Visual Debugging of Behavioural Models

Gianluca Barbon*, Vincent Leroy†, Gwen Salaün*, and Emmanuel Yah†
*Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, 38000 Grenoble, France
†Univ. Grenoble Alpes, CNRS, Grenoble INP, LIG, 38000 Grenoble, France

***Abstract*—In this paper, we present the CLEAR visualizer tool, which supports the debugging task of behavioural models being analyzed using model checking techniques. The tool provides visualization techniques for simplifying the comprehension of counterexamples by highlighting some specific states in the model where a choice is possible between executing a correct behaviour or falling into an erroneous part of the model. Our tool was applied successfully to many case studies and allowed us to visually identify several kinds of typical bugs.**
Video URL: https://youtu.be/nJLOnRaPe1A

I. 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. Model checking [1] is an established technique for automatically verifying that a model, e.g., a Labelled Transition System (LTS), satisfies a given temporal property, e.g., the absence of deadlocks. When the model violates 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 can contain hundreds (even thousands) of actions, (ii) the debugging task is mostly achieved manually, (iii) the counterexample does not explicitly highlight the source of the bug that is hidden in the model, and (iv) the counterexample describes only one occurrence of the bug and does not give a global view of the problem with all its occurrences.

The CLEAR visualizer aims at simplifying the debugging of concurrent systems whose specification compiles into a behavioural model. To do so, the tool first detects some specific 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 source of the bug. Once all these specific states have been identified, the tool proposes several visualization techniques in order to graphically observe the whole model and see how those states are distributed over that model.

More precisely, the CLEAR visualizer takes as input a behavioural model (LTS) describing all possible executions of a system. This LTS can be obtained by compilation from a higher-level textual specification language such as process algebra. Given such an LTS and a temporal property, in a first step, the tool first extracts all the counterexamples from the original model containing all the executions. This procedure is able to collect these counterexamples in a new LTS, called counterexample LTS, maintaining a correspondence with the original model. Second, an algorithm is applied for comparing the counterexample LTS with the original LTS and then identifying specific states where counterexamples and correct behaviours, that share a common prefix, split in different paths. Actions at those states are relevant since they are responsible for the choice between a correct and incorrect behaviour.

In a subsequent step, the tool relies on 3D visualization techniques to have a global view of the model where correct/incorrect transitions and the states where those transitions take place are highlighted using different colors. The CLEAR visualizer provides several functionalities to facilitate the manipulation of those models (forward/backward step-by-step animation, counterexample visualization, zoom in/out, etc.). The tool was applied on several case studies for evaluation purposes and these experiments allowed us to identify several interesting visualization patterns that correspond to typical cases of bugs. As a result, our techniques can be used for visual debugging in order to identify the source of the bug by looking at the graphical representation of the model extended with the aforementioned information.

In the rest of the paper, we do not present pre-processing steps for computing counterexamples and for analyzing them in order to identify correct/incorrect transitions and states where there are choices between those transitions, see [2] for details. We prefer to focus on the visualization techniques provided by our tool that we present in Section II. Section III illustrates these visualization techniques on three case studies. Section IV concludes this paper.

II. Visual Debugging Techniques

This section presents the CLEAR visualization techniques. The goal is to provide a support for visualizing the erroneous part of the LTS (a.k.a. tagged LTS) and for emphasizing all the states (a.k.a. neighbourhoods) where a choice is taken and makes the specification either lead to correct or incorrect behaviour. This visualization is very useful to have a global point of view during the debugging process and not only to focus on a specific erroneous trace (that is, a single counterexample). Section III will show several examples with unsatisfied properties and how our approach allows one in practice to visually identify bugs. The tool is available online [3] with all the examples presented in this paper.

Figure 1 gives an overview of our tool support. We rely on the LNT process algebra [4] as input specification language and MCL [5] for describing temporal properties. Note that

these are the only two inputs given by the developer. The rest of the approach and all generated models are computed automatically. Given those inputs, we rely on the CADP toolbox [6] to generate first the corresponding LTS model and second to compute the counterexample LTS. Finally, the tagged LTS is computed (using a Java program we implemented) by detecting first correct/incorrect transitions and then neighbourhoods.

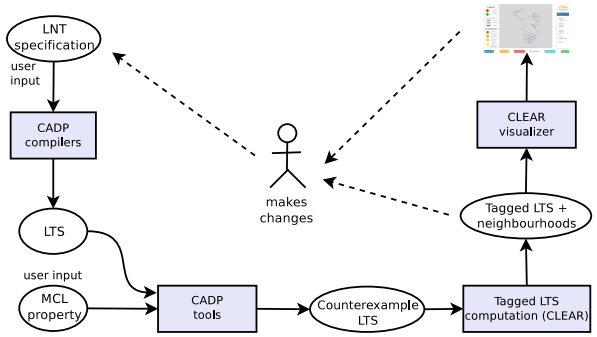

Fig. 1. Overview of the CLEAR tool.

We now present with more details the CLEAR visualizer, which supports the visualization of tagged LTSs 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[1] (black) transitions on the one hand, and all kinds of neighbourhoods[2] (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 visualize one specific counterexample as well and rely on the animation features introduced beforehand for exploring the details of that counterexample (correct/incorrect transitions and neighbourhoods).

As far as usability is concerned, here is what we advocate for using our tool from a methodological perspective. First, the developer can use CLEAR for taking a global look at the erroneous part of the LTS and possibly notice interesting structures in that LTS that may guide him/her to specific kinds of bug (see Section III for more details). Second, the developer can dive into the LTS by focusing on some special states/neighbourhoods and use the step-by-step animation features for that exploration. Finally, (s)he can load and visualize some specific counterexample in order to focus on a particular trace of interest (the shortest counterexample for instance) and use the CLEAR visualization functionalities to better understand the transitions and neighbourhoods on that specific trace. At any step, the developer can use the CLEAR tool outputs in order to make changes on the input LNT specification as depicted in Figure 1.

Figure 2 gives a screenshot of the CLEAR visualizer. One can see the different colors used in the LTS visualization with the legend on the left hand side. All functionalities appear in the bottom part. When the LTS is loaded, there is also the option to 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.

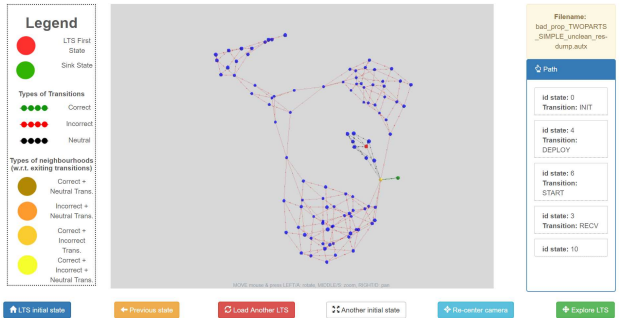

Fig. 2. Screenshot of the CLEAR visualizer tool.

III. Visual Debugging in Action

In this section, we present several examples of erroneous LNT specifications. Each specification is accompanied with a temporal property characterizing a requirement that is supposed to be satisfied by the specification. Model checking techniques are used and confirm that each property is violated by the corresponding specification. For each example, we show how our techniques can be helpful for visual debugging. In this section, we chose LNT specifications (see [4] for an introduction to LNT) exhibiting typical bugs inherent to concurrent systems.

Interleaving bug. The LNT specification given in Figure 3 consists of three parts in sequence. The initial part (INITi actions) and the final part (CLOSEi actions) are used, respectively, for initialisation and closing purposes. The central part consists of a parallel composition where several EXECi actions are executed in parallel with another branch where there is a 'select' construct, which allows one to choose between two branches with several SENDi actions. The property states that a SEND2 action should never be followed by a SEND1 action (see [5] for an introduction to MCL): ([true* . 'SEND2' . true* . 'SEND1' . true*] false).

The erroneous LTS is given in Figure 4. The red state on the left hand part corresponds to the initial state. Then,

[1]Neutral transitions can lead to both correct and incorrect behaviour.

[2]There are four kinds of neighbourhoods: (i) with at least one correct transition (and no incorrect transition), (ii) with at least one incorrect transition (and no correct transition), (iii) with at least one correct transition and one incorrect transition, but no neutral transition, (iv) with at least one correct transition, one incorrect transition and one neutral transition.

```
process Main [ EXEC1, EXEC2, EXEC3, EXEC4, EXEC5, LOSS: none,
               INIT1, INIT2, INIT3: none,
               CLOSE1, CLOSE2, CLOSE3, CLOSE4: none,
               SEND1, SEND2, SEND3, SEND4: none] is

  par      (* initialisation part *)
    INIT1 || INIT3; INIT1 || INIT1; INIT2
  end par;
  par      (* central part *)
    EXEC1; EXEC2; EXEC3; EXEC4; EXEC5
  ||
    select
      par
        SEND2; SEND3 || LOSS
      end par;
      SEND1; SEND4
    []
      SEND1;
      par
        SEND2 || SEND2; SEND3 || LOSS
      end par;
      SEND4
    end select
  end par;
  select   (* closing part *)
      par
        CLOSE3; CLOSE2 || CLOSE4; CLOSE1 || CLOSE2 || CLOSE1
      end par
   []
      CLOSE1; CLOSE2
  end select

end process
```

Fig. 3. LNT code for the interleaving bug.

we can clearly distinguish the initial part (left) with black transitions because all these transitions can lead to a possibly erroneous part of the system. Likewise, we can see on the right hand part of this figure the closing part of the specification where all transitions are incorrect (red) and where the bug cannot be avoided. These two parts (entirely black or entirely red) can be viewed as *noise* or actions that are not helpful from a debugging perspective. In contrast the central part of the figure is highly interesting. There are six neighbourhood states in that part of the LTS corresponding to a choice between executing a correct part of the specification (avoiding the sequence with a SEND2 action followed by a SEND1 action) leading to the white state (sink state) that abstracts the correct part of the LTS, or executing an incorrect part of the specification. There are six choices because this choice is in parallel with the sequence of EXECi actions and can then appear at different states (interleaving). This is typical of a bug which is interleaved with other actions, looking in that case like a *spider web* due to the attraction of the sink state in the visualization.

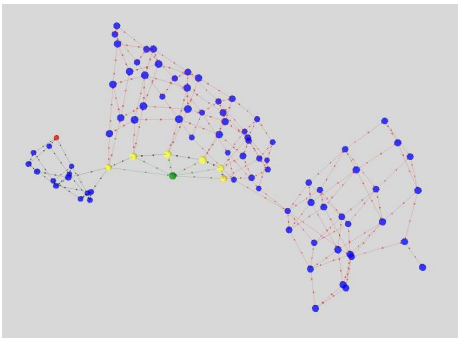

Fig. 4. Counterexample LTS visualization of the interleaving bug.

Iteration bug. This LNT specification (Fig. 5) exhibits a looping process with a nondeterministic choice executed at each iteration of that loop. In one of the two branches of the choice, there is a parallel construct that allows one to obtain a LOSS action followed by a REC action, which is the sequence of actions that must not happen according to the following MCL property: ([true* . 'LOSS' . 'REC' . true*] false).

```
process Main [WAIT, INIT, REC, EXEC1, EXEC2, LOSS, STORE: none] is
  var I,K : nat in
    I := 0;
    K := 10;
    for I:=0 while I<K by I:=I+1 loop
      WAIT;
      select
        par
          EXEC1; STORE || LOSS
        end par
      []
        par
          REC; EXEC2 || LOSS
        end par;
        I:=10
      end select
    end loop
  end var
end process
```

Fig. 5. LNT code for the iteration bug.

The visualization of the erroneous part of the LTS corresponding to this LNT specification looks like a *flower* and is given in Figure 6. Each *petal* corresponds to an iteration of the loop. There is a neighbourhood present at the beginning of each iteration, which represents a choice between reaching the incorrect behaviour, going to the sink state (both at the center of the picture), or continuing to the next petal. All the petals consist of neutral (black) transitions because the bug can still be avoided. There is a part of the LTS with red transitions, which is reached after executing an incorrect transition in one of the aforementioned neighbourhoods. After nine iterations, executing at each iteration the first branch of the select construct, a final correct transition leads to the sink state and makes the whole specification definitely avoid the incorrect part of the behaviour.

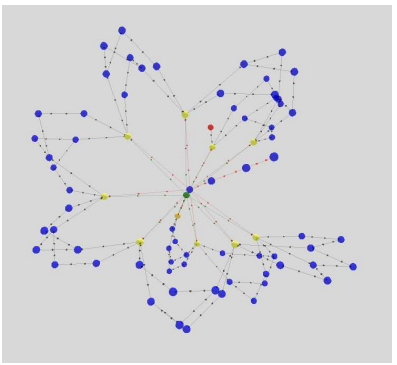

Fig. 6. Counterexample LTS visualization of the iteration bug.

Causality bug. The last example is a producer-consumer system. The LNT specification consists of about 100 lines of code and is available online [3]. The specification is composed of three main processes: a producer process, a consumer process and a process that can either be a consumer or a producer. This last process is given in Figure 7. Each process can loop infinitely or break the loop and terminate the execution. A

deployer process is also part of the specification in order to initiate the three other processes. The provided property states that a process cannot consume if something has not been produced before. This is written in MCL as follows: [(not PRODUCE)* . CONSUME . true*] false. The specification violates this property when the PRODCONS process (Fig. 7) acts as a consumer, because it can consume without ensuring that PRODUCE has been performed beforehand.

```
process PRODCONS  [  CONNECT, READY, SYNC, WAIT : none,
                     DEPLOY, START, IAMPRODUCER, IAMCONSUMER : WHOAMI_C,
                     CONSUME, PRODUCE : none
                  ] is
  var whoami : bool in
    DEPLOY(1 of nat);
    START(1 of nat);
    CONNECT;
    select
      whoami := true; IAMPRODUCER(1 of nat)
    []
      whoami := false; IAMCONSUMER(1 of nat)
    end select;
    WAIT;
    READY;
    if (whoami) then
      loop L in
        select
          NULL [] break L
        end select;
        par
          WAIT || PRODUCE; SYNC
        end par
      end loop
    else
      loop L in
        select
          NULL [] SYNC [] break L
        end select;
        par
          WAIT || CONSUME
        end par
      end loop
    end if
  end var
end process
```

Fig. 7. LNT code for the causality bug.

The erroneous LTS with colored transitions and neighbourhoods is given in Figure 8. The LTS is divided into three parts. The initial part (right) represents the portion of code in which every process performs the deployment and this part of the model has no impact on the bug (no neighbourhoods and all neutral transitions). Then, a set of neighbourhoods of the same type is present between the first part of the LTS (right) and the second (central) one. These neighbourhoods have all a correct and a neutral transition, and represent the first choice that contributes to the cause of the bug (when the PRODCONS process decides to be a consumer). Those neighbourhoods can be viewed as a *frontier* between the initial and central part of the LTS. All the correct transitions are directed to the sink state. A second frontier is present between the central part of the LTS and the third part (left, with all red transitions). This frontier is composed of neighbourhoods that represent the second cause of the bug, that happens when a CONSUME action has been performed without an initial PRODUCE action. The figure with the two frontiers helps in understanding that there is causality between both kinds of neighbourhoods.

For the sake of space, we have only presented three typical examples of bugs and their visualization, but there are other interesting ones, e.g., a model where the whole LTS is false (only red transitions), or a specification where the bug can be reached from a single neighbourhood, which represents a

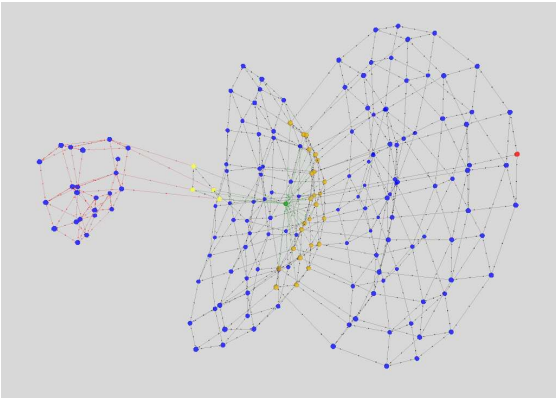

Fig. 8. Counterexample LTS visualization of the causality bug.

mandatory and unique choice to obtain the bug (as depicted in Figure 2). Those systems as well as other interesting examples are available online [3]. It is also worth noting that our visualization techniques are not always helpful because, in some cases, nothing can be deduced from the visual model or because the model is too large in terms of number of states/transitions. In those situations, the designer can use the step-by-step animation techniques presented in Section II.

IV. Concluding Remarks

In this paper, we have presented tool support for simplifying the comprehension of erroneous behavioural specifications under validation using model checking techniques. To do so, we focus on the choices in the model (neighbourhood) that may lead to a correct or incorrect behaviour. By looking more carefully at those states, we can better understand the source of the bug. The CLEAR visualizer provides visualization techniques of behavioural models (LTSs) that take into account this notion of correct/incorrect transitions as well as neighbourhood states. The tool support we implemented does not only provide visualization techniques of the whole erroneous part of the model but also 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. Last but not least, we have presented several examples of typical bugs where the visualizations exhibit specific structures that characterize the bug and are helpful for supporting the developer during his/her debugging tasks.

References

[1] C. Baier and J. Katoen, *Principles of Model Checking*. MIT Press, 2008.

[2] G. Barbon, V. Leroy, and G. Salaün, "Debugging of Concurrent Systems Using Counterexample Analysis," in *Proc. of FSEN'17*, ser. LNCS, vol. 10522. Springer, 2017, pp. 20–34.

[3] "CLEAR Debugging Tool." https://github.com/gbarbon/clear/.

[4] D. Champelovier *et al.*, "Reference Manual of the LNT to LOTOS Translator (Version 6.7)," 2018, INRIA/CONVECS, 153 pages.

[5] R. Mateescu and D. Thivolle, "A Model Checking Language for Concurrent Value-Passing Systems," in *Proc. of FM'08*, ser. LNCS, vol. 5014. Springer, 2008.

[6] H. Garavel, F. Lang, R. Mateescu, and W. Serwe, "CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes," *STTT*, vol. 15, no. 2, pp. 89–107, 2013.