GraphSeq Revisited: More Efficient Search for Patterns in Mobility Traces

Pierre André[1,2], Nicolas Rivière[1,2], and Hélène Waeselynck[1,2]

[1] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France,
[2] Univ de Toulouse, UPS, LAAS, F-31400 Toulouse, France,
pierre.andre@laas.fr nicolas.riviere@laas.fr helene.waeselynck@laas.fr

Abstract. GraphSeq is a graph matching tool previously developed in the framework of a scenario-based test approach. It targets mobile computing systems, for which interaction scenarios must consider the evolution of the spatial configuration of nodes. GraphSeq allows the analysis of test traces to identify occurrences of the successive configurations of a scenario. This paper presents a recent improvement made to the tool, to allow for better performance in the average cases. It consists in rearranging the configuration patterns extracted from the scenario, so that the most discriminating nodes are matched first. The improvement is assessed using randomly generated graphs and a test trace from a case study in ad hoc networks.

Keywords: Graph matching, Performance, Testing, Mobile computing systems

1 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. Compared to "traditional" distributed systems, such systems execute in an extremely dynamic context. The movement of devices yields an unstable topology of connection. Links with other mobile devices or with infrastructure nodes may be established or destroyed depending on the location. Moreover, mobile nodes may dynamically appear and disappear as devices are switched on and off, run out of power or go to standby.

Our work addresses a passive testing approach for such systems. Passive testing (see e.g., [1]) is the process of detecting errors by passively observing the execution trace of a running system. In our case, the properties to be checked are specified using graphical interaction scenarios. Figure 1 gives a schematic view of the approach. The system under test (SUT) is run in a simulated environment, using a synthetic workload. The SUT may involve both fixed nodes and mobile devices. The movement of the latter ones is managed according to some mobility model, a context simulator being in charge of producing location-based data. Execution traces are collected, including both communication messages and location-based data. The traces are then automatically analysed with respect to predefined scenarios, representing test requirements or test purposes.

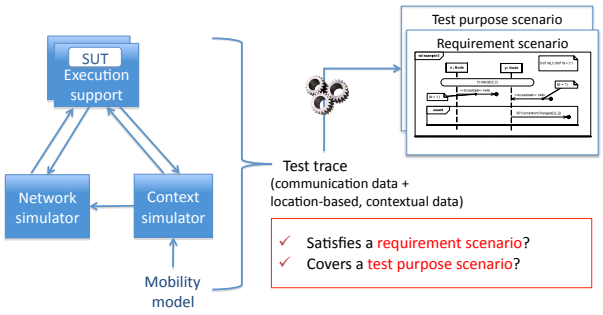


Fig. 1: Overview of the scenario-based approach

Test requirements specify mandatory (positive requirement) or forbidden (negative requirement) interactions. Any observed violation of a requirement must be reported. Test purposes specify interactions of interest, which we would like to observe at least once during testing. If the interaction appears in the trace, the test purpose is reported as covered.

Scenarios are described using a formal UML-based language called TERMOS (Test Requirement Language for Mobile Settings). TERMOS is a specialization of UML Sequence Diagrams [7]. Its genesis can be found in our earlier work [5,6,11]. We first noticed that the spatial configurations of nodes should be a first class concept [5]. As a result, a scenario should have both (i) a spatial view, depicting the dynamically changing topology of nodes as a sequence of graphs, and (ii) an event view representing the communications between nodes. To account for both views, the checking of test traces against scenarios should combine graph matching [6] and event order analysis [11]. In this paper, we focus on the graph matching part, which was implemented by a tool called GraphSeq.

The spatial configurations of a scenario provide a sequence of graphs (the patterns) and GraphSeq search for all occurrences of this sequence of patterns in mobility traces. The addressed graph matching problem is inherently costly, with a worst case complexity exponential in the size and number of the patterns. We present a functionality added to the original version of GraphSeq, in order to improve performance of the average cases. It consists in re-arranging the order of nodes in the patterns, so that GraphSeq tries to match the most discriminating nodes first. The improvement is measured using random graphs and a test trace from a case study in ad hoc networks.

Section II of this paper gives an overview of the TERMOS language. Section III describes the re-arrangement functionality we implemented. Section IV gives performance results. Section V discusses related work. Section VI concludes.

2 Overview of TERMOS

Figure 2 shows an exemplary TERMOS scenario, with its spatial and event views. Note that the shown syntax is not exactly the UML-based one presented in [11]. We adopt here a more compact representation that conveys the same

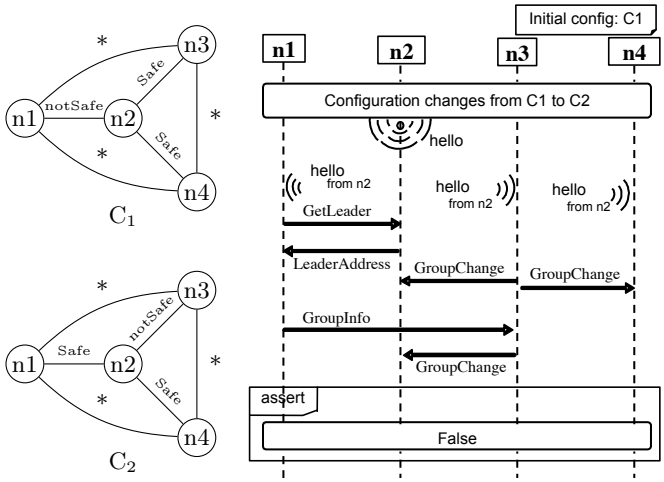
concepts. The scenario is extracted from a Group Membership Protocol (GMP) case study we performed [10]. In this GMP, groups split and merge according to the location of mobile nodes. The protocol uses the concept of safe distance to determine which nodes should form a group. Figure 2 presents a negative requirement (indicated by a false assertion). It describes a pathological interleaving of concurrent split and merge operations that should never occur.

The spatial view contains a set of spatial configurations for the nodes of the scenario. In [11], we depicted them using UML diagrams, but conceptually they consist of labelled graphs. The modeller chooses the labels that are meaningful for the target application. Edge labels characterize the connection of nodes, while node labels (not shown here) are used for contextual attributes of nodes. In Figure 2(a), nodes can have two types of connection, depending on their distance: Safe and NotSafe. Wildcards '*' indicate don't care connection types.

The event view shows the interaction of nodes. Lifelines are drawn for the nodes and the partial orders of their communications are shown. The successive spatial configurations underlying the communications are made explicit: the scenario has an initial configuration and configuration change events are represented (e.g., a change from C1 to C2 in Figure 2(b)).

We interpret TERMOS scenarios as generic patterns, instances of which are searched for in the execution trace. In Figure 2, the nodes ni are symbolic. Any subset of four SUT nodes can match them during execution, by exhibiting the proper spatial configurations and communication events. The search for scenario instances involves two steps:

1. Determine which physical nodes of the trace exhibit the sequence of configurations of the scenario, and when they do so.
2. Analyze the order of events in the identified SUT configurations.



(a) Spatial view (b) Event view

Fig. 2: a TERMOS scenario for groups of mobile nodes

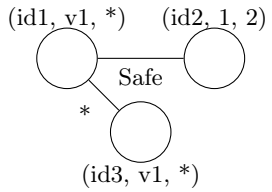


Fig. 3: A pattern graph with various forms of label

Step 1 amounts to a graph matching problem: we search for a sequence of graph patterns (coming from the scenario) to appear in a sequence of SUT configurations (retrieved from the location-based data in the trace). Step 2 requires an interpretation of the event view in terms of partial orders of events. The TERMOS semantics encodes the partial orders in a symbolic automaton to categorize trace fragments as valid, invalid or inconclusive.

In the processing of a scenario, the costliest part is Step 1. For example, we had a GMP execution with 15 nodes moving according to the Reference Point Group Mobility model, during 15 minutes. We checked the logged trace against the scenario of Figure 2. It took about one hour and fifty minutes for the graph matching, while less than five seconds for the event checking in all found configurations. Clearly, an improvement of the graph matching performance is the key for better efficiency.

3 Improvement: Re-arranging Patterns

GraphSeq takes as input two sequences of graphs: (1) a sequence of pattern graphs coming from a scenario description, (2) a sequence of concrete configuration graphs extracted from an execution trace. It compares the two sequences of graphs and returns all matches for the pattern sequence. A match identifies a subset of concrete nodes that exhibit the searched sequence of patterns.

The patterns may involve label variables and wildcards, as illustrated by Figure 3. Nodes have at most three labels. The first one is mandatory and has the form of a variable; 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 contextual attributes. They may have the form of constant values, variables or wildcards. In Figure 3, the pattern indicates that the concrete nodes playing the role of $id1$ and $id3$ have their first optional attribute at unknown, but identical, values. Each instance of the pattern determines a valuation for the variables. GraphSeq will explore all possibilities, with a sequential reasoning to account for successive patterns. The worst cases are exponential in the size and number of patterns. They occur when every pattern node can be mapped to every concrete node, and the choices made at some point of the sequence of graphs does not restrict the choices for the rest of the sequence.

Fortunately, such worst cases are unlikely to correspond to meaningful scenarios. Rather, the specified patterns should possess some specificities that make them of interest for the application. An idea is then to re-arrange the patterns so that the most discriminating nodes are matched first. To introduce the idea,

let us take the example of the C1 pattern in Figure 2(a). In the encoding supplied to GraphSeq, the first node of the pattern is n1. This node has don't care values for its optional labels (like have other nodes in this example) and also has don't care values for most of its connections to other nodes of the pattern. It is thus not discriminating: many concrete nodes are likely to match n1, and GraphSeq will explore all possibilities. If the pattern is re-arranged so that n2 appears first, GraphSeq will have fewer possibilities to explore, because n2 is more discriminating as regards its connections to other nodes.

We thus introduce a pre-processing step in GraphSeq. Given a pattern, a fitness score rewards the discriminative power of each node appearing in it. Then, the pattern is re-arranged so that the nodes are sorted in descending order of fitness, making the graph matching algorithm consider the fittest nodes first. Algorithm 1.1 shows the computation of the fitness score of a node. It rewards pre-determined attribute values and a high number of pre-determined connection types with other nodes.

```
Fitness = 0;

// Reward the node if optional labels are discriminating
for each optional label li
    if li is a constant value or a variable that appeared in a previous pattern then
        Fitness += 2;
    endif
endfor

// Reward the node if its connection types with other nodes are discriminating
for each other node nk of the pattern
    if connection to nk is not a don't care then
        Fitness += 1;
    endif
endfor
```

Algorithm 1.1: Fitness score for a node appearing in pattern P_i

4 Experimental results

The functionality optimizing the order of nodes in patterns was integrated into GraphSeq, in such a way that it can be activated/deactivated by the user. This allows us to assess its effect on the efficiency of the search for matches. Given two sequences of graphs to be compared, we successively run the tool with and without activation of the functionality to compare the obtained durations.

All experiments were performed on the same platform, a computer with two 2.26GHz quad core and 48GB of ram. The current implementation of GraphSeq is not multi-threaded and uses only one core. Some runs required an amount of memory greater than the available RAM. In such cases, we decided to forbid the use of virtual memory, which would anyway considerably decrease performance. A run is stopped whenever it consumes more than 90% of memory or its duration exceeds three hours.

We first considered the GMP scenario of Figure 2. We ran again the analysis of the GMP mobility trace, using the new version of GraphSeq with the optimization deactivated. The trace involves 15 concrete nodes exhibiting a sequence of 850 concrete configurations. It took GraphSeq 6600.78 seconds to analyze them and find 59 matches for the scenario. With the optimization activated, it took

GraphSeq only 40.63 seconds to find the same matches, more than 160 times faster than previously.

To further assess the improvement, we used randomly generated sequences of graphs. A generation function was available from previous work: when we developed the GraphSeq algorithms, we also developed a test tool to verify the correctness of the matching. The tool generates pairs of pattern and concrete configuration sequences, and by construction there is at least one match for each pair. It can then be verified whether this match is correctly found by GraphSeq. The size and number of graphs can be tuned. The test tool proved very useful to debug GraphSeq and perform regression testing of its successive versions. Here, we use it for evaluation purposes.

Table 1 shows the results of running GraphSeq on randomly generated sequences of graphs. The first column indicates some generation parameters. For example, quadruplet (5, 5, 5..35, 700..2100) means that:

- the generated pattern graphs have 5 nodes;
- the lenght of the pattern sequences is of 5 successive graphs;
- the concrete configuration graphs have a number of nodes in the range of 5..35;
- the lenght of the concrete configuration sequence is in the range of 700...2100 graphs.

We generated 20 pairs of sequences for each experimental quadruplet, yielding 20 runs of GraphSeq without and with optimization. The table gives the number of aborted runs in each case, due to either excessive memory consumption or excessive time duration. It was never the case that a run successfully completes with the optimization deactivated while being aborted with optimization. For the longest pattern sequences (second and fourth row of the table), the optimization proved very effective to allow completion of runs that had to be stopped in the original version of GraphSeq.

The duration values of completed runs could be compared. The tables give the mean, median and standard deviation we observed. The high value of σ

	♯ Aborted runs		Duration of completed runs in seconds: $\mu(\sigma)[median]$		p-Value
	w/o opt	opt	w/o opt	opt	
(5, 5, 5..35, 700..2100)	Mem: 2 Time: 0	Mem: 0 Time: 0	1110.29 (2335.47) [140.14]	382.55 (1369.86) [18.72]	$< 10^{-5}$
(5, 10, 10..40, 700..2100)	Mem: 7 Time: 3	Mem: 0 Time: 0	511.82 (635,93) [226.50]	213.18 (31.96) [207,76]	0.037
(10, 5, 5..35, 1200..3600)	Mem: 0 Time: 2	Mem: 0 Time: 1	909.22 (1786.19) [43.38]	259.93 (799.54) [39.68]	0.001
(10, 10, 10..40, 1200..3600)	Mem: 6 Time: 8	Mem: 0 Time: 0	281.92(396.61) [95.67]	47.07(6.16) [47.33]	0.031

Table 1: Runs with random sequences of graphs. Graph generation is characterized by a quadruplet (number of nodes per pattern, number of patterns, range for the number of nodes per concrete configurations, range for the number of concrete configurations). There are 20 runs in each experimental setting.

indicates that, for a given setting of the graph generation, the difficulty of the generated matching problems still largely varies. Moreover, the mean and median could be quite different indicating the values are not normally distributed. We observed lower mean and median when GraphSeq had optimization activated. To determine whether the improvement is statistically significant, we performed a paired difference test. We used the Wilcoxon T test since a normal distribution cannot be assumed. The p-values are reported in the tables.The null hypothesis can be rejected with a 95% confidence level for all experiments.

We conclude that the proposed pattern re-arrangement facility yields a significant improvement of GraphSeq.

5 Related work

Subgraph isomorphism detection is a problem well studied in the literature [9,4]. In GraphSeq, the core functionality to check whether a pattern appears as a subgraph of a concrete configuration is reused from an existing graph tool developed by colleagues at LAAS [3].

The salient feature of GraphSeq is its algorithm to match *sequences* of graphs: the sequence of symbolic configurations of the scenario, and the sequence of concrete configurations traversed during SUT execution. 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 [2] for a survey on graph matching). The closest work we found is for the analysis of video images. In [8], 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 a pattern node corresponds to at most one object in an image. In our case, several instances of a pattern may be found in a concrete configuration, with different possible valuations for the variables (including node ids). Hence, to the best of our knowledge, the sequential reasoning implemented by GraphSeq is original.

6 Conclusion

In this paper, we have presented an improvement made to our graph matching tool GraphSeq. Its principle is simple: reward pattern nodes according to their discriminating power, and re-arrange the encoding of the patterns so that the fittest nodes are matched first. While simple, the proposed optimization proved very effective to improve performance. A mobility trace issued from a case study, a group membership protocol in ad hoc networks, could be processed 160 times faster than previously. These promising results were confirmed by experiments on a sample of randomly generated sequences of graphs. The duration values were found significantly higher with than without optimization. Moreover, for the largest configurations we generated, the optimization made it possible to complete a significantly higher number of runs than the original version of GraphSeq. This of course does not change the theoretical limitation of the tool due to the

exponential complexity of the matching problem. But it improves our ability to handle the cases we are targetting in practice, that is, scenarios with small-size patterns that are unlikely to correspond to the worst cases (in the worst cases, no node would be discriminating, hence jeopardizing the node re-arrangement functionality).

GraphSeq is an important component of our scenario-based test platform for mobile computing systems. It addresses the processing of the spatial view of TERMOS scenarios, where the movement of nodes is abstracted by a sequence of labelled graphs. Another tool, integrated into the Papyrus UML environment, uses the outputs of GraphSeq to process the event view showing inter-node communication. The complete processing of a TERMOS scenario is dominated by the duration of the graph matching, which is the costliest part of test trace analysis. By significantly improving the performance of GraphSeq, we thus also significantly improve the overall approach.

References

1. Ana Cavalli, Stephane Maag, and Edgardo Montes de Oca. A passive conformance testing approach for a MANET routing protocol. In *Proceedings of the 2009 ACM symposium on Applied Computing*, SAC '09, pages 207–211, New York, NY, USA, 2009. ACM.
2. Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Thirty years of graph matching in pattern recognition. *IJPRAI*, 18(3):265–298, 2004.
3. Karim Guennoun and Khalil Drira. Using graph grammars for interaction style description: applications for service-oriented architectures. *Comput. Syst. Sci. Eng.*, 21(4), 2006.
4. Bruno T. Messmer and Horst Bunke. Efficient subgraph isomorphism detection: A decomposition approach. *IEEE Trans. Knowl. Data Eng.*, 12(2):307–323, 2000.
5. M.D. Nguyen, H. Waeselynck, and N. Rivière. Testing mobile computing applications: toward a scenario language and tools. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 29–35. ACM, 2008.
6. Minh Duc Nguyen, H. Waeselynck, and N. Rivière. Graphseq: A graph matching tool for the extraction of mobility patterns. In *Proc. Third Int Software Testing, Verification and Validation (ICST) Conf*, pages 195–204, 2010.
7. Omg. OMG Unified Modeling Language (OMG UML), Superstructure Specification (Version 2.4.1). Technical Report OMG Document Number: formal/2011-08-06, Object Management Group, August 2011.
8. Kim Shearer, Svetha Venkatesh, and Horst Bunke. Video sequence matching via decision tree path following. *Pattern Recognition Letters*, 22(5):479–492, 2001.
9. Julian R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1):31–42, 1976.
10. H. Waeselynck, Z. Micskei, Minh Duc Nguyen, and N. Riviere. Mobile systems from a validation perspective: a case study. In *Proc. Sixth Int. Symp. Parallel and Distributed Computing ISPDC '07*, 2007.
11. Helene Waeselynck, Zoltan Micskei, Nicolas Riviere, Aron Hamvas, and Irina Nitu. TERMOS: a formal language for scenarios in mobile computing systems. In *7th International ICST Conference on Mobile and Ubiquitous Systems*, Sydney, Australia, 2010.