induced disconnection for some time. A group is then safe if any two members are connected via a path along which all consecutive hosts are at a safe distance (multi-hop communication is allowed). When two groups can merge into a single safe group, they have to do so. When a group is no longer safe, it has to split. Nodes are kept informed about their relative position by “hello” messages that carry location information.
Fig. 1 provides a representation of a fail scenario found by testing the GMP [9]. The representation resembles a classical scenario, but with some extensions to account for the mobile setting. Note that the syntax used in Fig. 1 is unimportant, we focus here on the concepts.

The event view (Fig. 1b) shows a specific interleaving of split and merge operations of the protocol. The scenario is triggered by the “hello” message broadcasted by n2 after a change in spatial configuration. Specifically, n2 is no longer at a safe distance from n4, while getting close to n1. The configuration change is shown in Fig. 1a. Both the “GetLeader” message from n1 (initiating a merge) and the “GroupChange” message from n4 (performing the split) are causally related to the “hello” message: when receiving the location data from n2, the other nodes get aware of a change in the topology. At the end of the scenario, a violation of a property of the GMP occurs on node n2. Without entering into the details of the scenario, let us insist on the fact that both the ordering of messages and the topology of nodes are important for the failure to occur. In particular, it is important that:

- The change in spatial configuration breaks the transitive safe path between n4 and n3 (it is precisely the inconsistent treatment of n3 in the group change operations received by n2 that will induce the property violation);
- When asked about its leader address, n2 replies before being informed about the split.

As can be seen from Fig. 1, the description of such a scenario needs two connected views, a spatial view (describing the successive topological configurations of the system nodes) and an event view (describing communication and configuration change events). The spatial view consists of a set of labeled graphs, where labels attached to vertices and edges represent relevant attributes of system nodes and of communication links between nodes. In the example, nodes are merely characterized by their id, but tuples of labels may be allowed for applications needing a richer representation of contextual attributes. The event view makes it explicit which communication event occurs in which spatial configuration. It offers special symbols to represent broadcast in a local vicinity (see the hello message). Configuration change events are introduced as global synchronization points.

In our work, scenario descriptions are intended to allow the off-line analysis of execution traces, collected on a test platform with simulation and instrumentation facilities [7, 9]. The comparison of scenarios and traces may serve different objectives, depending on the role of the scenario. For example, requirements scenarios are used to check whether key properties are violated during testing. This offers an automated solution to the test oracle problem, namely how to determine acceptance or rejection of the test outputs. Test purposes are used to check whether desired fragments of behavior are covered at least once during testing. Such treatments are classical when testing distributed systems, and involve comparing the orders of events in the scenario and the trace. In our case, graph algorithms are also needed to identify occurrences of a target scenario. This is due to the need to determine whether the physical nodes appearing in the trace can match symbolic nodes appearing in the spatial view of the scenario. We, therefore, have developed GraphSeq to fulfill this need.

### III. Principle of GraphSeq

GraphSeq takes as input two sequences of graphs:

- A sequence \( P_0, \ldots, P_{m-1} \) of \( m \) graphs coming from a scenario description, called pattern graphs,
- A sequence \( C_0, \ldots, C_{n-1} \) of \( n \) graphs extracted from an execution trace, called concrete configuration graphs.

It computes the set of all matches, where a match identifies a subset of concrete nodes that exhibit the searched sequence of patterns for a certain interval of time. Fig. 2 gives the visual intuition of this. \( P_0 \) occurs as

---

(a) Spatial view

(b) Event view

Figure 1. Example of scenario for the GMP case study
a subgraph of the system in concrete configurations $C_1$, $C_2$. Then, there is a system configuration change yielding $P_1$, and finally $P_2$ occurs until the system changes to concrete configuration $C_7$. Note that a given pattern may persist for several successive concrete configurations. The temporal window computed by GraphSeq will be later used when processing the event view of the scenario. For example, if a scenario description involves a change event $C_i \rightarrow C_j$, and ending at the date of configuration $C_k \rightarrow C_l$. The match let us also know which concrete nodes should be the sender and receiver of $msg$.

The computation of matches requires a core functionality to compare two graphs, e.g., to determine whether $P_0$ occurs as a subgraph of $C_i$. Technically, this calls for graph homomorphism building, a problem well studied in the literature.

Based on such a facility, there is a need to reason on sequences of graphs, as illustrated in Fig. 2: this is the specific contribution of GraphSeq. Before introducing the principle of the sequential reasoning, we recall the definition of graph homomorphism and briefly present an example of graphs, as illustrated in Fig. 2: this is the specific contribution of GraphSeq. Before introducing the principle of the sequential reasoning, we recall the definition of graph homomorphism and briefly present an existing tool that GraphSeq uses as a core facility for its computation.

A. Graph homomorphism building

Let $L_V$ and $L_E$ denote sets of labels for vertices and edges, and let $G = (V, E, \lambda, \mu)$ denote a graph structure, where:

- $V$ is the set of vertices,
- $E \subseteq V \times V$ is the set of edges,
- $\lambda : V \rightarrow L_V$ is a function assigning labels to vertices,
- $\mu : E \rightarrow L_E$ is a function assigning labels to edges.

A graph homomorphism is a mapping between two graphs that respects their structure. It can be defined as follows.

**Definition.** Let $G_1 = (V_1, E_1, \lambda_1, \mu_1)$ and $G_2 = (V_2, E_2, \lambda_2, \mu_2)$ be two graphs. An injective function $f : V_1 \rightarrow V_2$ is a graph homomorphism from $G_1$ to $G_2$ if:

- $\lambda_1(v) = \lambda_2(f(v))$ for all $v \in V_1$,
- For any edge $e_1 = (v_{i1}, v_{i2}) \in E_1$, there exists an edge $e_2 = (f(v_{i1}), f(v_{i2})) \in E_2$, such that $\mu_1(e_1) = \mu_2(e_2)$.

This definition captures the idea of $G_1$ being matched by a subgraph of $G_2$. For us, $G_1$ is expected to be a pattern graph from a scenario, and $G_2$ a concrete configuration graph from an execution trace. Note that our internal encoding of graphs considers disconnection as an edge with a specific label, so that vertices that are unconnected in a pattern must also be unconnected in a matching concrete configuration.

Classical algorithms for graph homomorphism building can be found in [10, 11]. However, the basic definitions of graph structure and graph homomorphism need to be extended to fulfill our needs. First, it is convenient to assign labels of vertices in order to allow for a richer representation of node attributes. Second, we need label variables in pattern graphs. For example, a node has symbolic id $n1$ in a scenario description, and we may detect a matching by a physical node “140.93.130.95”, using substitution $n1::= “140.93.130.95”$. To accommodate label variables, a graph homomorphism has now the form $(f, val)$, where $f$ is a mapping and $val$ a valuation that consistently unifies the labels.

To fulfill these needs, GraphSeq uses an existing graph homomorphism tool developed by colleagues at LAAS [12]. Their work addresses the specification of dynamically reconfigurable architectures [13]. Like in our case, this induced matching concerns for graphs with multiple and symbolic labels. The resulting tool is implemented in C++ and can be reused for our purposes. It provides us with facilities to encode graphs with constant labels, variable labels and wildcards (for edges). We made a slight extension to have wildcards for vertices as well.

The syntax of our pattern graphs is exemplified by Fig. 3. Vertices have at most three labels. The first one is mandatory; it is a symbolic id to be matched by the concrete id of a physical node. The other two labels may be used to represent additional attributes of integral types (integers, or enumerated types). Their form may be:

- A constant value from the type. In Fig. 3, the attributes of node $id2$ have constant values 1 and 2.
- A variable name, denoting a value from the type. For example, the first optional attribute of nodes $id1$ and $id3$ must be identical, but their precise value is let unspecified (variable $v1$).
- A wildcard indicating a don’t care value, see e.g. the last attribute of node $id1$.

Edges can be labeled by constant values or wildcards.

![Figure 2. Matching sequences of graphs.](image)

![Figure 3. Graph with various forms of label.](image)
In Fig. 3, the connection type is \{Safe, RangeNotSafe\}. Nodes \textit{id}1 and \textit{id}2 have a safe distance connection; nodes \textit{id}2 and \textit{id}3 are disconnected; we do not care about the connection of nodes \textit{id}1 and \textit{id}3.

The syntax of concrete configuration graphs is simpler, because only constant labels are allowed. Concrete node ids are strings, while the other labels must be values from the appropriate integral type.

The graph homomorphism tool accommodates the above syntax, and allows us to identify instances of a pattern graph in a concrete configuration graph. It also provides convenient facilities to manage the valuation of pattern graphs. Two supplied functions will be mentioned later:

- **MATCHGRAPHS** \((G_1; \text{Graph}, G_2; \text{Graph})\) that returns the list of all homomorphisms \((f, \text{val})\) from \(G_1\) to \(G_2\).
- **VALUATEGRAPH** \((G; \text{Graph}, V; \text{Valuation})\) that returns a copy of \(G\) with the vertex labels rewritten according to the valuation \(V\).

GraphSeq uses these functions as core facilities to search for matches in the case of sequences of graphs.

### B. Comparing sequences of graphs

While the problem of comparing two graphs has been extensively studied, there has been relatively little work on the comparison of sequences of graphs (see [14] for a survey on graph matching). Existing work cannot be directly reused for our purposes, because it addresses simpler matching problems (e.g., search for graph subsequence "g1 g2" in "g0 g1 g2 g3") or different ones (e.g., infer predictions from the analysis of past time series of graphs). Examples of different problems in the field of dynamic network analysis can be found in [15]. The closest work we found is for the analysis of video images. In [16], the authors search for sequences of patterns (called pictorial queries) into a sequence of graphs extracted from video images. A difference with our work, however, is that the patterns do not involve label variables, and that a pattern node corresponds to at most one object in an image. Hence, to the best of our knowledge, the sequential reasoning implemented by GraphSeq is a new contribution.

In our case, there may be several instances of pattern \(P\) in concrete configuration \(C\), each corresponding to a different valuation of \(P\) variables. In particular, there may be several instances of the first pattern \(P_0\). Each candidate valuation for \(P_0\) may in turn yield several possibilities for the rest of the sequence, with alternative valuations for the variables that are new in \(P_1, \ldots, P_n\). Variables that are not new are expected to keep the same value as previously.

The key of the sequential reasoning is to explore all possibilities, while retaining consistent valuation choices across the sequence, and properly identifying the transition from one pattern to the next. Let us present the properties expected from the search results.

Given two graph sequences \(P_0, \ldots, P_m\) and \(C_0, \ldots, C_n\) to be compared, GraphSeq computes the set of all matches, where a match has the following data structure:

```
Match : Valuation val
int index[0..m]
```

The valuation \(\text{val}\) assigns a concrete value to every label variables appearing in the sequence of pattern graphs. It is a set of pairs \((\text{varName}, \text{value})\) such that each variable name appears in exactly one pair. In particular, each symbolic identifier of a system node is assigned a unique value by a match.

The index table encodes the temporal window \(\text{[start date, end date]}\) for each pattern. It is defined as follows:

- \(\text{index[0]}\) is the start date of the matching for \(P_0\). That is, the matching starts at \(C_{\text{index[0]}}\).
- For \(i > 0\), \(\text{index[i]}\) is the end date of the matching for \(P_{i-1}\). That is, the matching ends after \(C_{\text{index[i]}}\).

For example, the duration of patterns in Fig. 2 is encoded as follows:

```
start date   end date   end date   end date
\downarrow   \downarrow   \downarrow   \downarrow
data 4 . . 6
index: 1 2 3 6
```

The encoding means that \(P_0\) starts when the system exhibits configuration \(C_1\) and ends after the system leaves \(C_2\). The start date for pattern \(P_1\) is implicit: it can be retrieved as \(l + \text{end date of } P_0\), i.e. 3. The end date for \(P_1\) is then explicitly given, yielding a temporal window \(\text{[3, 3]}\). Likewise, the temporal window for \(P_2\) is \(\text{[4, 6]}\).

Let \(M\) be a match returned by GraphSeq. Let \(\text{[s, e]}\) be the temporal window for each pattern \(P_i\), as indicated by \(M.\text{index}\). Obviously, we expect that each concrete configuration \(C_{s_0}, \ldots, C_{e_0}\) contains the instance of \(P_i\) determined by valuation \(M.\text{val}\).
**Property 1** – \( \forall i \in [0, m-1], \forall k \in [s_i, e_i] \), there exists a mapping \( f \) such that:
**MATCH**\( \text{GRAPHS} \) (Valuate \( (P_i, M.val), C_k) = \{(f, \emptyset)\}

Note that for each pair \( (i, k) \), we are comparing two graphs without label variables, hence the empty valuation for the resulting homomorphism. Also, if \( f \) exists then it is necessarily unique because the valuation of nodes ids leaves no choice for the homomorphism.

A consistent valuation of variables, and more specifically of symbolic node ids, must also account for nodes that dynamically appear and disappear. Fig. 4 shows an example of patterns with node creation. For the sake of simplicity, the vertices of the graphs are labeled by their id, and we omit all other labels. Pattern \( P_i \) introduces a new node, with id \( n2 \), that was not present in \( P_o \). The valuation of \( n2 \) must correspond to a concrete node that never showed up in previous configurations matching \( P_o \). For example, the following match should not be retained (although it satisfies Property 1):

| val: \{ (n1, “1”), (n2, “2”) \} |
| Index: t | t+1 | t+2 |

Here, concrete node “2” cannot play the role of a new node \( n2 \). However, the following match is OK:

| val: \{ (n1, “1”), (n2, “2”) \} |
| Index: t+1 | t+1 | t+2 |

The sequential reasoning must thus introduce consideration for nodes that are forbidden in certain configurations. Let \( F.Id_i \) be the set of concrete values (determined from \( M.val) \) for symbolic node ids that do not appear in \( P_i \) but appear in some \( P_j, j \neq i \).

**Property 2** – \( \forall i \in [0, m-1], \forall k \in [s_i, e_i] \), \( C_k \) has no vertex with id label in \( F.Id_i \).

Finally, we require that each temporal window for pattern matching is maximal, that is, \( s_i - l \) would not be a start date with valuation \( M.val \), and \( l + e_i \) would not be an end date. The following match is not maximal, because the temporal window for \( P_o \) could start earlier:

| val: \{ (n1, “1”), (n2, “3”) \} |
| Index: t+1 | t+1 | t+2 |

**Property 3** – \( \forall i \in [0, m-1] \), let \( P_i = \text{Valuate} \ (P_i, M.val) \). Temporal window \( f_{s_i, e_i} \) is maximal, that is:

- \( s_i = 0 \), or there is no homomorphism from \( P_i \) to \( C_{s_i, e_i} \), or \( C_{s_i, e_i} \) contains a vertex with id in \( F.Id_0 \).
- \( e_i = n-1 \), or there is no homomorphism from \( P_i \) to \( C_{1+e_i} \), or \( C_{1+e_i} \) contains a vertex with id in \( F.Id_0 \).

Property 3 implies that start and end dates really correspond to configuration change events. A scenario with two successive identical patterns would yield no match. Transition from \( P_i \) to \( P_{i+1} \) must be triggered by a change in connectivity or attributes of concrete nodes, the introduction of a node, the suppression of a node, or any combination of these.

With the introduction of new nodes, the computation of start and end dates may become tricky. For example, in Fig. 4, if concrete node “1” plays the role of \( n1 \), the end date of \( P_o \) may be \( t+1 \) (before appearance of concrete nodes “2” or “3”), \( t+2 \) (before appearance of “4”) or later. Depending on which choice is taken, candidate start dates for \( P_o \) may, or not, yield a maximal temporal window.

The purpose of GraphSeq is to explore alternative choices, so as to build the set of all matches satisfying properties 1, 2 and 3.

IV. **ALGORITHM OF GRAPHSEQ**

The implementation of GraphSeq is about 2K lines of C++ code (not including the graph homomorphism tool). A detailed presentation of the algorithm can be found in a technical report [17]. For space constraints, we only provide a high-level view of the algorithm in this paper. We then discuss complexity issues.

A. **High-level view of the search for matches**

In order to gradually build matches, GraphSeq uses an intermediate data structure called \( \text{PartialMatch} \):

<table>
<thead>
<tr>
<th>PartialMatch : Valuation val</th>
</tr>
</thead>
<tbody>
<tr>
<td>int index[0..m]</td>
</tr>
<tr>
<td>int depth</td>
</tr>
</tbody>
</table>

**PartialMatch** is thus like **Match**, but with an additional \( \text{depth} \) field. The depth value gives the number of patterns that have been successfully matched so far. A depth of \( i \) indicates that patterns \( P_0, ... P_{i-1} \) have been matched, but \( P_o, ... P_{m-1} \) are still to be processed. Back to the example in Fig. 4, a partial match could be:

| val: \{ (n1, “1”) \} |
| index: t+1 | t+1 | -1 |
| depth: 1 |

The valuation \( \text{val} \) assigns a value only to variables of the processed patterns. In \( \text{index} \), the spurious value -1 means that end dates of unmatched patterns are not determined yet.

A partial match of depth \( i \) is extended by the processing of the next \( P_i \) yielding a partial match of depth \( i+1 \). When a partial match has been extended up to depth \( m \), then a complete match has been found.

GraphSeq uses depth-first search (DFS) to extend partial matches. That is, if a partial match has several possible extensions, the tool will explore as far as possible along each branch before backtracking. The DFS control structure is shown in Fig. 5. The algorithm involves steps...
indicated in bold characters: preprocessing of pattern data, creation of partial matches of depth 1, extension of partial matches, final check before retaining a match.

\[
\text{Preprocess pattern data}
\]

Let L be an empty stack of PartialMatch elements
For (s=0; s<n; s++)
    build all partial matches of depth 1 with start date s, push each of them in L

While L is not empty
    Let pm = pop (L)
    If (pm.depth < m)
        Build all one step extensions of pm, push each of them in L
    Else
        // found
        If (FinalCheck(pm))
            Write pm.val and pm.index in output file
        Endif
    Endif
End While
End For

\[\text{Figure 5. DFS control structure of GraphSeq.}\]

B. Steps for searching matches

A preprocessing step is performed before the search is entered. Its first task is to identify nodes that are forbidden in some configurations, hence preparing the checking of Property 2. They can be nodes that already appeared in the past or will appear for the first time in the future. For the example in Fig. 4, the analysis determines that configurations matching \( P_0 \) should not contain the node playing the role of \( n2 \) in the future. The second task of the preprocessing step is to analyze the differences between two successive patterns. It is used to facilitate the determination of start and end dates of the patterns (see Property 3). The analysis distinguishes two classes of transitions \( P_i \rightarrow P_{i+1} \):

- after the transition, concrete configurations matching \( P_{i+1} \) cannot match \( P_i \).
- after the transition, concrete configurations matching \( P_{i+1} \) can match \( P_i \) as well.

In the first class, there is only one possible end date for \( P_i \): the date \( j \) such that \( C_{j+1} \) no longer matches \( P_i \). The second class occurs when the appearance of a node is possibly the only trigger of the transition. Then, the search must consider a set of candidate end dates, namely all intermediate dates before \( P_i \) ceases to be matched. For example, in Fig. 4, \( P_0 \) could end at \( t+1, t+2 \) or later.

When the search is entered, we first build partial matches of depth 1. The corresponding algorithm is described in Fig. 7a, with auxiliary functions in Fig. 7b. Given a candidate start date \( s \), the search is initialized by looking at all possible instances for \( P_0 \). Deciding whether \( s \) is really a start date depends on ForbiddenNodes (0), the forbidden nodes at \( P_0 \). If this set is empty, we do not retain \( s \) as a candidate start date if \( C_{s+1} \) already matched \( P_0 \) with the proposed valuation. But if the set is not empty, we always retain \( s \). For example, in Fig. 4, \( s=t+1 \) is retained as a candidate start date for a partial match with valuation \( \{n1, l\} \) because ForbiddenNodes (0) is not empty. Decision of whether \( s \) fulfills Property 3 will be taken by the final check (see the general control structure in Fig. 5), when all symbolic ids have a valuation in ForbiddenNodes (0), and in particular node \( n2 \). As regards the end date, the preprocessing step tells us whether one or several dates should be considered. The function ComputeEnddate() detects an end of matching when it is no longer possible to find an homomorphism. Note that the last two parameters of the function would also allow end-of-matching detection in the following case: a concrete configuration graph contains forbidden nodes. This detection mode will be used when computing the one-step extensions of a partial match. It is not used here (the relevant parameters are set to \( \emptyset \) and NULL by the call statement), because no concrete id of forbidden nodes is known at this stage. A different partial match is built for each date, e.g., \( s=t+1 \) and valuation \( \{n1, l\} \) yield several partial matches with end date \( t+1, t+2, \ldots \).

The search then gradually extends a partial match of depth \( d \) to produce partial matches of depth \( d+1 \), until \( d=m \) or the extension fails. Here, the start date is fixed: it comes just after the end of matching of the previous pattern. The processing of the end date is as already explained above. The extension accounts for forbidden nodes. Their concrete id is known if they appeared in the previous patterns, and it is ensured that they are not present in concrete configurations matching the current pattern. Also, if the current \( P_d \) introduces new nodes to valuate, it is checked that they never showed up since the start date of \( P_0 \).

When we get a complete match, Properties 1 and 2 are ensured by construction. For Property 3, we still need to verify the start date for \( P_0 \); this is done by the final check. The final check is shown in Fig. 6. In the case where \( P_0 \) has forbidden nodes (like in Fig. 4), the search algorithm retained potential start dates \( s \) even if \( C_{s+1} \) already matched \( P_0 \). We must now decide whether or not \( s \) is a real start date. Decision will be negative if \( C_{s+1} \) matches \( P_0 \) and does not contain any of the forbidden nodes. Back to the example in Fig. 4, the final check will accept \( t+1 \) as a start date for a match with \( n2:=2 \), but not for \( n2:=3 \) or \( n2:=4 \). It ensures that the temporal window of \( P_0 \) is indeed maximal.

```
Boolean FINALCHECK (PartialMatch pm)
If (pm.index[0]=0 || ForbiddenNodes(0)=\emptyset)
    return (true)
Else
    Let P' = VALUATEGRAPH (P0, pm.val)
    Let H= NEWMATCHGRAFP (P', Cpm.index[0], -1,
        ForbiddenNodes(0), pm.val)
    If (H is empty)
        Return (true)
    Else return (false)
Endif
End FINALCHECK
```

\[\text{Figure 6. Final check before outputting.}\]
Complexity problems only come from the repeated calls most quadratic in the number and size of the graphs.

Generally speaking, the cost of auxiliary treatments (pre-processing, final check) exhibits an increase that is at most quadratic in the number and size of the graphs.

C. Complexity of GraphSeq algorithm

Graph homomorphism building is an NP-complete problem. Let \( N_p \) be the number of nodes of a pattern graph, and \( N_c \) be the number of nodes of a concrete configuration graph. In the worst case, when every pattern node can be mapped to every concrete node, function MATCHGRAPHS( ) has an exponential complexity \( O(N_c^N) \). In the favorable case when every pattern node can be mapped to at most one concrete node, complexity is only quadratic in \( N_p \) and \( N_c \).

Let us now discuss the complexity added by the sequential reasoning. We are comparing a sequence of \( m \) patterns to a sequence of \( n \) concrete configurations.

Generally speaking, the cost of auxiliary treatments (pre-processing, final check) exhibits an increase that is at most quadratic in the number and size of the graphs. Complexity problems only come from the repeated calls to MATCHGRAPHS( ). For the sake of simplicity, let us abstract the cost of a call by \( H \) (as explained above, \( H \) is actually a function of the compared graphs, and may range from quadratic to exponential complexity class). Two cases can be distinguished for the sequential search: 1) there is no node creation in the sequence of patterns, and 2) there are node creations.

In the first case, for each candidate start date \( s \), the valuation of \( P_0 \) assigns a concrete id to every nodes. Every subsequent comparison of graphs yields at most one possibility for the mapping of nodes. The total cost is then dominated by the cost of comparing \( P_0 \) to each \( C_i \), yielding complexity \( O(nH) \).

In the second case, the impact of node creation is twofold. First, when \( P_{i+1} \) is processed, we must investigate a valuation for node ids that are new with respect to \( P_0, \ldots, P_i \).

This may involve several candidate mappings, defining different extensions of the current partial match. Second, given a valuation for new node ids, the algorithm must consider many candidate end dates if transition \( P_i \rightarrow P_{i+1} \) is triggered solely by the new nodes, yielding again different partial match extensions. Combining these effects, a pessimistic complexity estimate is \( O(n^2H^2) \).

// build all partial matches of depth 1
// with start date s,
// push each of them in L
Let H = MATCHGRAPHS (P0, C0)
While H is not empty
Extract h from H
Let P' = VALUEGRAPH (P0, h.val)
// Can s be a start date for P'?
Let start_OK = true
If (s > 0 && ForbiddenNodes (0) = ∅)
Let H' = MATCHGRAPHS (P', C0)
If (H' is not empty)
    // previous config did match
    // -- not a start date
    start_OK = false
Endif
Endif
// Now, check the end date
If (start_OK = true)
Let end = COMPUTEENDDATE (P', s, ∅, NULL)
If (StopBefore(0))
    // can stop at an intermediate date
    For [j=s; j=min(end,n-m); j++]
        let pm = CREATEPARTIALMATCH1 (h.val,s,j)
        push pm in L
    End for
Else If (end ≤ n-m)
    // end date is not too late
    let pm = CREATEPARTIALMATCH1 (h.val,s,end)
    push pm in L
Endif
End if
End While

(a) Core algorithm for building the matches

(b) Auxiliary functions
the worst case for $H$, we thus have an exponential increase in both the number and size of patterns. It is obvious that GraphSeq could not process large sequences of (large) graphs. However, we argue that it still is practically useful for its purpose, namely the analysis of test traces against graphical scenarios.

Graphical scenarios typically involve few entities, the interactions of which are the focus. For example, the scenarios we derived from the GMP case study (from the analysis of requirements, or from the analysis of failures observed during testing) never involved more than five symbolic nodes. This is comparable to what can be seen in other work, and anyway the graphical representation could not accommodate a very large number of lifelines. Similarly, it is expected that the number of successive patterns in scenarios remain tractable (e.g., typically two or three successive patterns in a GMP scenarios).

The size and number of concrete configurations is much less critical to the complexity of the analysis. Note that in-house testing rarely involves very large experimental settings. For example, in [18, 19, 20, 21, 22], the experimental configurations involve from 5 to 50 nodes. The duration of a test run range from 28s to 15 minutes. First experimentation with GraphSeq shows that such configurations are well in the reach of the tool.

V. VALIDATION AND EXPERIMENTS OF GRAPHSEQ

GraphSeq has been validated by using hundreds of randomly generated sequences of graphs. We also report on first experiments demonstrating the usage of GraphSeq to extract mobility patterns. They involve the analysis of test traces from the GMP case study, and of outputs from a mobility simulator.

A. Validation of GraphSeq

We performed a number of tests to validate GraphSeq. We first started with small examples that were manually produced (e.g., the example in Fig. 4 was included as one test case), but quickly came to the conclusion that we would need an automated solution for both the generation of graph sequences and the analysis of results. We developed a tool that produces random sequences of graphs $P_j$ and $C_j$ such that, by construction, $C_j$ contains at least one match. The GraphSeq results can then be automatically analyzed and a fail verdict is issued if the expected match is not found. Note that GraphSeq may find several matches, but the test verdict only concerns the one known to be there by construction.

The random generation can be parameterized, and we produced about 900 test cases exhibiting various characteristics:

- Number of patterns for each sequence from 1 to 5,
- Number of concrete configurations from 1 to 100,
- Number of nodes in patterns from 1 to 5,
- Number of nodes in concrete configurations from 1 to 25,
- For each individual $P_j$, duration of a matching from 1 to 20 steps in the concrete configurations.
- The transitions from $P_i$ to $P_{i+1}$ may involve a change in a node label, in an edge label, a node that appears, that disappears, or any combination (up to 5 changes).

The test tool proved very useful to debug GraphSeq, and to perform regression verification after changes in the C++ code.

B. GraphSeq applied to the GMP case study

The purpose of GraphSeq is to allow us to identify all occurrences of a scenario in a test trace. The graph matching part corresponds to the processing of the spatial view, to be supplemented by an analysis of the events occurring in the temporal window of the match, for the identified nodes. The principle of scenario extraction can be exemplified using the GMP case study mentioned in Section 2.

In this case study [9], our test experiments reported numerous violations of the protocol requirements defined in [8]. It was not possible to manually examine all of them, but a sample of 164 fail cases were selected and analyzed. The scenario shown in Fig. 1 is one of the outcomes from this analysis. Note that some observed failures could correspond to the same scenario occurring several times during the runs.

Manual analysis was tedious, and we now demonstrate how automated scenario extraction can be performed with the help of GraphSeq. In order to show an example of patterns with node creation, we investigate a variant of the previous scenario that should yield exactly the same failure. This variant differs in the spatial view (Fig. 8). Node n1 does not exist in the initial configuration. It then wakes up at a safe distance from n2, while n2 gets away from n4. After the configuration change event, communication events are like in Fig. 1b.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of nodes</td>
<td>Random in [6..15]</td>
</tr>
<tr>
<td>Start date of nodes</td>
<td>Random in [0..300] s</td>
</tr>
<tr>
<td>Death date of nodes</td>
<td>Random in [StartDate..300] s</td>
</tr>
<tr>
<td>Transmission range</td>
<td>300 m</td>
</tr>
<tr>
<td>Safe Distance</td>
<td>140 m</td>
</tr>
<tr>
<td>Maximal speed</td>
<td>10 m.s$^{-1}$</td>
</tr>
<tr>
<td>Mobility model</td>
<td>Random movement at maxi speed</td>
</tr>
<tr>
<td>Duration of a run</td>
<td>5 minutes</td>
</tr>
</tbody>
</table>

Figure 8. Spatial view of the scenario variant.
The experiments consist in testing the GMP implementation, recording execution traces, and searching for occurrences of the scenario variant. A total number of 100 test runs were generated with parameters in Table 1.

GraphSeq returns 75 matches of the spatial view in the execution traces. This allowed us to extract the subtraces of communication events occurring during the matches. Note that, at the date of the writing of this paper, the analysis of communication events has not been automated yet. We thus had to compare manually the events in the scenario and the subtraces. In 48 cases, the communication events did not match. The remaining 27 corresponded to occurrences of the scenario. The GMP exhibited the expected failure in each of these cases. This allowed us not only to confirm our diagnosis of the problem, but also to discriminate between fail cases that match or do not match the identified scenario.

### C. Connection to a mobility simulator

The GMP case study used a very simple management of node movement, with GPS stubs delivering random coordinates for each node. However, more realistic test platforms would include a mobility simulator.

Mobility simulators manage the relative positions of nodes according to some mobility models, and produce contextual data (e.g., location-based data) needed by applications. Examples of test platforms with a mobility simulator can be found in [23, 24]. The data produced at each simulation step may be used to identify the configurations of the tested system. To demonstrate the concept, we experimented with the connection of GraphSeq to a simulation tool.

We chose a mobility simulator developed at the University of Southern California, USA, as part of the IMPORTANT (Impact of Mobility Patterns On Routing proTocol in the mobile Ad hoc NeTworks) framework [25]. The tool is freely available on the web\(^1\). It offers a rich set of parameterized mobility models, including Reference Point Group, Freeway and Manhattan mobility models. The generated traces are compatible with the ns-2 network simulator.

The connection of GraphSeq to such a mobility simulator requires the development of an interfacing component (see Fig. 9) that abstracts the raw simulation data into a sequence of configuration graphs. In our experiments, the raw data are the position of nodes at each simulation step. Then, the abstraction consists in assigning edge labels according to the distance of nodes, like in the GMP case study.

We made trials with various mobility models, different parameterizations of the models and different patterns to be searched. We describe here an example of run using the freeway model.

This model emulates the motion behavior of vehicles in a freeway. It has the following characteristics:

- Each mobile node is restricted to its lane on the freeway.
- The velocity of a mobile node is temporally dependent on its previous velocity.
- If two mobile nodes on the same freeway lane are within a distance SD, the velocity of the following node cannot exceed the velocity of the preceding node.

Table 2 provides the parameters used for the example run. The map was built using predefined fragments of freeways and lanes, available in the simulation environment.

Our format translator extracts 345 different configuration graphs from the simulation trace. The concrete ids of mobile nodes are labeled “N0”, “N1”, “N2”,... The edges representing the spatial relations between two mobile nodes depend on their distance \(d\) and are defined as follows:

- \(0 < d < 140\text{m}: \text{edge} = 1\),
- \(140 \leq d < 300\text{m}: \text{edge} = 2\),
- \(d \geq 300\text{m}: \text{nodes are disconnected (300 m is thus the transmission range)}\).

The pattern graphs are as in Fig. 1a, with Safe = 1 and RangeNotSafe = 2.

GraphSeq searches for the sequence of patterns in the 345 concrete configurations, and returns 11 matches. Their manual extraction would have been unfeasible.

---

\(^1\) [http://nile.usc.edu/important/](http://nile.usc.edu/important/)
VI. CONCLUSION AND PERSPECTIVES

This paper has presented GraphSeq, a graph matching tool developed to process the spatial view of interaction scenarios in mobile settings. The spatial view explicitly represents the dynamically changing topology of system nodes, with mobility induced connections and disconnections, as well as dynamic creations and shutdowns of nodes.

GraphSeq takes as inputs two sequences of labeled graphs, intended to come respectively from a scenario and a trace, and generates the set of all matches. The corresponding algorithm uses existing facilities for graph homomorphism building, taken from another graph tool developed by colleagues at LAAS. Here, the novelty of our tool consists in reasoning on sequences of spatial configurations. Specifically, the accounting for scenarios where nodes dynamically appear and disappear proved a tricky issue. We thoroughly validated GraphSeq by challenging test cases that were manually designed, and then by hundreds of randomly generated graph sequences. We also started proof-of-concept experiments to demonstrate the usage of GraphSeq in connection with a mobility simulator, and in relation with a case study that is representative of mobile computing problems.

Our work will now tackle the integrated processing of the spatial and event views of scenarios. Joint work with the Technological University of Budapest has defined a UML profile for sequence diagrams in mobile settings, as well as a semantics for them [17]. This will allow us to complete the automated analysis of test cases by matching the communication events.

ACKNOWLEDGMENT

This work was partially supported by the ReSIST network of Excellence (IST 026764) and the HIDENETS project (IST 26979) funded by the European Union under the Information Society Sixth Framework Program. We wish to thank Khalil Drira, Karim Guennoun and Ismael Bouassida for having made their graph tool available to us, and for having spent time to respond to our questions.

REFERENCES

http://www.hidenets.aau.dk/Public+Deliverables