Quantifying the Similarity of Non-bisimilar Labelled Transition Systems

Gwen Salaün

Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, F-38000 Grenoble France

Abstract. Equivalence checking is an established technique for automatically verifying that two behavioural models (Labelled Transition Systems, LTSs) are equivalent from the point of view of an external observer. When these models are not equivalent, the checker returns a Boolean result with a counterexample, which is a sequence of actions leading to a state where the equivalence relation is not satisfied. However, this counterexample does not give any indication of how far the two LTSs are one from another. One can wonder whether they are almost identical or totally different, which is quite different from a design or debugging point of view. In this paper, we present an approach for measuring the similarity between two LTS models. The set of metrics is computed automatically using a tool we implemented. Beyond presenting the foundations of the proposed solution, we will show how it can be applied to a concrete application domain for supporting the construction of IoT applications by composition of existing devices.

1 Introduction

Designing and developing distributed software has always been a tedious and error-prone task, and the ever increasing software complexity is making matters even worse. Although we are still far from proposing techniques and tools avoiding the existence of bugs in a software under development, we know how to automatically chase and find bugs that would be very difficult, if not impossible, to detect manually.

Model checking [1] is an established technique for automatically verifying that a model (Labelled Transition System, LTS), obtained from higher-level specification languages such as process algebra, satisfies a given temporal property. Equivalence checking [12] is an alternative solution to model checking and is very helpful to check that two models (requirements and implementation for instance) are equivalent from the point of view of an external observer. When these models are not equivalent, the checker returns a Boolean result with a counterexample, which is a sequence of actions leading to a state where the equivalence relation is not satisfied. However, this counterexample does not give any indication of how far the two LTSs are one from another. One can wonder whether they are almost identical or totally different, which is quite different from a design or debugging point of view.

In this paper, we propose a set of metrics for quantifying the similarity of two behavioural models described using LTS. More precisely, our solution takes as input two LTS models and applies first the partition refinement algorithm [17, 7] to identify bisimilar and non-bisimilar states between the two LTSs. Then, we focus on non-bisimilar states and compute a set of local and global metrics for each couple of non-bisimilar states. This allows us to build a matrix with a measure between 0 (totally different states) and 1 (bisimilar states) for each couple. To do so, we rely on several criteria such as the matching of incoming/outgoing transitions, the similarity of neighbour states, the shortest distance from the initial state, and the distance to the closest bisimilar state. Once this matrix is computed, we use it to finally obtain a global measure of similarity of both LTSs. All these measures are computed automatically using a tool we implemented in Python and applied on a large set of examples.

Better understanding and measuring the difference between two behavioural models can be of interest in many different contexts and application areas. It can be used for debugging purposes when the counterexample is not sufficient for detecting the source of the bug, for measuring the distance between two versions of a software, for process model matching in the context of business process and management, etc. We will show in this paper how it can be helpful in the Internet of Things (IoT). One of the main challenges in this area is to build a new application by composing existing objects or devices. This application or composition is satisfactory if it conforms to what the user expects from it. These requirements are formalised using an abstract goal in this work. We will show how we use the proposed measures to compare the candidate composition and the goal. The quantitative results help in understanding what parts of the composition are correct or not *wrt.* this goal, and in guiding the user to finally end up with a satisfactory composition.

The rest of this paper is organized as follows. Section 2 defines LTSs and the notion of strong bisimulation used in equivalence checking. Section 3 presents the details of our approach to compute both the similarity matrix and the global measure of similarity. Section 4 illustrates the proposed solution on a case study in the IoT application domain. Section 5 reviews related work and Section 6 concludes the paper.

2 Labelled Transition Systems

In this work, we rely on *Labelled Transition System (LTS)* as low-level behavioural model of concurrent programs. An LTS consists of states and labelled transitions connecting these states.

Definition 1. *(LTS) An LTS is a tuple* $M = (S, s^0, \Sigma, T)$ *where* S *is a finite set of state identifiers;* $s^0 \in S$ *is the initial state identifier;* Σ *is a finite set of labels;* $T \subseteq S \times \Sigma \times S$ *is a finite set of transitions.*

A transition is represented as $s \xrightarrow{l} s' \in T$, where $l \in \Sigma$. An LTS can be produced from a higher-level specification of the system described using process

algebra for instance. Process algebraic specifications can then be compiled into an LTS using specific compilers. We support nondeterministic LTSs in this work, that is, there may be several transitions outgoing from a specific state labelled with the same action.

When comparing two LTSs, we can use different notions of equivalence, from weak ones such as trace or observational equivalence to stronger ones such as strong bisimulation. In this work, we chose to use strong bisimulation as originally defined in [12]. Supporting weaker notions of bisimulations where silent actions are handled separately is part of future work. It is worth noting that both LTSs are reduced using standard minimization techniques in this work before comparing them.

Definition 2. *(Strong Bisimulation) A relation R is a strong bisimulation between states in S iff for all $s_1, s_2 \in S$ such that $R(s_1, s_2)$, both conditions hold:*

- $(\forall b \in A, t_1 \in S)\ (s_1, b, t_1) \in T \implies (\exists t_2 \in S)\ (s_2, b, t_2) \in T \wedge R(t_1, t_2)$
- $(\forall b \in A, t_2 \in S)\ (s_2, b, t_2) \in T \implies (\exists t_1 \in S)\ (s_1, b, t_1) \in T \wedge R(t_1, t_2)$

Two states s_1 and s_2 are strongly bisimilar (written $s_1 \approx_s s_2$) iff there exists a strong bisimulation R such that $R(s_1, s_2)$. Two LTS $M_1 = (S_1, s_1^0, \Sigma_1, T_1)$ and $M_2 = (S_2, s_2^0, \Sigma_2, T_2)$ are strongly bisimilar (written $M_1 \approx_s M_2$) iff $s_1^0 \approx_s s_2^0$.

Equivalence checking is usually checked using partition refinement algorithms [17, 7]. These algorithms aim at building the minimal number of blocks, where a block is a set of (strongly) bisimilar states. One block is called an equivalence class. In order to check whether two LTSs are equivalent, the partition refinement algorithm is called with the union of both LTSs as input. At the end of this computation, if both initial states are in the same block, the LTSs are equivalent.

3 Comparing Non-bisimilar States

In this section, we present the measure of similarity between two LTSs. This measure applies when the equivalence checker indicates that both LTSs are not strongly bisimilar. In that case, we apply our approach to quantify the difference between the two subparts of both LTSs that are not equivalent. The measure relies on two kinds of criteria, namely local and global criteria, which focus on two non-bisimilar states (one in each LTS). We also present a global measure that gives a single measure of how far both LTSs are. Finally, we introduce a tool that allows us to automatically compute all these results.

3.1 Overall Approach

Given two LTSs, we first use the partition refinement algorithm mentioned in Section 2 to compute bisimilar and non-bisimilar states. Then, we focus on non-bisimilar states and propose a measure comparing all non-bisimilar states according to several local and global criteria. For each couple of non-bisimular

states (one non-bisimilar state from each LTS), we compute a degree of similarity which belongs to [0..1]. All these results are stored in a matrix where non-bisimilar states of one LTS appear in row and non-bisimilar states of the other LTS appear in column.

Given two non-bisimilar states s_1, s_2 where $s_1 \in LTS_1$ and $s_2 \in LTS_2$, we compute the similarity of those states using global and local criteria. Global criteria aim at considering the structure of both LTSs and looking at the respective positions of both states in their LTSs. More precisely, there are two global criteria. The first one computes the distance from the initial state to the given state in both LTSs and compares those distances. The second one computes the distance from a given state to the closest bisimilar state and compares those distances. In both cases, we compute the shortest distance.

There are four local criteria. The first one compares outgoing transitions to see the number of matching labels. The second does the same with incoming transitions. The third one checks whether the nature of states differ (initial or not). The last one compares the similarity of neighbour states.

Given all these values for a couple of states, we can then compute its value in the matrix ($matrix[s_1, s_2]$). This is obtained by using the weighted average of these values (*e.g.*, 1/6 or arbitrary weights). Since the similarity of neighbour states uses the matrix itself, we use an iterative algorithm that stops when the matrix stabilizes. Once the matrix is computed, we can compute a global measure of similarity, which gives a degree of similarity of both LTSs.

In the rest of this section, we explain in more details the metrics used in this work for computing the similarity measure.

3.2 Global Criteria

These criteria aim at comparing two states $s_1 \in LTS_1$ and $s_2 \in LTS_2$ by looking at their positions in their respective LTSs. We rely on two measures: (i) comparison of distance from initial states to states s_1 and s_2 (d_{init}), (ii) comparison of distance between s_1 and s_2 to their closest bisimilar states (d_{bis}). In both cases, we search for the shortest path. Both measures are then computed in the same way as follows: $1-(abs(d1-d2)/max(d1,d2))$, where d_1, d_2 is the distance from s_1, s_2 to the closest bisimilar state or from initial states to s_1, s_2, abs is the absolute value function, and max returns the longest distance.

Example. We illustrate with a simple example where we take the shortest distance from the initial states to two states $s_1 \in LTS_1$ and $s_2 \in LTS_2$ (Fig. 1). Assume first that $d1 = 1$ and $d2 = 8$. In that case $d_{init} = 1-((8-1)/8) = 0.125$ corresponding to a quite low value for this distance criterion. Consider now that $d1 = 3$ and $d2 = 5$. This results in a highest value $d_{init} = 1-((5-3)/5) = 0.6$. If we take equal values such as $d1 = 4$ and $d2 = 4$, we obtain the highest value $d_{init} = 1-((4-4)/4) = 1$, which means that these two states are not distinguishable with respect to this metric.

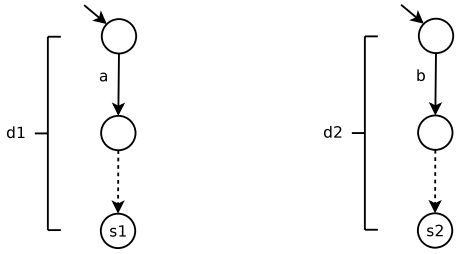

Fig. 1. Example for the Distance Comparison Criterion.

3.3 Local Criteria

These criteria aim at comparing two states $s_1 \in LTS_1$ and $s_2 \in LTS_2$ by looking at their transitions and states (nature and neighbours). We consider four local criteria:

- counting the number of matching outgoing transitions
- counting the number of matching incoming transitions
- comparison of nature of states (initial or not)
- comparison of neighbour states

Given two sets of transitions T_1 and T_2 outgoing from states s_1 and s_2, resp., we compute the similarity of those transitions (m_{out}) as follows: ((number of matching transitions in T_1) / $|T_1|$ + (number of matching transitions in T_2) / $|T_2|$) / 2. This measure is undefined if there is no outgoing transitions. The same measure is computed for incoming transitions.

Example. We illustrate with two simple examples where we compare the transitions outgoing from two states $s_1 \in LTS_1$ and $s_2 \in LTS_2$ (Fig. 2). Consider first the two states on the left hand side of Fig. 2. We obtain $m_{out} = ((1/1) + (1/2))/2 = 0.75$ because the transition outgoing from s_1 has a counterpart whereas only one of the two transitions outgoing from s_2 has a matching transition. If we now look at the second example on the right hand side of Fig. 2, we have $m_{out} = ((1/2) + (1/2))/2 = 0.5$ because from s_1 (s_2, resp.), only half of the transitions have a match.

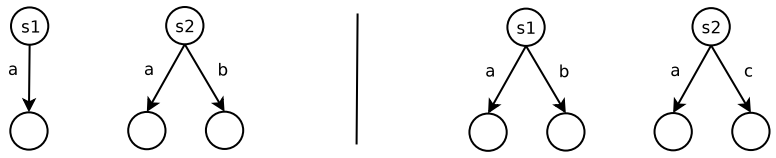

Fig. 2. Example for the Transition Matching Criterion.

The nature of two states is simple. If both states are initial or not, we return 1. Otherwise, they have a different nature (one is initial, the other is not), and in that case, we return 0.

The fourth metric takes into account the similarity of neighbour states. More precisely, given two states s_1 and s_2, this similarity measure (m_{neig}) is obtained by computing the average of the similarity of all its neighbours.

Example. Suppose two states $s_1 \in LTS_1$ and $s_2 \in LTS_2$ as depicted in Fig. 3. The similarity of neighbour states is computed as follows: $m_{neig}[s1, s2] = (m[s1', s2'] + m[s1'', s2''] + m[s1'', s2'''])/3$.

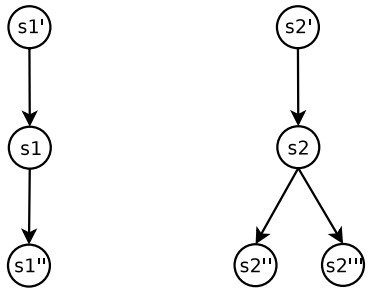

Fig. 3. Example for the Neighbour Similarity Criterion.

Since the computation of state similarity uses the matrix itself (for neighbour states), we use an iterative algorithm that stops when the matrix stabilizes. In practice, the iterative process terminates when the distance δ between two versions of the matrix goes below a fixed threshold. The distance between two matrices is obtained by computing the arithmetic mean of the difference of the two same states in each matrix.

Note that the computation of the matrix always converges to a unique similarity matrix. This convergence can be proven as achieved in [14] by using Banach's fixed point theorem.

3.4 Global Measure

Once the similarity matrix is computed, there are several options for computing a global score out of the matrix. We adopted here an optimistic point of view by computing the average of the best score for each row and for each column. Other global measures are possible, *e.g.*, by computing the average of all values greater than a given threshold.

Example. Tab. 1 gives an example of matrix obtained applying the aforementioned computations. We keep the best scores for each row and for each column. The best score for each row coincides with the best score for each column on this example, but this is not always the case. We finally compute the average of all these best scores, which results in a global similarity value of 0.9 in this case.

	s0	s1	s2
s0	0.9	0.25	0.42
s1	0.25	0.98	0.39
s2	0.42	0.39	0.83

Table 1. Example for the Global Similarity Measure.

3.5 Tool Support

The partition refinement algorithm and the similarity measures (matrix and global measure) are computed via a tool (DLTS) we implemented in Python, see Fig. 4 for an overview. It takes as input two LTSs specified in the textual 'aut' format. In practice, we use the LNT process algebra [4] for specifying high-level concurrent systems and compile these specifications to LTSs in 'aut' format by using CADP compilers [6].

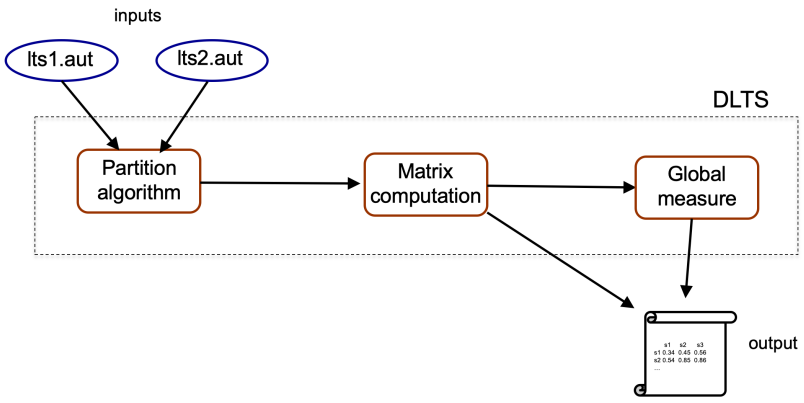

Fig. 4. Tool Support.

We applied our tool to many examples (we have about 100 aut files in our repository of examples). Experiments were carried out to evaluate the quality of the results using the well-known precision and recall measures. In this specific context, we consider the matching of states as basis for these measures, and we assume two states to be a correct match if each state cannot obtain a higher result in the similarity matrix with any other state. Precision computes the number of correct matched states out of all matched states detected by our approach. This allows us to verify that we do not have too many false positives (irrelevant matched states). Recall corresponds to the number of correct matched states detected by our approach out of all expected matched states. This allows us to measure the number of matched states our approach fails to identify. We have computed the precision and recall measures for several examples taken from our

repository. These measures show the good quality of our approach with high values for precision and recall.

Performance and scability are however an issue. It takes several minutes to compute the matrix for LTSs involving thousands of states/transitions. This comes from both the computation of the partition refinement algorithm and from the matrix computation. The good point is that our approach does not target large LTSs but rather small ones as we will show in the next section with a real-world case study taken from the IoT application domain.

4 Application to Composition of IoT Objects

In this section, we illustrate with one possible application of the measure of similarity between LTSs, namely the design of IoT applications by composition of devices and software (object for short in the rest of this section). Each object must exhibit the actions it can execute as well as the order in which these actions must be triggered. Such a public interface can be described using an LTS, as proposed in [9, 8], where labels on transitions correspond to these actions. Two objects interact one with another by synchronizing on same action name (synchronous binary communication model).

Given such a behavioural model for objects, the overall objective is to build a satisfactory composition of objects that satisfies a given goal. The goal is an abstract specification of what the user expects from the resulting composition. It can be modelled using an LTS too using interactions as labels (synchronization of actions) as suggested in [5]. A composition is satisfactory if it satisfies the goal. This can be verified using first the synchronous product to build a unique LTS out of a set of object LTSs, and then comparing the resulting LTS with the goal LTS using equivalence techniques (strong bisimulation here).

This case study aims at building a new IoT application for home security and more precisely for home intrusion detection. The goal of this application is given in Fig. 5 using an LTS, which indicates that when a move is detected, the camera is turned on, an alert message is sent to a mobile phone, and the light in the house is switched on.

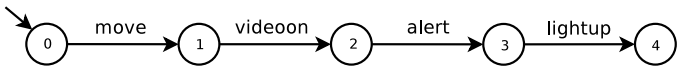

Fig. 5. Case Study: Goal.

Usually, in order to build an application satisfying these requirements, the end-user needs a recommender system listing all objets available nearby with their interface. We assume here that after this task the four objects given in Fig. 6 are selected as possible candidates. There is first a security sensor that detects movement in the house. When a movement is detected it turns the video on and sends an alert message to a mobile phone. Once the alert is over, the

sensor reinitializes and turns the video off. The second object is a connected light that can be repeatedly switched on and off. The third object is a security camera whose video can be activated or not. When activated, watching the video is possible. Finally, the final object is a home security app that can be installed on a smartphone. This app is triggered when receiving an alert. Then, there are several functionalities available for the user such as watching the video or switching the light on/off. Once the alert is over, the app allows the user to initialize it again.

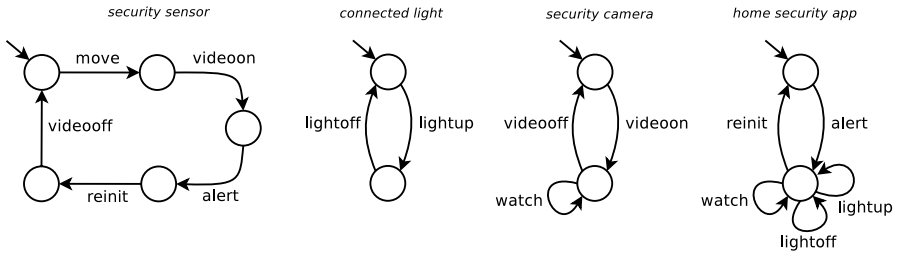

Fig. 6. Case Study: Objects.

The next question is the following: are we sure that this selection of objects does satisfy the given abstract goal? This is when equivalence checking comes into play. From the objects given in Fig. 6, we can build the resulting LTS using the classic parallel composition operator available in process algebra or the synchronous product of communicating automata. Here, we synchronize two objects on same actions. If an action in one object does not have any counterpart in another object, this is an independent evolution. As a result we obtain the LTS (generated with CADP) depicted in Fig. 7.

We now compare both LTSs (goal, Fig. 5, and composition LTS, Fig. 7) using the DLTS tool. Tab. 2 shows the resulting matrix obtained after four iterations. The comparison method also indicates that all states are non-bisimilar. The global similarity measure returns a value of 76%, indicating that both LTSs are not totally different and exhibit portions of their behaviours that are very similar. By looking more carefully at the matrix given in Tab. 2, we can see that the first three states are very similar with values higher than 80%. But then, most values are very low. If we look at the states in the composition LTS, we understand that this is due to actions present in the objects and their composition that are not taken into account by the abstract goal (e.g., lightoff, videooff).

There are now several options. One can try another selection and combination of objects. Another option is to go further in the analysis and comprehension of the current solution. As far as the latter is concerned, we can decide to refine the goal by integrating the missing actions, or we can keep the abstract goal as is and hide in the composition LTS these irrelevant actions. We decide to go for this final option by hidding the actions where light and camera are switched off

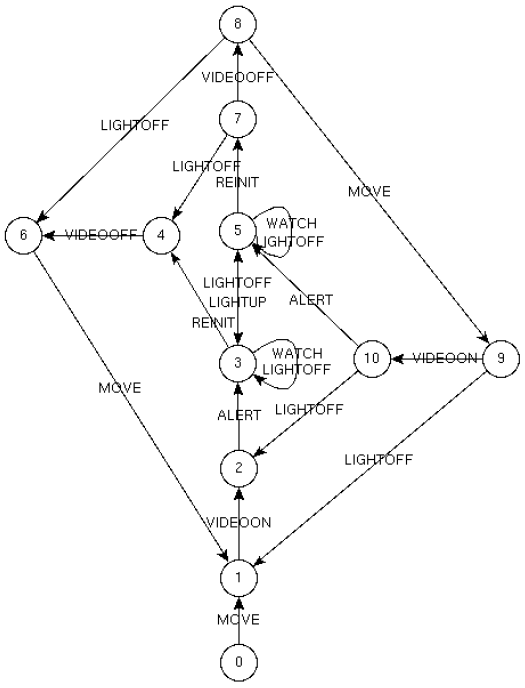

Fig. 7. Case Study: Composition LTS.

(lightoff and videooff). Fig. 8 shows the resulting LTS (generated with CADP) where hidden actions have been removed for the sake of readability.

Tab. 3 gives the similarity matrix computed by comparing the goal (Fig. 5) with this second composition LTS (Fig. 8). All states are non-bisimilar but the global measure increases to 89%. The matrix shows very similar states for the three first states (s0, s1, s2) and lower values for the remaining states (s3, s4). This is quite normal because the two LTSs exhibit several differences in states s3 and s4: (i) the goal is non-looping whereas all objects can loop forever and so the composition LTS, (ii) the reinit and watch actions were not made explicit in the goal but they totally make sense, and (iii) the lightup action is in sequence

	s0	s1	s2	s3	s4	s5	s6	s7	s8	s9	s10
s0	0.97	0.18	0.17	0.17	0.16	0.17	0.47	0.16	0.35	0.22	0.17
s1	0.18	0.86	0.36	0.33	0.3	0.31	0.3	0.3	0.29	0.68	0.33
s2	0.17	0.38	0.88	0.43	0.37	0.39	0.35	0.36	0.34	0.33	0.72
s3	0.19	0.34	0.45	0.77	0.44	0.59	0.4	0.43	0.4	0.37	0.38
s4	0.4	0.42	0.48	0.58	0.62	0.74	0.55	0.58	0.52	0.5	0.48

Table 2. Case Study: First Similarity Matrix.

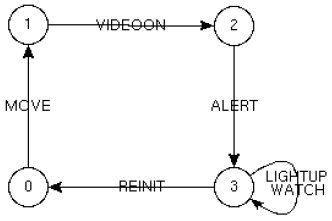

Fig. 8. Case Study: Composition LTS (V2).

in the goal whereas it can be repeated in the composition. It is worth noting that these differences make the equivalence not satisfied, but once better understood using our measures, there are not real problems from a functional perspective. Therefore, although the two LTSs are not bisimilar, the end-user could be satisfied by the proposed composition of objects and accept it as a correct solution for his/her application.

	s0	s1	s2	s3
s0	0.9	0.18	0.17	0.15
s1	0.05	0.99	0.36	0.32
s2	0.07	0.35	0.98	0.43
s3	0.08	0.33	0.44	0.79
s4	0.26	0.37	0.47	0.72

Table 3. Case Study: Second Similarity Matrix.

Last but not least, the user can take advantage of the first LTS generated for the initial composition (Fig. 7) where all possible executions are enumerated to verify additional properties (absence of deadlocks, a certain action is always reachable, some action occurs after another one, etc.). This can be achieved using temporal logic and model checking (we use MCL [10] and the CADP model checker [6], resp., in this work).

5 Related Work

Comparing automata-based models using equivalence techniques is not a new problem. It was studied for instance in the context of the composition of web services, see, e.g. [21, 2]. [21] relied on observational equivalence for checking that two versions of a service composition were the same. [2] proposed one compatibility definition based on bisimulation techniques for checking whether two web services can interact properly. In this work, our focus is on quantitative aspects of non-equivalent behavioural models.

[22] measures the similarity of Labelled Transition Systems (LTSs) w.r.t. a simulation and a bisimulation notion inspired from the equivalence relations. The measuring techniques use weighted quantitative functions which consist in a simple (not iterative), forward, and parallel traversal of two LTSs. This work does not return a global similarity measure and the differences which distinguish one entity from another.

In [11, 13], the authors rely on a similarity flooding algorithm for computing the matrix of correspondences between models. [11] considers a forward and backward similarity propagation to compare data structures described with directed labelled graphs. However, the tool does not enable a fully automated matching because the user should manually adjust some matches. The match operator introduced in [13] measures the similarity between different versions of software units described using Statecharts. The similarity measuring combines a set of static and behavioural matchings. The behavioural matching is computed using a flooding algorithm and relies on the bisimulation notion presented in [22]. Flooding algorithms were also used for measuring the compatibility of behavioural models of Web services in [15, 16]. Our iteration process is very similar to similarity flooding algorithm but tackles the problem with a different angle by focusing only on non-bisimilar states.

[3] extends the simulation preorder to a quantitative setting. It presents three notions of distances (correctness, coverage, robustness), which resides in making each player of a simulation game pay a certain price for his/her choices. These distances are comparable to the global measure proposed in this paper. There is no local criterion used in their work.

[24] presents an approach (SpecDiff) to compute the differences between two LTSs obtained by compilation from CSP, representing the evolving behaviors of a concurrent program. SpecDiff considers LTSs as Typed Attributed Graphs (TAGs), in which states and transitions are encoded in finite dimensional vector spaces. It then computes a maximum common subgraph of two TAGs, which represents an optimal matching of states and transitions between two LTSs. This approach aims at pairing states and transitions for debugging purposes whereas we analyze the structure of both LTSs without mandatorily finding a match. Moreover, our approach is more general-purpose and not only designed for program debugging.

[23] aims at comparing state machines in terms of their language (the externally observable sequences of events that are permitted or not), and in terms of their structure (the actual states and transitions that govern the behaviour). The language comparison exploits model-based testing approach. The structure comparison uses what we call local criteria in our paper, by looking at the similarity of surrounding transitions and source/target states. They do not rely on any notion of distances as we did to compare the situation of both states in their respective LTSs. They do not focus on non-bisimilar states only as we do. As far as application is concerned, they apply their approach to reverse-engineering state machines from program traces.

[18, 19] define a distance between processes modelled as trees by computing the costs to transform one of the trees into the other. This notion of distance between processes is defined using coinduction. This approach applies in the case of both finite and infinite trees. The notion of k-bisimulation was introduced in [20]. It considers weak bisimulation and more specifically the weak equivalence notion introduced by Milner in [12]. K-bisimulation measures the number of actions to be hidden for establishing weak equivalence between two processes modelled using LTSs. Thoses measures are less precise than ours since they do not give any detailed measure of distance among the states of both LTSs. It is closer to our global measure of similarity, which gives a rough estimation of how far the two LTSs are one from another.

6 Concluding Remarks

We have presented in this paper a set of metrics that allows us to quantify the difference between the non-bisimilar parts of two LTSs. This similarity measure combines local and global criteria for computing a matrix that compares all non-bisimilar states in both LTSs. The computed matrix is used in a second step for computing a global measure of similarity that is helpful to distinguish totally different LTSs and almost bisimilar ones. Our approach is implemented in a tool and was applied on a set of about 100 examples for validating the ideas. Beyond that, we applied our solution to a concrete application area, namely the design of IoT applications by composition of objects. This case study allows us to show how our similarity measure can be used in practice to solve concrete problems.

As far as future work is concerned, we plan to extend our work to support other notions of bisimulations. Another perspective aims at taking advantage of all the values gathered in the similarity matrix to refine our comprehension of the differences between the two LTSs. We also plan to apply our similarity measure to other application domains such as the business process models matching or the debugging of concurrent software. Finally, we would like to work on the optimization of the tool support to make our approach scalable on large LTSs consisting of possibly millions of states/transitions.

References

1. C. Baier and J. Katoen. *Principles of Model Checking*. MIT Press, 2008.
2. L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella. When are Two Web Services Compatible? In *Proc. of TES'04*, volume 3324 of *LNCS*, pages 15–28. Springer, 2004.
3. P. Cerný, T. A. Henzinger, and A. Radhakrishna. Simulation Distances. In *Proc. of CONCUR'10*, volume 6269 of *LNCS*, pages 253–268. Springer, 2010.
4. D. Champelovier, X. Clerc, H. Garavel, Y. Guerte, F. Lang, C. McKinty, V. Powazny, W. Serwe, and G. Smeding. Reference Manual of the LNT to LOTOS Translator (Version 6.7). INRIA/VASY and INRIA/CONVECS, 153 pages, 2018.

5. F. Durán, G. Salaün, and A. Krishna. Automated Composition, Analysis and Deployment of IoT Applications. In *Proc. of TOOLS 2019*, volume 11771 of *LNCS*, pages 252–268. Springer, 2019.
6. H. Garavel, F. Lang, R. Mateescu, and W. Serwe. CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes. *STTT*, 15(2):89–107, 2013.
7. P. C. Kanellakis and S. A. Smolka. CCS Expressions, Finite State Processes, and Three Problems of Equivalence. *Inf. Comput.*, 86(1):43–68, 1990.
8. A. Krishna, M. L. Pallec, R. Mateescu, L. Noirie, and G. Salaün. IoT Composer: Composition and Deployment of IoT Applications. In *Proc. of ICSE'19, Montreal*, pages 19–22. IEEE / ACM, 2019.
9. A. Krishna, M. L. Pallec, R. Mateescu, L. Noirie, and G. Salaün. Rigorous Design and Deployment of IoT Applications. In *Proc. of FormaliSE'19*. ACM, 2019.
10. R. Mateescu and D. Thivolle. A Model Checking Language for Concurrent Value-Passing Systems. In *Proc. of FM'08*, volume 5014 of *LNCS*. Springer, 2008.
11. S. Melnik, H. Garcia-Molina, and E. Rahm. Similarity Flooding: A Versatile Graph Matching Algorithm and Its Application to Schema Matching. In *Proc. of ICDE'02*, pages 117–128. IEEE Computer Society, 2002.
12. R. Milner. *Communication and Concurrency*. Prentice Hall, 1989.
13. S. Nejati, M. Sabetzadeh, M. Chechik, S. M. Easterbrook, and P. Zave. Matching and Merging of Statecharts Specifications. In *Proc. of ICSE'07*, pages 54–64. IEEE Computer Society, 2007.
14. M. Ouederni, U. Fahrenberg, A. Legay, and G. Salaün. Compatibility Flooding: Measuring Interaction of Services Interfaces. In *Proc. of SAC'17*, pages 1334–1340. ACM, 2017.
15. M. Ouederni, G. Salaün, and E. Pimentel. Quantifying Service Compatibility: A Step beyond the Boolean Approaches. In *Proc. of ICSOC'10*, volume 6470 of *LNCS*, pages 619–626. Springer, 2010.
16. M. Ouederni, G. Salaün, and E. Pimentel. Measuring the Compatibility of Service Interaction Protocols. In *Proc. of SAC'11*, pages 1560–1567. ACM, 2011.
17. R. Paige and R. E. Tarjan. Three Partition Refinement Algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.
18. D. Romero-Hernández and D. de Frutos-Escrig. Defining Distances for All Process Semantics. In *Proc. of FMOODS/FORTE'12*, volume 7273 of *LNCS*, pages 169–185. Springer, 2012.
19. D. Romero-Hernández and D. de Frutos-Escrig. Coinductive Definition of Distances between Processes: Beyond Bisimulation Distances. In *Proc. of FORTE'14*, volume 8461 of *LNCS*, pages 249–265. Springer, 2014.
20. G. D. Ruvo, G. Lettieri, D. Martino, A. Santone, and G. Vaglini. k-Bisimulation: A Bisimulation for Measuring the Dissimilarity Between Processes. In *Proc. of FACS'15*, volume 9539 of *LNCS*, pages 181–198. Springer, 2015.
21. G. Salaün, L. Bordeaux, and M. Schaerf. Describing and Reasoning on Web Services using Process Algebra. In *Proc. of ICWS'04*, pages 43–50. IEEE, 2004.
22. O. Sokolsky, S. Kannan, and I. Lee. Simulation-Based Graph Similarity. In *Proc. of TACAS'06*, volume 3920 of *LNCS*, pages 426–440. Springer, 2006.
23. N. Walkinshaw and K. Bogdanov. Automated Comparison of State-Based Software Models in Terms of Their Language and Structure. *ACM Trans. Softw. Eng. Methodol.*, 22(2):13:1–13:37, 2013.
24. Z. Xing, J. Sun, Y. Liu, and J. S. Dong. Differencing Labeled Transition Systems. In *Proc. of ICFEM'11*, volume 6991 of *LNCS*, pages 537–552. Springer, 2011.