Debugging of Behavioural Models with CLEAR

Gianluca Barbon[1], Vincent Leroy[2], and Gwen Salaün[1(✉)]

[1] Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, 38000 Grenoble, France
gwen.salaun@inria.fr

[2] Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

Abstract. This paper presents a tool for debugging behavioural models being analysed using model checking techniques. It consists of three parts: (i) one for annotating a behavioural model given a temporal formula, (ii) one for visualizing the erroneous part of the model with a specific focus on decision points that make the model to be correct or incorrect, and (iii) one for abstracting counterexamples thus providing an explanation of the source of the bug.

1 Introduction

Model checking [2] is an established technique for automatically verifying that a behavioural model satisfies a given temporal property, which specifies some expected requirement of the system. In this work, we use Labelled Transition Systems (LTS) as behavioural models of concurrent programs. An LTS consists of states and labelled transitions connecting these states. An LTS can be produced from a higher-level specification of the system described with a process algebra for instance. Temporal properties are usually divided into two main families: safety and liveness properties [2]. Both are supported in this work. If the LTS does not satisfy the property, the model checker returns a counterexample, which is a sequence of actions leading to a state where the property is not satisfied.

Understanding this counterexample for debugging the specification is a complicated task for several reasons: (i) the counterexample may consist of many actions; (ii) the debugging task is mostly achieved manually (satisfactory automatic debugging techniques do not yet exist); (iii) the counterexample does not explicitly point out the source of the bug that is hidden in the model; (iv) the most relevant actions are not highlighted in the counterexample; (v) the counterexample does not give a global view of the problem.

The CLEAR tools (Fig. 1) aims at simplifying the debugging of concurrent systems whose specification compiles into a behavioural model. To do so, we propose a novel approach for improving the comprehension of counterexamples by highlighting some of the states in the counterexample that are of prime importance because from those states the specification can reach a correct part of the model or an incorrect one. These states correspond to decisions or choices that are particularly interesting because they usually provide an explanation of the

© The Author(s) 2019
T. Vojnar and L. Zhang (Eds.): TACAS 2019, Part I, LNCS 11427, pp. 386–392, 2019.
https://doi.org/10.1007/978-3-030-17462-0_26

source of the bug. The first component of the CLEAR toolset computes these specific states from a given LTS (AUT format) and a temporal property (MCL logic [5]). Second, visualization techniques are provided in order to graphically observe the whole model and see how those states are distributed over that model. Third, explanations of the bug are built by abstracting away irrelevant parts of the counterexample, which results in a simplified counterexample.

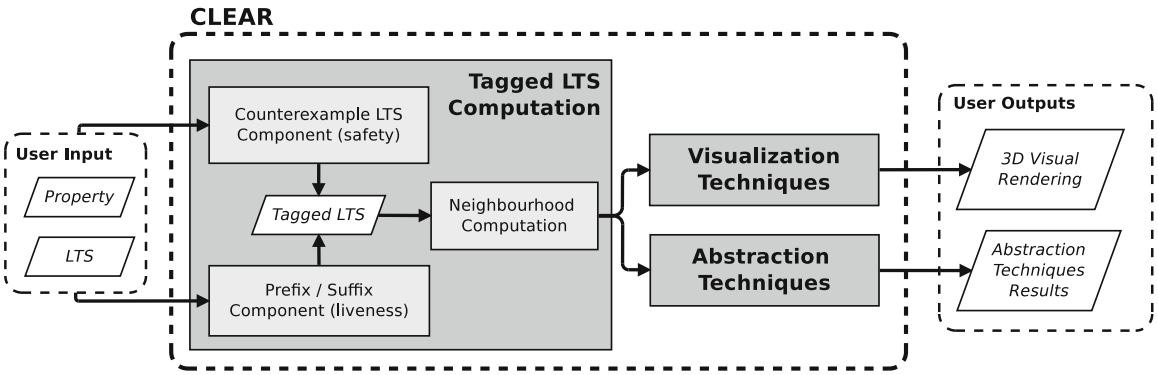

Fig. 1. Overview of the CLEAR toolset.

The CLEAR toolset has been developed mainly in Java and consists of more than 10K lines of code. All source files and several case studies are available online [1]. CLEAR has been applied to many examples and the results turn out to be quite positive as presented in an empirical evaluation which is also available online.

The rest of this paper is organised as follows. Section 2 overviews the LTS and property manipulations in order to compute annotated or tagged LTSs. Sections 3 and 4 present successively our techniques for visualizing tagged models and for abstracting counterexamples with the final objective in both cases to simplify the debugging steps. Section 5 describes experiments we carried out for validating our approach on case studies. Section 6 concludes the paper.

2 Tagged LTSs

The first step of our approach is to identify in the LTS parts of it corresponding to correct or incorrect behaviours. This is achieved using several algorithms that we define and that are presented in [3,4]. We use different techniques depending on the property family. As far as safety properties are concerned, we compute an LTS consisting of all counterexamples and compare it with the full LTS. As for liveness properties, for each state, we compute the set of prefixes and suffixes. Then, we use this information for tagging transitions as correct, incorrect or neutral in the full LTS. A correct transition leads to a behaviour that always satisfies the property, while an incorrect one leads to a behaviour that always violates the property. A neutral transition is common to correct and incorrect behaviours.

Once we have this information about transitions, we can identify specific states in the LTS where there is a choice in the LTS that directly affects the compliance with the property. We call these states and the transitions incoming to/outgoing from those states *neighbourhoods*.

There are four kinds of neighbourhoods, which differ by looking at their outgoing transitions (Fig. 2 from left to right): (1) with at least one correct transition (and no incorrect transition), (2) with at least one incorrect transition (and no correct transition), (3) with at least one correct transition and one incorrect transition, but no neutral transition, (4) with at least one correct transition, one incorrect transition and one neutral transition. The transitions contained in neighbourhood of type (1) highlight a choice that can lead to behaviours that always satisfy the property. Note that neighbourhoods with only correct outgoing transitions are not possible, since they would not correspond to a problematic choice. Consequently, this type of neighbourhood always presents at least one outgoing neutral transition. The transitions contained in neighbourhood of type (2), (3) or (4) highlight a choice that can lead to behaviours that always violate the property.

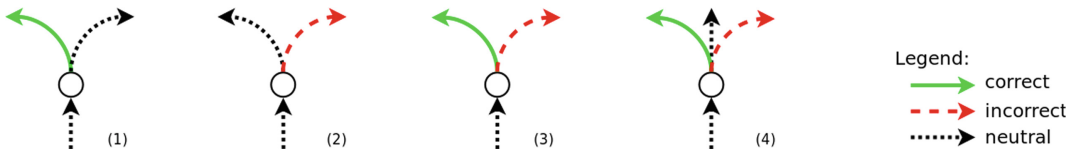

Fig. 2. The four types of neighbourhoods. (Color figure online)

It is worth noting that both visualization and counterexample abstraction techniques share the computation of the tagged LTS (correct/incorrect/neutral transitions) and of the neighbourhoods.

3 Visualization Techniques

The CLEAR visualizer provides support for visualizing the erroneous part of the LTS and emphasizes all the states (a.k.a. neighbourhoods) where a choice makes the specification either head to correct or incorrect behaviour. This visualization is very useful from a debugging perspective to have a global point of view and not only to focus on a specific erroneous trace (that is, a counterexample).

More precisely, the CLEAR visualizer supports the visualization of tagged LTSs enriched with neighbourhoods. These techniques have been developed using Javascript, the AngularJS framework, the bootstrap CSS framework, and the 3D force graph library. These 3D visualization techniques make use of different colors to distinguish correct (green), incorrect (red) and neutral (black) transitions on the one hand, and all kinds of neighbourhoods (represented with different shades of yellow) on the other hand. The tool also provides several functionalities in order to explore tagged LTSs for debugging purposes, the main one

being the step-by-step animation starting from the initial state or from any chosen state in the LTS. This animation keeps track of the already traversed states/transitions and it is possible to move backward in that trace. Beyond visualizing the whole erroneous LTS, another functionality allows one to focus on one specific counterexample and rely on the animation features introduced beforehand for exploring the details of that counterexample (correct/incorrect transitions and neighbourhoods).

Figure 3 gives a screenshot of the CLEAR visualizer. The legend on the left hand side of this figure depicts the different elements and colors used in the LTS visualization. All functionalities appear in the bottom part. When the LTS is loaded, one can also load a counterexample. On the right hand side, there is the name of the file and the list of states/transitions of the current animation. Note that transitions labels are not shown, they are only displayed through mouseover. This choice allows the tool to provide a clearer view of the LTS.

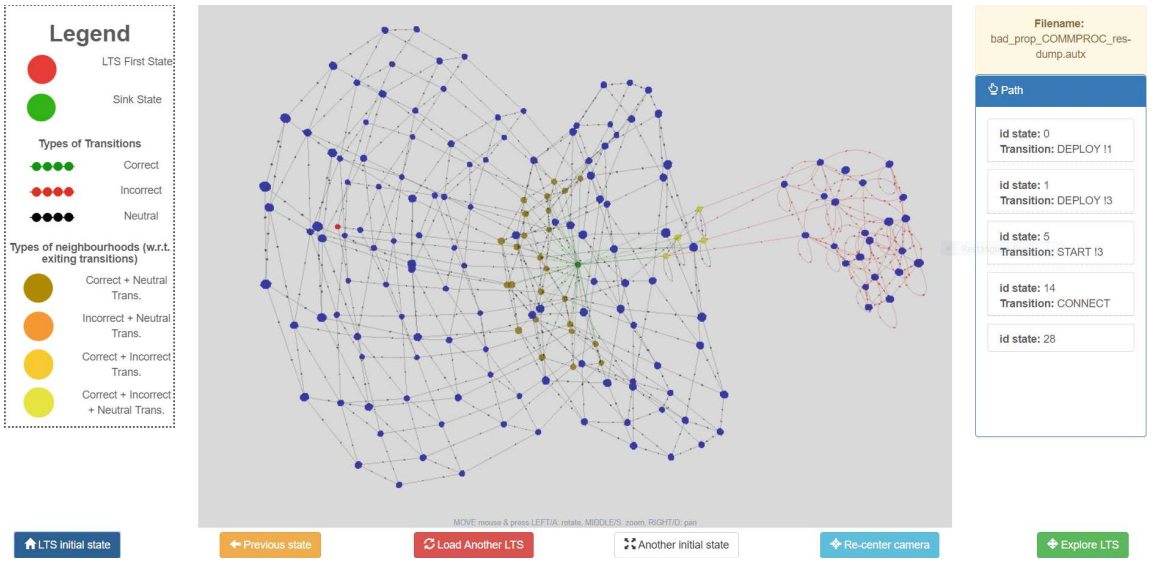

Fig. 3. Screenshot of the CLEAR visualizer. (Color figure online)

From a methodological point of view, it is adviced to use first the CLEAR visualizer during the debugging process for taking a global look at the erroneous part of the LTS and possibly notice interesting structures in that LTS that may guide the developer to specific kinds of bug. Step-by-step animation is also helpful for focusing on specific traces and for looking more carefully at some transitions and neighbourhoods on those traces. If the developer does not identify the bug using these visualization techniques, (s)he can make use of the CLEAR abstraction techniques presented in the next section.

4 Abstraction Techniques

In this section, once the LTS has been tagged using algorithms overviewed in Sect. 2, the developer can use abstraction techniques that aim at simplifying a

counterexample produced from the LTS and a given property. To do so we make a joint analysis of the counterexample and of the LTS enriched with neighbourhoods computed previously. This analysis can be used for obtaining different kinds of simplifications, such as: (i) an abstracted counterexample, that allows one to remove from a counterexample actions that do not belong to neighbourhoods (and thus represent noise); (ii) a shortest path to a neighbourhood, which retrieves the shortest sequence of actions that leads to a neighbourhood; (iii) improved versions of (i) and (ii), where the developer provides a pattern representing a sequence of non-contiguous actions, in order to allow the developer to focus on a specific part of the model; (iv) techniques focusing on a notion of distance to the bug in terms of neighbourhoods. For the sake of space, we focus on the abstracted counterexample in this paper.

Abstracted Counterexample. This technique takes as input an LTS where neighbourhoods have been identified and a counterexample. Then, it removes all the actions in the counterexample that do not represent incoming or outgoing transitions of neighbourhoods. Figure 4 shows an example of a counterexample where two neighbourhoods, highlighted on the right side, have been detected and allow us to identify actions that are preserved in the abstracted counterexample.

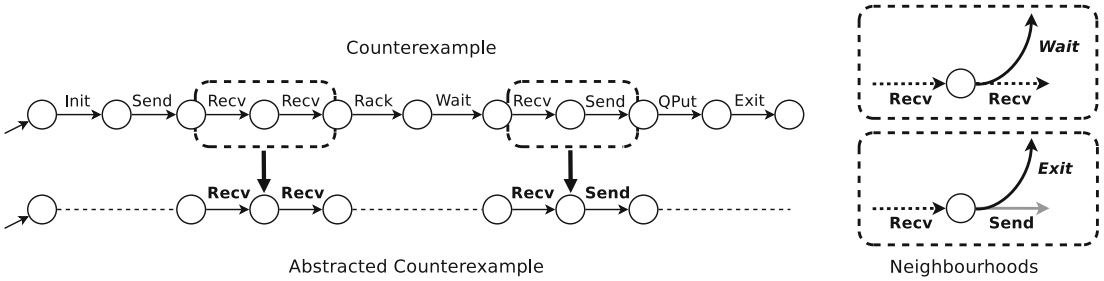

Fig. 4. Abstracted counterexample.

5 Experiments

We carried out experiments on about 100 examples. For each one, we use as input a process algebraic specification that was compiled into an LTS model, and a temporal property. As far as computation time is concerned, the time is quite low for small examples (a few seconds), while it tends to increase w.r.t. the size of the LTS when we deal with examples with hundreds of thousands of transitions and states (a few minutes). In this case, it is mainly due to the computation of tagged LTSs, which is quite costly because it is based on several graph traversals. Visualization techniques allowed us to identify several examples of typical bugs with their corresponding visual models. This showed that the visualizations exhibit specific structures that characterize the bug and are helpful for supporting the developer during his/her debugging tasks. As for abstraction techniques, we observed some clear gain in length (up to 90%) between the original counterexample and the abstracted one, which keeps only relevant actions using our approach and thus facilitates the debugging task for the developer.

We also carried out an empirical study to validate our approach. We asked 17 developers, with different degrees of expertise, to find bugs on two test cases by taking advantage of the abstracted counterexample techniques. The developers were divided in two groups, in order to evaluate both test cases with and without the abstracted counterexample. The developers were asked to discover the bug and measure the total time spent in debugging each test case. We measured the results in terms of time, comparing for both test cases the time spent with and without the abstracted counterexample. We observed a gain of about 25% of the total average time spent in finding the bug for the group using our approach. We finally asked developers' opinion about the benefit given by our method in detecting the bug. Most of them agreed considering our approach helpful.

The CLEAR toolset is available online [1] jointly with several case studies and the detailed results of the empirical study.

6 Concluding Remarks

In this paper, we have presented the CLEAR toolset for simplifying the comprehension of erroneous behavioural specifications under validation using model checking techniques. To do so, we are able to detect the choices in the model (neighbourhood) that may lead to a correct or incorrect behaviour, and generate a tagged LTS as result. The CLEAR visualizer takes as input a tagged LTS and provides visualization techniques of the whole erroneous part of the model as well as animation techniques that help the developer to navigate in the model for better understanding what is going on and hopefully detect the source of the bug. The counterexample abstraction techniques are finally helpful for building abstractions from counterexamples by keeping only relevant actions from a debugging perspective. The experiments we carried out show that our approach is useful in practice to help the designer in finding the source of the bug(s).

References

1. CLEAR Debugging Tool. https://github.com/gbarbon/clear/
2. Baier, C., Katoen, J.: Principles of Model Checking. MIT Press, Cambridge (2008)
3. Barbon, G., Leroy, V., Salaün, G.: Debugging of concurrent systems using counterexample analysis. In: Dastani, M., Sirjani, M. (eds.) FSEN 2017. LNCS, vol. 10522, pp. 20–34. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-68972-2_2
4. Barbon, G., Leroy, V., Salaün, G.: Counterexample simplification for liveness property violation. In: Johnsen, E.B., Schaefer, I. (eds.) SEFM 2018. LNCS, vol. 10886, pp. 173–188. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-92970-5_11
5. Mateescu, R., Thivolle, D.: A model checking language for concurrent value-passing systems. In: Cuellar, J., Maibaum, T., Sere, K. (eds.) FM 2008. LNCS, vol. 5014, pp. 148–164. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-68237-0_12

Open Access This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.