Clusters of Faulty States for Debugging Behavioural Models

Irman Faqrizal and Gwen Salaün
Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG 38000 Grenoble, France

***Abstract*—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 is an established technique for automatically finding bugs by verifying that a model satisfies a given temporal property. 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 or program is a complicated task because the counterexample gives only a partial view of the source of the problem, and because there is usually little support beyond that counterexample to identify the source of the problem.**

In this paper, we focus on behavioural models (Labelled Transition Systems) and we propose some techniques for simplifying the debugging of erroneous models. We first focus on the erroneous part of the model and we detect specific states (called faulty states) where a choice is possible between executing a correct behaviour or falling into an erroneous part of the model. The goal of this paper is to group these faulty states into clusters. Clusters help the user to identify the source of the bug since each cluster of states provides some information about the bug. We implemented this technique into a tool, which allows the visualization of the faulty model and the computation of clusters.

***Index Terms*—behavioural models, model checking, debugging, Internet of Things**

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), obtained from higher-level specification languages such as process algebra 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, (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 idea behind our debugging techniques is that we extract from the whole behavioural model the part which does not satisfy the given property. Then, in that LTS, we identify some specific states from which the specification can reach a correct part of the model or an incorrect one. These *faulty* states correspond to decision points or choices that are particularly interesting because they usually point out a part of the model and specification that may cause the bug. Once all these specific states have been identified, visualization techniques are used to graphically observe the whole model and see how those states are distributed over that model. This visualization is helpful but not enough to understand the source of the bug because in many cases there are plenty of faulty states. In this work, we propose to build groups or clusters of faulty states to simplify their comprehension. A cluster gathers faulty states belonging to the same category and corresponding to the same problem in the original program. A cluster is also more helpful than a single faulty state because it summarizes the information about all states which are part of a same cluster.

More precisely, our approach takes as input a behavioural model (LTS) describing all possible executions of a system. This LTS is usually obtained by compilation from a higher-level textual specification language such as process algebra (LNT [2] in our work). Given such an LTS and a temporal property, we apply existing results [3], [4] to extract the erroneous part of the LTS and identify in this LTS all faulty states. Starting from here, we have proposed a solution to traverse this LTS and compute clusters of faulty states. A cluster is a set of faulty states with the same type[1], sharing a set of common labels, and all states in a cluster are reachable from one of them. A cluster is useful because it regroups states that may be numerous but actually concern the same buggy portion of the program / model. The computation of the clusters is fully automated by an extension of the CLEAR tool [5]. For validation purposes, we have applied this approach to many examples. In particular, we will show in this paper how our techniques help the user to better understand where the bugs come from on a case study from the Internet of Things (IoT) area.

To sum up, the main contributions of this paper are: (i) a formal notion of cluster of faulty states; (ii) a set of algorithms for automatically detecting all the clusters given an LTS model and a temporal property; (iii) a tool implementing the computation of clusters; (iv) the illustration of our approach on a real-world case study from the Internet of Things area.

The rest of this paper is organised as follows. Section II in-

[1] We will see in this paper that there are several types of faulty states.

troduces behavioural models and model checking techniques. Section III presents the notion of faulty state, which is at the heart of the debugging techniques used in this work. Section IV presents our definition of clusters and how they are computed. Section V illustrates the application of our approach on a real-world case study in the IoT area. Section VI presents related work. Section VII concludes the paper.

II. Background

In this work, we adopt Labelled Transition System (LTS) as 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 states; $s^0 \in S$ is the initial state; 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 is produced from a higher-level specification of the system described with a process algebra for instance. Specifications can be compiled into an LTS using specific compilers. In this work, we use LNT as specification language [2] and compilers from the CADP toolbox [6] for obtaining LTSs from LNT specifications. However, our approach is generic in the sense that it applies on LTSs produced from any specification language and any compiler/verification tool. An LTS can be viewed as all possible executions of a system. One specific execution is called a *trace*.

Definition 2: (Trace) Given an LTS $M = (S, s^0, \Sigma, T)$, a trace of size $n \in \mathbb{N}$ is a sequence of labels $l_1, l_2, \ldots, l_n \in \Sigma$ such that $s^0 \xrightarrow{l_1} s_1 \in T, s_1 \xrightarrow{l_2} s_2 \in\ T, \ldots, s_{n-1} \xrightarrow{l_n} s_n \in T$. The set of all traces of M is written as $t(M)$.

Model checking consists in verifying that an LTS model satisfies a given temporal property φ, which specifies some expected requirement of the system. Temporal properties are usually divided into two main families: safety and liveness properties [1]. In this work, we focus on safety properties, which are widely used in the verification of real-world systems. Safety properties state that "*something bad never happens*". A safety property is usually formalised using a temporal logic (we use MCL [7] in this work). It can be semantically characterised by an infinite set of traces t_φ, corresponding to the traces that violate the property φ in an LTS. If the LTS model does not satisfy the property, the model checker returns a *counterexample*, which is one of the traces characterised by t_φ.

Definition 3: (Counterexample) Given an LTS $M = (S, s^0, \Sigma, T)$ and a property φ, a counterexample is any trace which belongs to $t(M) \cap t_\varphi$.

III. Extended LTS Models

This approach takes as input a specification, which compiles into an LTS model, and a temporal property. The original idea of this work is to identify decision points where the specification (and the corresponding LTS model) goes from a (potentially) correct behaviour to an incorrect one. These choices turn out to be very useful to understand the source of the bug. These decision points are called *faulty states* in the LTS model.

In order to detect these faulty states, we first need to categorize the transitions in the model into different types. The transition type allows to highlight the compliance with the property of the paths in the model that traverse that given transition. Transitions in the LTS can be categorized into three types:

- *correct transitions*, which belong to paths in the model that represent behaviours which always satisfy the property.
- *incorrect transitions*, which belong to paths in the model that represent behaviours which always violate the property.
- *neutral transitions*, which belong to portions of paths in the model which are common to correct and incorrect behaviours.

We add the information concerning the detected transitions type (correct, incorrect and neutral transitions) to the initial LTS in the form of tags. We define the set of transition tags as $\Gamma = \{correct, incorrect, neutral\}$. Given an LTS $M = (S, s^0, \Sigma, T)$, a tagged transition is represented as $s \xrightarrow{(l,\gamma)} s'$, where $s, s' \in S$, $l \in \Sigma$ and $\gamma \in \Gamma$. Thus, an LTS in which each transition has been tagged with a type is called *tagged LTS*.

Definition 4: (Tagged LTS) Given an LTS $M = (S, s^0, \Sigma, T)$, and the set of transition tags Γ, the tagged LTS is a tuple $M_T = (S_T, s^0_T, \Sigma_T, T_T)$ where $S_T = S$, $s^0_T = s^0$, $\Sigma_T = \Sigma$, and $T_T \subseteq S_T \times \Sigma_T \times \Gamma \times S_T$.

The tagged LTS where transitions have been typed allows us to identify faulty states in which an incoming neutral transition is followed by a choice between at least two transitions with different types (correct, incorrect, neutral). Such a faulty state consists of all the neutral incoming transitions and all the outgoing transitions.

Definition 5: (Faulty State) Given the tagged LTS $M_T = (S_T, s^0_T, \Sigma_T, T_T)$, a state $s \in S_T$, such that $\exists t = s' \xrightarrow{(l,\gamma)} s \in T_T$, t is a neutral transition, and $\exists t' = s \xrightarrow{(l,\gamma)} s'' \in T_T$, t' is a correct or an incorrect transition, the faulty state s consists of the set of transitions $T_{nb} \subseteq T_T$ such that for each $t'' \in T_{nb}$, either $t'' = s' \xrightarrow{(l,\gamma)} s \in T_T$ or $t'' = s \xrightarrow{(l,\gamma)} s''' \in T_T$.

By looking at outgoing transitions of a faulty state, we can identify four categories of faulty states (Figure 1):

1) with at least one correct transition and one neutral transition (no incorrect transition),
2) with at least one incorrect transition and one neutral transition (no correct transition),
3) with at least one correct and one incorrect transition (no neutral transition), and
4) with at least one correct, one incorrect, and one neutral transition.

The visualization techniques give to the developer a graphical representation of the tagged LTS extended with faulty states, where correct/incorrect/neutral transitions and faulty

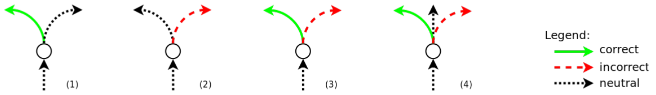

Fig. 1. The Four Types of Faulty States

states are highlighted. These 3D visualization techniques make use of different colours to distinguish correct (green), incorrect (red) and neutral (black) transitions on the one hand, and all kinds of faulty states (represented with different shades of yellow) on the other hand. The goal of this visual rendering is to provide a support for visualizing the erroneous part of the tagged LTS. This visualization emphasizes all the faulty states where a choice is taken and makes the specification either head to a correct or an incorrect behaviour.

Figure 2 gives an example of visualization obtained with the aforementioned techniques. On this figure one can see a first group of orange states in the middle and a second group of yellow states on the left hand side. We will show in the next section how we formally define the notion of cluster and how we compute them automatically.

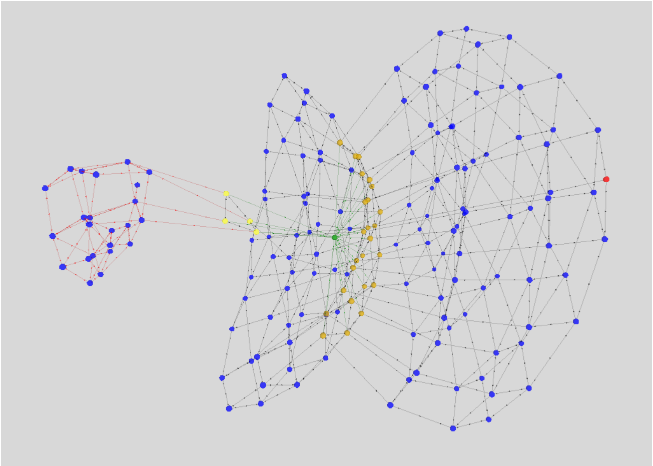

Fig. 2. Example of Tagged LTS and Faulty States

The reader interested in more details, in particular regarding the algorithms for computing tagged LTSs and for detecting faulty states, can refer to [3], [4].

IV. Clusters of Faulty States

Faulty states are helpful for debugging purposes, but in realistic examples the number of faulty states can be rather high. This is due to the size of the whole state space (we focus on the wrong part of the state space here), which can consist of thousands of states and transitions, and to the enumerative approach used for generating the LTS, which exhibit all possible executions of the corresponding specification. We noticed that there are often several faulty states for a same portion of code. This comes from the fact that we focus on concurrent specification languages where the interleaving of actions is a common construct and widely used in these specifications. As a consequence, faulty states often come by groups. All faulty states in a group share common characteristics. We will present in this section the definition of cluster and, in a second step, how these clusters are automatically computed. In Section V, we will show how clusters help to identify bugs in the corresponding models and specifications.

Definition. A cluster of faulty states shares the following features. First, a cluster stores faulty states of a same category or type. As an example, a cluster consists of faulty states where each state is of type (3), that is, with correct and incorrect outgoing transitions only. Second, all faulty states within a same cluster must share at least one common label appearing on their outgoing transitions. Third, there is one state in the cluster from which all the other states in the cluster are reachable.

Definition 6: (Cluster) Given a tagged LTS $M_T = (S_T, s_T^0, \Sigma_T, T_T)$ with the set of faulty states $FS \subseteq S_T$, a cluster C is the maximal subset of faulty states FS where each $s \in C$ has at least one outgoing transition with a same label l, all s are in the same faulty state category, and there exists one $s' \in C$ such that all states in $C \backslash \{s'\}$ are reachable from s'.

When computing clusters, beyond storing the set of faulty states, the category, and the common labels, we also keep all labels appearing on outgoing transitions for all faulty states in the cluster. This is useful to precisely identify the portion of the specification which should be analyzed by the developer in order to find the bug.

Computation. To compute clusters of faulty states, our algorithm traverses the tagged LTS until it finds a faulty state which is not part of any cluster yet. Then, it checks if the corresponding faulty state is a valid state to start a cluster. The next step is to traverse the LTS starting from this state, to compute the set of faulty states, the common labels, and the other involved labels that constitutes the cluster, and add it to the set of clusters. Thus, our algorithm consists of three main parts. The first part is to traverse the whole LTS based on a Depth-First Search (DFS) algorithm. The next part is to traverse backward from a faulty state to check if this state is the first faulty state in the cluster. Lastly, the algorithm traverses forward from the first faulty state in a new cluster, until the cluster is entirely computed. These three traversals are implemented using recursive algorithms.

The main algorithm is Algorithm 1. This algorithm takes as input the first/initial state of the LTS (*fState*), the current state of the traversal (*cState*), the set of visited states (*visited*) to avoid loops and unnecessary computations, and the set of clusters (*clusters*) which is also the output returned when the algorithm ends. There are also additional temporary variables declared in the algorithm: *tmpVisited* and *cVisited* keep track of visited states, *tmpFS* and *FS* are sets of faulty states, and *cLbl* and *iLbl* are sets of common labels and involved labels, respectively. These additional variables are also used as input and output for the other algorithms (2 and 3).

Algorithm 1 ComputeClusters

Inputs: First State, Current State, Set of Visited States, Set of Clusters

Output: Set of Clusters

```
1:  if $cState \in visited$ then
2:      return
3:  else
4:      $visited.Add(cState)$
5:      if $cState.IsFaultyState()$
6:              and $cState \notin clusters$ then
7:          $tmpVisited \leftarrow \{\}$
8:          $tmpFS \leftarrow \{\}$
9:          $cLbl \leftarrow cState.Labels$
10:         checkCluster($cState$, $cState$, $tmpVisited$,
11:                             $tmpFS$, $cLb$)
12:         if $tmpFS.Count(cState)$ = $|cState.IncTrans|$
13:                 or $|tmpFS|$ = 1 then
14:             $FS \leftarrow \{\}$
15:             $cVisited \leftarrow \{\}$
16:             $iLbl \leftarrow \{\}$
17:             traverseCluster($cState$, $cState$, $cVisited$,
18:                             $FS$, $cLbl$, $iLbl$, $clusters$)
19:             $cluster = createCluster(FS, cLbl, iLbl)$
20:             $clusters.Add(cluster)$
21:
22:     for all $trans \in cState.OutTrans$ do
23:         if $trans.Dst.Id$ != $cState.Id$
24:                 and $trans.IsNeutral()$ then
25:             computeClusters($fState$, $trans.Dst$, $visited$,
26:                             $clusters$)
```

Algorithm 1 relies on a set of visited states to keep track of the states already traversed (lines 1 to 4). The algorithm checks if the current state is a faulty state and does not belong to any cluster yet (lines 5-6). Afterwards, we need to check if the current state is a valid state to start a cluster. To check this condition, we call another recursive algorithm called *checkCluster* (line 10), which returns a set of faulty states in a variable *tmpFS*. This function recursively goes backward to traverse the states before the current state. *tmpFS* contains the current faulty state and the set of faulty states appearing before the current state in the LTS with the same type and the same common labels than the current state. If this set contains a single faulty state ($|tmpFS|$ = 1), this is the current state and we can start building a new cluster from that state. It is also possible that the current state is involved in one or several loops. This means that there may be some faulty states with the same type and common labels when traversing backward, but then the traversal comes back to the current state. As a result, if for each incoming transition, by traversing backwards, we can come back to the current state ($tmpFS.Count(cState)$ = $|cState.IncTrans|$), this also means that the current state can be used as first state for building a new cluster. When a state satisfies these conditions, the algorithm traverses the LTS by moving forward until it finds all faulty states belonging to this new cluster (lines 17-18). From the current state, in order to keep on traversing the LTS, we recursively call the same algorithm for each transition outgoing from that state (lines 22-23). Note that since we know that we will not find any faulty state from correct and incorrect transitions, we only need to traverse neutral transitions (line 24). *OutTrans* returns all outgoing transitions from a given state. *trans.Dst* stands for the transition destination.

Algorithm 2 checks if a faulty state is the correct one to start the construction of a cluster. We recall that, in a cluster, there is one state from which all other faulty states are reachable, so we need to start the cluster computation from that state. This algorithm takes the same inputs as Algorithm 1, except that it does not need the set of clusters. Instead, it takes a set of faulty states (*FS*) as input and output. As explained for Algorithm 1, to check if the first state is involved in a loop we need to know how many times we visit the first state during the backward traversal. This is why at the first line of the algorithm, along with checking the set of visited states we also need to check if the current state is not the first state. This allows the algorithm to exit from the recursion only when the current state is not the first state of the traversal (*fState.Id != cState.Id*). Each time we find a faulty state with the same type and common labels as the first state, we put the corresponding faulty state into the set of faulty states (lines 6-8). As we can see at line 10, this algorithm traverses the LTS backward because it recursively calls itself by picking a transition in the set of transitions incoming to the current state (*IncTrans*).

Algorithm 2 CheckCluster

Inputs: First State, Current State, Set of Visited States, Set of Faulty States

Output: Set of Faulty States

```
1:  if $fState.Id$ != $cState.Id$ and $cState \in visited$ then
2:      return
3:  else
4:      $visited.Add(cState)$
5:
6:  if $cState.Type = fState.Type$
7:          and $cState.Labels \cap fState.Labels$ != {} then
8:      $FS.Add(cState)$
9:
10: for all $trans \in cState.IncTrans$ do
11:     if $trans.Dst.Id$ != $cState.Id$ then
12:         checkCluster($fState$, $trans.Dst$, $visited$, $FS$)
```

Algorithm 3 traverses forward the LTS from a given faulty state and builds the corresponding cluster (set of faulty states, common labels, involved labels). This algorithm, in addition to the inputs we have mentioned in the previous algorithms, takes a set of common labels and a set of involved labels as input. It returns as output the set of faulty states and these two sets of labels (*cLbl* and *iLbl*, respectively). A faulty state is added to the cluster if it has the same type as the first faulty state in the cluster, if it has at least one common label, and if it does not belong to another cluster (lines 5-7). If these conditions are satisfied, this state is added to the cluster and labels information updated (lines 8-10). Then, similarly to Algorithm 1, all successor states/transitions are traversed forward (lines 11-13).

Algorithm 3 TraverseCluster

Inputs: First State, Current State, Set of Visited States, Set of Faulty States, Common Labels, Involved Labels, Set of Clusters

Outputs: Set of Faulty States, Common Labels, Involved Labels

```
 1: if $cState \in visited$ then
 2:     return
 3: else
 4:     $visited.Add(cState)$
 5:     if $cState.Type = fState.Type$ then
 6:         if $cState.Labels \cap cLbl$ != {}
 7:             and $cState \notin clusters$ then
 8:             $FS.Add(cState)$
 9:             $cLbl \leftarrow cState.Labels \cap cLbl$
10:             $iLbl \leftarrow cState.Labels \cup iLbl$
11:     for all $trans \in cState.OutTrans$ do
12:         if $trans.Dst.Id$ != $cState.Id$
13:             and $trans.IsNeutral()$ then
14:             traverseCluster($fState$, $trans.Dst$, $visited$,
15:                             $FS$, $cLbl$, $iLbl$, $clusters$)
```

The computational complexity of Algorithm 1 is $O(n^2)$, where n is the number of states of the tagged LTS. In the worst case, the entire LTS is traversed and for each state, we call the other recursive algorithms, which both traverse the whole LTS as well. Performance of our algorithms could be improved and this is part of future work, but this was not our prime concern in this work.

Methodology. As far as methodology is concerned, our approach should be used when a developer is working on a buggy specification or model, and (s)he is not able to find the bug(s) in the specification. In that situation, we suggest first to use visualization techniques, which allow one to highlight the faulty part of the model. Moreover, visualization may exhibit groups of faulty states as defined previously as clusters. This is the case of the example given in Figure 2 where we can easily distinguish two clusters of faulty states. As a second step, if visualization was worthy or not, we can get more details using the computation of clusters. Note that these two results are complementary one from the other: visualization gives the whole picture with all faulty states and possibly a clear view on clusters whereas the cluster computation algorithm returns precise information about clusters which allows the developer to identify the buggy part(s) of the specification.

Before showing in the next section how to use our techniques to support the debugging tasks, we would like to introduce a few cases for which our approach is not always helpful. When the specification is entirely false, the whole tagged LTS is red and there are no faulty states. Another example is when there are several bugs in sequence. In such a case, if the first bug can be avoided, there is a cluster of faulty states only for that first bug. The solution in that situation is to resolve the first bug using the information coming with that cluster, and to apply the debugging techniques again to see whether there are other further bugs. However, if the first bug is inevitable, the tagged LTS becomes red without any cluster. Lastly, note that there is no correspondence between the number of clusters and the number of bugs. In the case of bugs in sequence, we may have a single cluster or several clusters. In contrast, we may have several clusters for a single bug. This is the case in Figure 2 for example where there are two clusters but a single bug. There are two clusters because there are two decision points that make the bug occur.

Tool support. The detection and computation of clusters is fully automated by a Java program we implemented as an extension of the CLEAR tool [5], [8]. We applied this tool to many examples (∼100) for validation purposes. The examples have been either prepared by ourselves or taken from the literature, e.g., [9]–[16]. Our solution turned out to be helpful by giving additional information (compared to the classic counterexample approach) and by thus guiding the user to the bug(s). To illustrate these experiments, we present in the next section one concrete example taken from the Internet of Things area.

V. IoT Case Study

In this section, we focus on a case study taken from the IoT area. The application describes a smart home application based on simple interacting objects (light, connected window, temperature sensor, etc.) and IFTTT [17] rules for implementing specific scenarios and orchestrating objects. More precisely, this IoT application consists of five objects: a motion sensor, a light, a connected window, a temperature sensor, and a global switch to allow one to turn off all electronic devices available in the place at night. There are several rules governing the application: if someone enters the house turn on the light, if someone leaves the house turn off the light, if temperature goes above a threshold open the window, if temperature goes down a threshold close the window, if it is late at night switch off the power supply, if it is early in the morning switch on the power supply, etc. This example is realistic yet simple enough for illustrating our approach within a couple of pages.

This IoT application is inspired from recent works on this topic [15], [18], [19]. As shown in this series of papers, this is useful from a validation perspective to transform this kind of IoT application to a formal specification language (we chose LNT [2] in this work) for verifying properties of interest (we formalise properties with MCL [7] in this work). LNT is a value-passing process algebraic specification language designed for modelling concurrent systems. LNT is equipped with a formal operational semantics, which enables to translate an LNT description into an LTS using the compilers of the

CADP toolbox [6]. The LNT operators used in this paper are: **select** (choice), **par** (interleaving or parallel composition), **;** (sequential composition), and **loop** (repetition). Behaviours are encoded using a **process** and they are parameterized by gates (communication endpoints). Behaviours communicate via rendezvous on gates.

The specification first encodes each object in LNT. As an example, a light can be turned on and off as shown in the LNT excerpt below (Listing 1). Each object is equipped with a FIFO message buffer from which it can consume to execute actions. Thus, each object is encoded into LNT as a couple *(object, buffer)*.

```
process light [lighton_read: any, lightoff_read:
    any] is
  loop
      select
        lighton_read
      []
        lightoff_read
      end select
  end loop
end process
```

Listing 1. LNT Process of a Light

Once all objects and buffers are specified in LNT, we encode all IFTTT rules into another LNT process, where a rule like ”*if someone enters the house turn on the light*” is encoded as a sequence of two actions ’movein ; lighton’ (Listing 2). All these rules can apply several times, so they are inside a loop. This loop executes until it is late and the power supply is switched off.

```
x:=1 ;
while (x==1) loop
  select
     movein ; lighton
  []
     moveout ; lightoff
  []
     warm ; openwindow
  []
     cold ; closewindow
  []
     lighton
  []
     lightoff
  []
     late ; x:=0; switchoff
  []
     ...
  end select
end loop
```

Listing 2. Excerpt of LNT Process for Rules

The main process finally describes how all these processes interact together to model the entire application. All couples *(object, buffer)* are interleaved and synchronize on all actions with the ’rules’ process, which acts as an orchestrator for the application (Listing 3). This code also shows that processes ’motion’, ’temperature’ and ’time’ do not need any buffer because they only produce events and thus do not receive anything.

```
process MAIN [ movein: any, moveout: any, ... ] is

 par  movein, moveout, lighton, lightoff, warm, ...
     in
    rules [lighton, lightoff, ...]
 ||
  par
     motion [movein, moveout]
  ||
     par lighton_read, lightoff_read in
          light [lighton_read, lightoff_read]
     ||
          bufferlight [lighton_read, lightoff_read,
              ...]
     end par
  ||
     temperature [warm, cold]
  ||
     par openwindow_read, closewindow_read in
          window [openwindow_read, closewindow_read]
     ||
          bufferwindow [openwindow_read, ...]
     end par
  ||
     time [early, late]
  ||
     par switchon_read, switchoff_read in
          switchbox [switchon_read, switchoff_read]
     ||
          bufferswitchbox [switchon_read, ...]
     end par
  end par
 end par
end process
```

Listing 3. Main Process for the IoT Application

The property used here for illustration purposes states that the power supply should not be switched off if the light is still on. This can be indeed dangerous to switch off everything in a house whereas people are still awake. This safety property is specified in MCL as follows:

[**true*** **.** ”LIGHTON” **.**
  (**not** ”LIGHTOFF”)* **.** ”SWITCHOFF” **. true*] false**

When we check this property on the LNT specification, the CADP model checker returns false with a counterexample. This is now that our proposal comes into play. A counterexample is just a trace leading to a state where the property is violated. Our approach gives more information about the bug. Let us first look at the visualization of the whole erroneous part of the corresponding state space in Figure 3. We recall that we use different colours to distinguish correct (green), incorrect (red) and neutral (black) transitions. We can see two clusters of faulty states, one yellow and one orange. The yellow states corresponds to states where there is a choice between going to a correct part of the specification (green transitions) and going to black transitions. The orange faulty states correspond to a second portion of the specification where there is a choice between black and red transitions. These two clusters appear in sequence in the whole behaviour, showing that first something happens (yellow states) and later on a final move (orange states) makes the specification fall into the erroneous portion (red transitions). This visualization is good to get a global picture of the problem, but it does not give any precise information about the source of the bug.

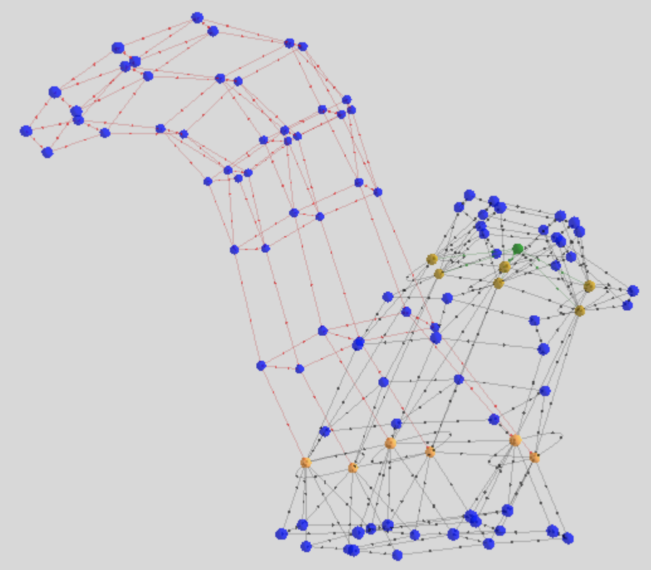

Fig. 3. Visualization of the Tagged LTS and Faulty States for the IoT Case Study

Let us now look at the two clusters with more details (Listing 4). The interest of clusters is that they give more information *wrt.* visualization, and this allows us to identify the buggy portion of the specification. The first cluster clearly points out the ’rules’ process where the set of common labels (shared by all the faulty states appearing in the cluster) correspond to all actions of the ’select’ construct used in the ’rules’ process (Listing 2). The set of involved labels show all labels used by these faulty states (shared or not). These labels refer again to the ’rules’ process (e.g., LIGHTON, LIGHTOFF, ..). We also see that the common label LATE is green, meaning that by doing this action in the first cluster, the property is not violated. When looking at the second cluster, this same label is red meaning that by doing LATE in the second cluster, the property is systematically false. Then, when going back to the tagged LTS to see what is going on between these two clusters, we see that LIGHTON systematically appears on all traces between these two states, causing the violation of the property. When looking at this part of the LNT specification (Listing 2), we can see that the encoding of the ’rules’ process was not achieved properly. Indeed, the actions LIGHTON and LIGHTOFF appear in separate portions of the choice. The action LIGHTON can execute at any time and is not mandatorily followed by any action LIGHTOFF, thus causing the violation of the property.

Once this rule is corrected or removed, we call again our tool to compute faulty states and clusters. Surprinsingly, we still have the same faulty states and the same two clusters. This means that there is another bug in the same portion of the LNT specification (’rules’ process). By looking again carefully at this code (Listing 2), we see that the two following rules execute independently one from the other: ’movein ; lighton’ and ’moveout ; lightoff’. This means that after executing the first rule, the action LIGHTON is executed. But the second one is not necessarily executed, and at night, the power supply can be switched off with the light on, thus violating the property. A solution for correcting this bug is to allow the power supply switchoff only when the light has been turned off. Once the specification has been modified, the model checker returns true for the given property.

```
Cluster Id = 1
Number of Faulty States = 6
Faulty State Type = correct and neutral transitions
Faulty States => 0, 3, 8, 21, 24, 32
Common Labels =>
    ”COLD” (black),
    ”LATE” (green),
    ”MOVEIN” (black),
    ”MOVEOUT” (black),
    ”WARM” (black)
Involved Labels =>
    ”CLOSEWINDOW_READ”,
    ”COLD”,
    ”LATE”,
    ”LIGHTOFF”,
    ”LIGHTOFF_READ”,
    ”LIGHTON”,
    ”MOVEIN”,
    ”MOVEOUT”,
    ”OPENWINDOW_READ”,
    ”WARM”

Cluster Id = 2
Number of Faulty States = 6
Faulty State Type = incorrect and neutral
    transitions
Faulty States => 4, 17, 25, 39, 53, 69
Common Labels =>
    ”COLD” (black),
    ”LATE” (red),
    ”MOVEIN” (black),
    ”MOVEOUT” (black),
    ”WARM” (black)
Involved Labels =>
    ”CLOSEWINDOW_READ”,
    ”COLD”,
    ”LATE”,
    ”LIGHTOFF”,
    ”LIGHTOFF_READ”,
    ”LIGHTON”,
    ”MOVEIN”,
    ”MOVEOUT”,
    ”OPENWINDOW_READ”,
    ”WARM”
```

Listing 4. Two Clusters

VI. Related Work

There are several research results focusing on interpreting counterexample and favouring their comprehension, see, *e.g.* [20]–[25]. In [25], sequential pattern mining is applied to execution traces for revealing unforeseen interleavings that may be a source of error, through the adoption of the well-known mining algorithm CloSpan [26]. CloSpan is also adopted in [24], where the authors apply sequential pattern mining to traces of counterexamples to reveal unforeseen interleavings that may be a source of error. However, reasoning on traces as achieved in [24], [25] induces several issues. The handling of looping behaviours is non-trivial and may result in the generation of infinite traces or of an infinite number of traces. Coverage is another problem, since a high number of traces does not guarantee to produce all the relevant behaviours for analysis purposes. As a result, we decided to work on the debugging of LTS models, which represent in a finite way all possible behaviours of the system. It is also worth noting that the approaches presented in [24], [25] are usually more scalable for large systems and do not require a complete model of the system, which may be difficult to obtain for real software artefacts.

Two other works have a specific focus on a finer analysis of counterexample traces that is more similar to our approach. In [20] the authors propose a method to interpret counterexamples traces from liveness properties by dividing them into fated and free segments. Fated segments represents inevitability w.r.t. the failure, pointing out progress towards the bug, while free segments highlight the possibility to avoid the bug. The proposed approach classifies states in a state-based model in different layers (which represent distances from the bug) and produces a counterexample annotated with segments by exploring the model. Both our work and [20] aim at building an explanation from the counterexample. However, our method focuses on locating branching behaviours that affect the property satisfaction whereas their approach produces an enhanced counterexample where inevitable events (w.r.t. the bug) are highlighted. Moreover our approach has a specific focus on safety properties, while they focus on liveness properties.

In [22] the authors propose automated methods for the analysis of variations of a counterexample, in order to identify portions of the source code crucial to distinguishing failing and successful runs. These variations can be distinguished between executions that produce an error (negatives) and executions that do not produce it (positives). By relying on a notion of control location, their method tries to make sure that such variations are for one bug to avoid multi-bug confusions. The authors then propose various methods to extract common features and differences between the two sets in order to provide feedbacks to the user. Three different extraction methods are proposed: transition analysis, invariant analysis and transformation of positives into negatives. Similarly to our work, the work in [22] also wants to better explain the counterexample with a focus on safety properties. However, while our approach has a global view on the whole LTS and focuses on the study of faulty states to understand how they affect the property satisfaction, their method relies on the analysis of a single counterexample and its variations, making sure that negative variations are from the same bug.

A part of our method relies on bug visualization techniques. The closest work to ours is a tool for visualizing the structure of very large state spaces developed by Groote and Van Ham [27], [28]. This approach relies on a clustering method to generate a simplified representation of the state space, and can be useful for better understanding the overall structure of the model. To do this, this method builds a 3D representation in the form of a tree, which constitutes the backbone of the whole graph. The rest of the tree structure is built by clustering sets of states. The use of clustering techniques provides an high scalability to the approach, since individual states and transitions are not displayed but are grouped in sets. This tool is particularly useful for better understanding the overall structure of the model, but this work does not necessarily target debugging, which is our main objective here. Indeed, in contrast, we do not have only as input a specification and its corresponding model, but also a temporal property. This property allows us to distinguish correct and incorrect behaviours in our model, and our visualization techniques focus on this question in order to identify a particular structure that would help the developer to understand and identify the source of the bug.

VII. Concluding Remarks

In this paper, we have presented a solution for simplifying the comprehension of bugs when analysing a specification using model checking techniques. The core idea is to focus on specific decision points in the specification, that make the corresponding model go to a correct or an incorrect portion of the model. These faulty choices are particularly helpful during the debugging task because they highlight parts of the specification, that require attention and correction. In most cases, when a property is violated the model exhibits many faulty states and it is not always obvious how to use this information to precisely identify the source of the bug. In this paper, we propose an algorithm to group similar faulty states in clusters. Each cluster gathers information about a part of the specification where a given property of the model is violated. As a result, the developer does not have to study all faulty states, but can focus on clusters of such states, which is simpler from a debugging perspective. The computation of clusters is fully automated by an extension of the CLEAR tool that we implemented. Our approach and tool support was validated on several examples and we particularly used one taken from the IoT area for illustration purposes in this paper.

As far as future work is concerned, we would like first to refine the definition of cluster. This would allow us to have better information regarding the source of the bug, and thus develop a more precise method to guide the user to find and correct the bug(s). Performance of our algorithms was not of prime focus in this work, but we plan to improve them in order to make our approach efficient on large specifications. We also plan to exploit this notion of clusters for quantifying the faulty part of the program, and to detect whether there is one bug in the model whose occurrences are repeated in several places or if there are several different bugs (and how many). These quantitative techniques for evaluating the number of bugs will contribute to measure the effort for debugging the program (pay-as-you-go verification). Finally, we plan to refine our techniques in order to not only detect the source of the bug(s) but also to propose a solution for automatic repair of the bug(s).

References

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

[2] 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)," 2018, INRIA/VASY and INRIA/CONVECS, 153 pages.

[3] 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.

[4] ——, "Counterexample Simplification for Liveness Property Violation," in *Proc. of SEFM'18*, ser. LNCS, vol. 10886. Springer, 2018, pp. 173–188.

[5] ——, "Debugging of behavioural models with CLEAR," in *Proc. of TACAS'19*, ser. LNCS, vol. 11427. Springer, 2019, pp. 386–392.

[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.

[7] 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, pp. 148–164.

[8] G. Barbon, V. Leroy, G. Salaün, and E. Yah, "Visual Debugging of Behavioural Models," in *Proc. of ICSE'19*, J. M. Atlee, T. Bultan, and J. Whittle, Eds. IEEE / ACM, 2019, pp. 107–110.

[9] R. Mateescu, P. Poizat, and G. Salaün, "Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques," *IEEE TSE*, vol. 38, no. 4, pp. 755–777, 2012.

[10] G. Salaün, T. Bultan, and N. Roohi, "Realizability of Choreographies Using Process Algebra Encodings," *IEEE Transactions on Services Computing*, vol. 5, no. 3, pp. 290–304, 2012.

[11] R. Mateescu and W. Serwe, "Model Checking and Performance Evaluation with CADP Illustrated on Shared-memory Mutual Exclusion Protocols," *Sci. Comput. Program.*, vol. 78, no. 7, pp. 843–861, 2013.

[12] F. Aarts, H. Kuppens, J. Tretmans, F. W. Vaandrager, and S. Verwer, "Improving Active Mealy Machine Learning for Protocol Conformance Testing," *Mach. Learn.*, vol. 96, no. 1-2, pp. 189–224, 2014.

[13] F. Durán and G. Salaün, "Verifying Timed BPMN Processes Using Maude," in *Proc. of COORDINATION'17*, ser. LNCS, vol. 10319. Springer, 2017, pp. 219–236.

[14] X. Etchevers, G. Salaün, F. Boyer, T. Coupaye, and N. D. Palma, "Reliable Self-deployment of Distributed Cloud Applications," *Softw. Pract. Exp.*, vol. 47, no. 1, pp. 3–20, 2017.

[15] A. Krishna, M. L. Pallec, R. Mateescu, L. Noirie, and G. Salaün, "Rigorous Design and Deployment of IoT Applications," in *Proc. of FormaliSE'19*, 2019, pp. 21–30.

[16] U. Ozeer, G. Salaün, L. Letondeur, F. Ottogalli, and J. Vincent, "Verification of a Failure Management Protocol for Stateful IoT Applications," in *Proc. of FMICS'20*, ser. LNCS, vol. 12327. Springer, 2020, pp. 272–287.

[17] IFTTT, "If This Then That," 2020, https://ifttt.com/.

[18] 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*. IEEE / ACM, 2019, pp. 19–22.

[19] A. Krishna, M. L. Pallec, A. Martinez, R. Mateescu, and G. Salaün, "MOZART: Design and Deployment of Advanced IoT Applications," in *Proc. of WWW'20*. ACM, 2020.

[20] H. Jin, K. Ravi, and F. Somenzi, "Fate and Free Will in Error Traces," in *Proc. of TACAS'02*, ser. LNCS, vol. 2280. Springer, 2002, pp. 445–459.

[21] T. Ball, M. Naik, and S. K. Rajamani, "From symptom to cause: localizing errors in counterexample traces," in *Proc. of POPL'03*. ACM, 2003, pp. 97–105.

[22] A. Groce and W. Visser, "What went wrong: Explaining counterexamples," in *Proc. of SPIN'03*, ser. LNCS, vol. 2648. Springer, 2003, pp. 121–135.

[23] K. Ravi and F. Somenzi, "Minimal assignments for bounded model checking," in *Proc. of TACAS'04*, ser. LNCS, vol. 2988. Springer, 2004, pp. 31–45.

[24] S. Leue and M. T. Befrouei, "Mining Sequential Patterns to Explain Concurrent Counterexamples," in *Proc. of SPIN'13*, ser. LNCS, vol. 7976. Springer, 2013, pp. 264–281.

[25] M. T. Befrouei, C. Wang, and G. Weissenbacher, "Abstraction and Mining of Traces to Explain Concurrency Bugs," in *Proc. of RV'14*, ser. LNCS, vol. 8734. Springer, 2014, pp. 162–177.

[26] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining Closed Sequential Patterns in Large Datasets," in *Proc. of SDM'03*. SIAM, 2003, pp. 166–177.

[27] J. F. Groote and F. van Ham, "Large State Space Visualization," in *Proc. of TACAS'03*, ser. LNCS, vol. 2619. Springer, 2003, pp. 585–590.

[28] ——, "Interactive visualization of large state spaces," *STTT*, vol. 8, no. 1, pp. 77–91, 2006.