Generating Multiple Diverse Counterexamples for an EFSM

Alma L. Juarez Dominguez and Nancy A. Day

Cheriton School of Computer Science
University of Waterloo, Canada
{aljuarez,nday}@uwaterloo.ca

Technical Report CS-2013-06

September 6, 2013

Abstract

Model checking is a powerful technique for debugging a system description because it generates a counterexample showing a path of the system that fails a property. Instead of the traditional cycle of find bug – fix bug – re-run model checker, often we would like to study multiple bugs before fixing the model to help isolate the cause of the error and to improve the user's experience by avoiding iterations of this cycle. However, the set of all counterexamples is often too large to generate or comprehend, and several counterexamples may be caused by the same error. We present a novel method of using a model checker to generate a set of diverse counterexamples to an invariant of an extended finite state machine (EFSM) model. The goal is that each diverse counterexample reveals distinct information about a bug in the model. We use the modelling abstractions of control states and transitions of an EFSM to define whether two counterexamples are equivalent or not. Our method reduces the set of counterexamples on-the-fly and can be used with any LTL model checker.

1 Introduction

Model checking is a powerful technique for finding bugs (errors, inconsistencies and contradictions) in a model because it searches exhaustively all behaviours of the system and generates a counterexample showing a path of the system that fails a property [9]. We are interested in verifying an invariant, which is a property that must be true at all times during the execution of the model. The traditional use of model checking is the cycle consisting of find bug - fix bug - re-run model checker, until no more counterexamples are found. However, it can be useful to see multiple counterexamples prior to fixing the model to

help isolate the cause of the error [4, 10, 14, 7], as one counterexample by itself may not contain enough information to fix correctly the bug [10]. Additionally, seeing multiple counterexamples at once rather than the user iterating this cycle can improve the user's experience of model checking and ultimately, the amount of time it takes to create a correct model, in a similar way to having a compiler return multiple errors in one pass [4, 10, 7].

If we use a tool that is able to generate multiple counterexamples (*e.g.,* SPIN [17]), we find that the paths produced are often slight data variations of each other. For example, if a counterexample follows a path that concludes at a state where $x = 5$, seeing another counterexample following a path through the same states but resulting in $x = 6$ could just be a data variation from the designer's perspective and might not show a distinct bug. The complete set of counterexample paths is often too large to sift through to find those that illustrate distinct bugs in the model.

Our goal is to create a method that produces for the user a set of diverse counterexamples, which each reveal information about distinct bugs in the model. There is no single definition of what is a distinct bug in a model, and the error is not always in the step immediately before an invariant fails. For the large family of languages based on extended finite state machine (EFSM) [8] models (*e.g.,* Statecharts [15], Requirements State Machine Language (RSML) [25], MATLAB's Stateflow [1], Specification and Description Language (SDL) [2])), we have found that the modelling abstractions of explicit control states and transitions provide a useful way of differentiating diverse counterexamples and removing the non-essential details of the data variations of a path that fails an invariant. These abstractions match the internal conceptual models that people use to understand and represent complex systems [26].

Detecting multiple counterexamples requires either (1) a change to the model checking engine [17, 10, 19, 7] or (2) the creation of an automatic method that iteratively changes either (a) the model [4] or (b) the property, until a sufficient set of counterexamples is generated. Some model checkers, such as SPIN [17], generate all counterexamples by having the model checking algorithm continue to search the state space after finding a counterexample until no more counterexamples exist. Yet, most of these counterexamples are slight data variations of each other. Moreover, it can take a long time to generate all counterexamples and the result is often too large to comprehend, providing little help in isolating the actual bugs.

Ball *et al.* [4] follow the approach of modifying the model: when a counterexample is generated, their method identifies a transition in the counterexample that does not appear in the set of correct paths computed by their method so far. Then, the identified transition is removed from the model and their method continues, searching for another counterexample. By changing the model, they eliminate the possibility of finding a different path that includes the removed transition and, therefore, the set of counterexamples may not include a representative of all distinct paths that fail the invariant. If the bug is actually caused by a combination of factors along the path, an incorrect resolution may be chosen, thus, leaving the model still susceptible to failures.

We present a novel solution to the problem of generating multiple counterexamples based on the third possible approach: automatically modifying the property. In contrast to modifying the model checking engine or the model, our method can work with any linear temporal logic (LTL) [27] model checker (explicit or symbolic) and it produces representatives of the complete set of counterexamples to an invariant because the model is never changed during the process.

Our first contribution is four definitions of equivalence among counterexamples. Each definition (or level) groups the complete set of counterexamples into equivalence classes based on their properties in the EFSM, making paths that are just data variations of each other equivalent. Our definitions of equivalence were chosen carefully after experimentation with many examples, with the goal that one representative from each equivalence class forms a useful set of distinct counterexamples for the user to review and find all the bugs in the model. For example, one level groups together all counterexamples that follow the same sequence of transitions in the EFSM, but if a modeller wants less detail, another level groups together all counterexamples that end at the same control state. We describe how each level can be useful to isolate errors at different times during the analysis process.

Our second contribution is to show how these definitions can be used to generate one counterexample from each equivalence class on-the-fly during model checking iterations to produce a set of distinct counterexamples. It is fairly straightforward to create a property that disallows a previously output counterexample (disjunct the invariant with a property describing the counterexample path). However, an approach that produces all counterexamples and groups them afterwards (*e.g.,* [10]), is extremely time-consuming. Instead, we add an LTL property that describes all paths in the equivalence class, thus disallowing the generation of any counterexamples that are equivalent to those that we have already seen. This "on-the-fly" nature of our method dramatically reduces the time it takes to produce a useful set of counterexamples as only one counterexample per equivalence class is generated.

We demonstrate the use of our method on several case studies of hierarchical EFSMs, including an air conditioning system, an audio player, as well as, four automotive feature design models created in Stateflow [21], which are representative of industrial models. These EFSM case studies show both the usefulness of our definitions of what are equivalent counterexamples and the reduction we achieve in the amount of information provided to the user. In future work, we plan to tackle the problem of generalizing our definitions and methods for concurrent state machines.

2 Motivating Example

Consider the EFSM model of a simple air conditioning system (AC) in Figure 1. The input variable `e` can take on the values `enter` and `exit`, while the input variable `t` (temperature) and controlled variable `pt` (previous temperature) range

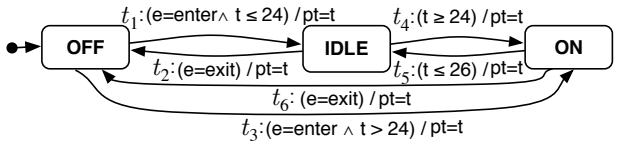


Figure 1: EFSM of an air conditioning system (AC)

over the values 0..40. Checking the invariant that model AC is in the control state **ON** if and only if the previous temperature is greater than 24, the model checker first generates the counterexample

$$\begin{aligned} c_1\colon \langle(&\textbf{OFF},\ -,\ \mathsf{e{=}enter},\ \mathsf{t{=}22},\ \mathsf{pt{=}0}),\\ &(\textbf{IDLE},\ t_1,\ \mathsf{e{=}enter},\ \mathsf{t{=}24},\ \mathsf{pt{=}22}),\\ &(\textbf{ON},\ \ t_4,\ \mathsf{e{=}exit},\ \ \mathsf{t{=}26},\ \mathsf{pt{=}24})\rangle \end{aligned}$$

where the transition taken is part of the counterexample information. The bug is that the model reaches **ON** when the previous temperature is 24. After examining the model, we observe that there is an error on t_4: the condition ($\mathsf{t}{\geq}24$) should be ($\mathsf{t}{>}24$). However, if we consider all counterexamples prior to correcting this one, the model checker could generate another counterexample for the same property, such as,

$$\begin{aligned} c_2\colon \langle(&\textbf{OFF},\ -,\ \mathsf{e{=}enter},\ \mathsf{t{=}24},\ \mathsf{pt{=}0}),\\ &(\textbf{IDLE},\ t_1,\ \mathsf{e{=}enter},\ \mathsf{t{=}24},\ \mathsf{pt{=}24}),\\ &(\textbf{ON},\ \ t_4,\ \mathsf{e{=}enter},\ \mathsf{t{=}22},\ \mathsf{pt{=}24})\rangle \end{aligned}$$

and yet another counterexample is

$$\begin{aligned} c_3\colon \langle(&\textbf{OFF},\ -,\ \mathsf{e{=}enter},\ \mathsf{t{=}20},\ \mathsf{pt{=}0}),\\ &(\textbf{IDLE},\ t_1,\ \mathsf{e{=}enter},\ \mathsf{t{=}24},\ \mathsf{pt{=}20}),\\ &(\textbf{ON},\ \ t_4,\ \mathsf{e{=}enter},\ \mathsf{t{=}25},\ \mathsf{pt{=}24})\rangle. \end{aligned}$$

From the modeller's perspective, counterexamples c_1, c_2 and c_3 are all instances of the EFSM path $\langle$**OFF**-t_1-**IDLE**-t_4-**ON**$\rangle$ and identify the same bug: an incorrect transition guard on t_4. These variations would not be eliminated by cone of influence reduction [24] because the verification of the property depends on the values of both t and pt. However, the data variations found in these counterexamples do not help to find another bug in the model. We would much rather find a path that identifies another bug in the model, such as counterexample

$$\begin{aligned} c_4\colon \langle(&\textbf{OFF},\ -,\ \mathsf{e{=}enter},\ \mathsf{t{=}23},\ \mathsf{pt{=}0}),\\ &(\textbf{IDLE},\ t_1,\ \mathsf{e{=}enter},\ \mathsf{t{=}25},\ \mathsf{pt{=}23}),\\ &(\textbf{ON},\ \ t_4,\ \mathsf{e{=}enter},\ \mathsf{t{=}26},\ \mathsf{pt{=}25}),\\ &(\textbf{IDLE},\ t_5,\ \mathsf{e{=}enter},\ \mathsf{t{=}26},\ \mathsf{pt{=}26})\rangle, \end{aligned}$$

which is an instance of the EFSM path $\langle$**OFF**-t_1-**IDLE**-t_4-**ON**-t_5-**IDLE**$\rangle$. This counterexample, where the model reaches **IDLE** when the previous temperature is 26, illustrates a different bug in the model: the condition ($\mathsf{t}{\leq}26$) on transition t_5 should be ($\mathsf{t}{\leq}24$). The approach by Ball *et al.* would not produce counterexample c_4 because transition t_4 would have been eliminated from the model after identifying either counterexample c_1, c_2 or c_3. Section 4 will show concretely the reduction that our method achieves for a variant of model AC in Figure 1.

3 Background: EFSMs

An **extended finite state machine** (EFSM) is a model with a finite set of control states and labelled transitions, extended with variables [8]. These variables can be used in transition triggers or as part of the actions of the transitions. We explain our method using a generic flat EFSM as a common basis amongst the many EFSM-based modelling languages. Our method generalizes to models with hierarchical control states easily in Section 6. Graphically, control states are represented as nodes with transitions as edges, and the initial control states are designated with edges that have no source control state, as shown in Figure 2.

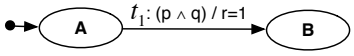


Figure 2: Example of simple EFSM

The **syntax** of an EFSM consists of a tuple

$$\langle CS,\ IV,\ OV,\ T,\ InitCS,\ FinalCS,\ InitV \rangle$$

where

- CS is a finite set of control states.
- IV is a set of input variables, OV is a set of output (controlled) variables, and the sets IV and OV are disjoint. We use $V = IV \cup OV$. All variables are of finite type.
- T is a finite set of **progressing** transitions. Each transition has a name, a source and destination control state, and a label of the form $(c)/a$, where (c) is an optional condition on variables in V called a guard, and a is an optional set of assignments to variables in OV called actions.
- $InitCS$ is a set of initial control states ($InitCS \subseteq CS$).
- $FinalCS$ is a set of control states that are not the source of any progressing transition ($FinalCS \subseteq CS$).
- $InitV$ is a set of sets of initial values for variables. Each set contains one pair for each variable, and each pair assigns a value to the variable from the variable's type.

Events, which are often present in EFSMs, can be modelled as Boolean variables with values that do not persist.

The **semantics** of an EFSM is a set of infinite paths. A **path** of an EFSM is a sequence of configurations. Each **configuration** consists of a control state, a set of assignments from variables to values in each variable's type, and the name of the transition taken to reach the control state in the configuration. A transition moves the model from one configuration to the next along a path. A transition is taken when the model is in a configuration consisting of the source control state of the transition and the values of the variables satisfy the guard of the transition. For example, for the model in Figure 2, transition t_1 is taken when the model is in source control state **A** and the input variables `p` and `q` both are *true*. When a transition is taken, the model moves to a configuration containing the destination state and the effects of executing the assignments

to the variables. Any output variable that is not assigned a value keeps its previous value. Input variables can change arbitrarily between configurations. Continuing the example, the execution of transition t_1 causes the model to move to control state **B**, changing the value of the controlled variable r to 1.

Semantically, every control state implicitly has a single self-looping transition, which is taken when no guard on any other transition exiting the state is satisfied. We call these transitions **non-progressing**, and use the transition name ***tn*** for them. There are no actions associated with non-progressing transitions so the values of the output variables do not change, but input changes may occur. A self-looping transition in T with no guard or actions implicitly has the guard *true*, unlike a non-progressing transition, whose guard is implicitly the conjunction of the negation of the guards of any other transitions exiting the state. Non-progressing transitions ensure that at every configuration there is a next configuration.

We can describe the meaning of an EFSM in a Kripke structure (KS) for model checking. In a KS, the control states of an EFSM are typically modelled as a variable with the control state names as values. The other EFSM variables are modelled as KS variables of appropriate types. A KS representing an EFSM will have many KS states with the same control state and many KS transitions related to one transition in the EFSM. The control states of the EFSM are abstractions created by the modeller to group together a set of past behaviours that have the same set of possible future behaviours. The KS representation of an EFSM will include KS transitions for non-progressing EFSM transitions (tn).

4 Levels of Counterexample Equivalence Classes

In this section, we define our levels of equivalence classes for the counterexamples of an EFSM. The intuition for our grouping into equivalence classes is that counterexamples that follow the same path in the EFSM with different data values should be considered equivalent because they do not add to the modeller's understanding of the error. For each equivalence class, the user sees an actual counterexample including the sequence of data values that lead to the error as a representative of the class.

A counterexample is an infinite path of the model that contains a configuration that fails the invariant, and its representation returned by the model checker is a finite sequence of configurations (part of which may represent a cycle at the end of the path). We call the set of all counterexamples ***CE***. The first step in our definitions is to create a representation of *CE*, which we call ***FIPaths*** (failed invariant paths)[1], which takes care of obvious reductions in how we view a counterexample. Some model checkers automatically do some of

[1]Because the set of configurations is finite (given that the number of states and transitions is finite plus the domains from which the variables can take values are also finite), and the elements of *FIPaths* are finite sequences of non-repeating configurations (paths contain no configuration loops), the set *FIPaths* must be finite.

these reductions before presenting the counterexample to the user, but for generality, we start from a counterexample as an infinite path with no reductions. The set *FIPaths* is defined as:

$$\begin{aligned} FIPaths = \{ q \mid \exists c \in CE \bullet \\ q = reduce\_vals(reduce\_init\_config \\ (reduce\_config\_loops(trunc(progress(c))))\} \end{aligned}$$

where:

- *progress*(c): Removes all non-progressing transitions from c to avoid stuttering, except for a constant loop of non-progressing transitions that might appear at the end of the path if the model reaches a final control state[2]. The inputs in the last configuration of a sequence of non-progressing transitions are the only ones that might cause the error and these are copied back to the configuration with the progressing transition just before the sequence begins.
- *trunc*(c): Creates the subpath of c that ends in the first configuration that fails the invariant as this finite prefix can identify the cause of violation of the invariant.
- *reduce_config_loops(c)*: Removes copies of a configuration loop from c. A **configuration loop** is one that reaches the same configuration more than once in c. These loops are unnecessary because the path without these loops has all the steps that cause the error.
- *reduce_init_config*(c): Removes from c a loop that starts at an initial configuration and reaches another initial configuration. An **initial configuration** is one that contains a control state in *InitCS* and assignments to variables in *InitV*. This loop contains excessive information because the bug can be reached without traversing this loop.
- *reduce_vals(q):* Removes from c: (1) the value of the transition name in the first configuration and (2) the input variables in the final configuration of the path. These values do not contribute to the error because (1) the transition name in the first configuration is irrelevant and (2) the inputs to the model in the final configuration are used to calculate the next configuration (not the current configuration).

Two elements of *CE* that map to the same element of *FIPaths* fairly obviously should be considered equivalent. To illustrate the use of the definition of *FIPaths*, consider the following counterexample for model AC in Figure 1:

$\langle$(OFF, tn, e=enter, t=15, pt=0),
(IDLE, t_1, e=enter, t=18, pt=15),
(IDLE, tn, e=enter, t=22, pt=18),
(IDLE, tn, e=enter, t=24, pt=22),
(**ON**, t_4, **e=exit**, **t=26**, **pt=24**),
(OFF, t_6, e=enter, t=23, pt=26), $\cdots\rangle$

with (a) a loop of non-progressing transitions indicated by a bar at the left, eliminated by *progress*, (b) the configuration that fails the invariant in bold, with

[2]It might appear easier to swap functions *trunc* and *progress* in the definition to avoid dealing with loops at the end. However, as it will be seen in Section 5, this order is more convenient for implementation.

Level 1	Level 2	Level 3	Level 4
$[\langle t_1, t_4 \rangle] - 18$	$[t_4] - 30$	[OFF,ON] – 30	[ON] – 30
$[\langle t_3, t_5, t_4 \rangle] - 12$			
$[\langle t_3, t_5 \rangle] - 9$	$[t_5] - 15$	[OFF,IDLE] – 15	[IDLE] – 15
$[\langle t_1, t_4, t_5 \rangle] - 6$			

Table 1: Levels of counterexample equivalence classes for reduced variable value model AC in Figure 1

any configuration after that one truncated by *trunc*, and (c) the application of *reduce_vals* to the resulting subpath, therefore generating the following element of *FIPaths* (where *reduce_config_loops* and *reduce_init_config* have no effect on this path):

$$\langle(\mathsf{OFF}, \quad \mathsf{e{=}enter},\ \mathsf{t}{=}15,\ \mathsf{pt}{=}0),\\ (\mathsf{IDLE},\ t_1,\ \mathsf{e{=}enter},\ \mathsf{t}{=}24,\ \mathsf{pt}{=}15),\\ (\mathbf{ON},\ \ t_4, \quad\quad\quad\quad\quad\quad\ \mathbf{pt{=}24})\rangle.$$

Next, we present, in order from the most detailed to the least detailed, our levels of equivalence classes, which define distinct counterexamples. We chose these levels of equivalence classes based on what is deemed as relevant and useful in our case studies and in the literature. Table 1 shows the equivalence classes created by each of our levels for the example model AC in Figure 1. To show the reduction achieved by our method, we wanted to show how many elements of *FIPaths* are contained within each equivalence class, however, even for a small example such as AC, we found the set *FIPaths* is too large to generate easily because of the ranges of the variable values. Thus, we could only generate the elements of *FIPaths* for a variant of model AC with t (temperature) and pt (previous temperature) ranging over 0..2 (instead of 0..40). There are 45 elements of *FIPaths* for this variant of model AC. The number beside an equivalence class in Table 1 indicates the number of elements of *FIPaths* in each class for this reduced AC model.

We use the following notation to describe our levels:

- *FICS:* The set of control states that are in a reachable configuration in which the invariant fails.
- *FIT*: The set of transitions that are in a reachable configuration in which the invariant fails. These are the transitions that lead to a state in *FICS*.
- *fst_cs(p):* The control state of the first configuration in path p.
- *lst_cs(p):* The control state of the last configuration in path p.
- *lst_tr(p):* The last transition taken in path p.
- *tr_seq(p):* The finite sequence of transitions in path p.
- *all_but_last*(p): The finite sequence of configurations in path p except for the last one.
- *reduceEFSM(p):* Removes EFSM loops from path p. An **EFSM loop** is a sequence of configurations within p that begins and ends in the same control state.

We use the notation $[x]$ for the equivalence class of x, which consists of the set of equivalent elements of *FIPaths* in the class x. x may be a control state, a path, or a transition, *etc.*

Level 1: Distinct Paths - Our definition groups paths with multiple iterations of an EFSM loop together because from the user's perspective iterating the EFSM loop does not add information about the cause of the error. For example, the path of the model AC in Figure 1, with **ON** in *FICS*

$$\langle \textbf{OFF}\text{-}t_1\text{-}\textbf{IDLE}\text{-}t_2\text{-}\textbf{OFF}\text{-}t_1\text{-}\textbf{IDLE}\text{-}t_4\text{-}\textbf{ON}\rangle$$

identifies the same bug, from the modeller's perspective, as the EFSM path

$$\langle \textbf{OFF}\text{-}t_1\text{-}\textbf{IDLE}\text{-}t_2\text{-}\textbf{OFF}\text{-}t_1\text{-}\textbf{IDLE}\text{-}t_2\text{-}\textbf{OFF}\text{-}t_1\text{-}\textbf{IDLE}\text{-}t_4\text{-}\textbf{ON}\rangle$$

and also the same information regarding the error as the path without EFSM loops

$$\langle \textbf{OFF}\text{-}t_1\text{-}\textbf{IDLE}\text{-}t_4\text{-}\textbf{ON}\rangle.$$

The modeller would rather see another path that shows a different kind of error! Thus, we consider all paths with iterations of an EFSM loop to be equivalent to one without such loops, using *reduceEFSM* in the definition of Level 1.

However, EFSM loops that end in the configuration that fails the invariant cannot be eliminated without losing too much information because the actions on the transition leading to the failed configuration are the immediate source of the failure. Therefore, we make Level 1 differentiate paths by their last transition. The immediate cause of the failure (although not necessarily the bug in the model) can then be found by analyzing the guard or the actions of this transition. The small example of Figure 3 shows why counterexamples should be differentiated by their last transition, otherwise, path $\langle t_1, t_2, t_2\rangle$ and path $\langle t_1, t_3, t_3\rangle$ would both be reduced to $\langle t_1\rangle$ by *reduceEFSM* with respect to C and considered equivalent. Instead, Level 1 generates the equivalence classes $[\langle t_1, t_2\rangle]$ and $[\langle t_1, t_3\rangle]$.

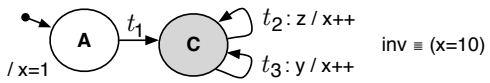


Figure 3: Small example for Level 1

Level 1 considers as equivalent all the counterexamples that end in the same transition and that have the same sequence of transitions after removing EFSM loops in the rest of the path[3]. The counterexample presented to the user shows one instantiation of the EFSM path, thus, even though EFSM loops are removed to define the equivalence class, if an EFSM loop is needed to fail the invariant (because of data changes, such as a loop for a counter as in Figure 3), this loop will be present in the counterexample the user sees.

Level 1: $\forall p \in FIPaths \bullet [p] =$
$\{q \in FIPaths \mid (lst\_tr(q) = lst\_tr(p))$

[3]For Levels 1-2, the counterexample path must be of length at least one.

$$\wedge\ (tr\_seq(reduceEFSM(all\_but\_last(p))) = tr\_seq(reduceEFSM(all\_but\_last(q))))\}$$

There are four equivalence classes at Level 1 for the EFSM in Figure 1. For example, the data variations of the path $\langle t_1, t_4, t_5, t_4\rangle$ are all part of the equivalence class $[\langle t_1, t_4\rangle]$ after removing the EFSM loop with respect to control state **IDLE**. Four distinct counterexamples can represent these four equivalence classes. Two of them were shown in Section 2 (c_1 and c_4). The third counterexample is an instance of the EFSM path $\langle$**OFF**-t_3-**ON**-t_5-**IDLE**$\rangle$. The fourth one is an instance of the EFSM path $\langle$**OFF**-t_3-**ON**-t_5-**IDLE**-t_4-**ON**$\rangle$.

Level 2: Distinct Last Transitions - In Level 2, all counterexamples that have the same last transition are considered equivalent. The rationale for this level is that it distinguishes bugs where the error is in the transition immediately before the invariant fails.

Level 2: $\forall t \in FIT \bullet [t] = \{p \in FIPaths \mid lst\_tr(p) = t\}$

There are two equivalence classes at Level 2 for the EFSM in Figure 1. For example, the path $\langle t_1, t_4, t_5, t_4\rangle$ is part of the equivalence class $[t_4]$.

Groce and Visser consider a variation of this level to be appropriate for debugging Java programs, as they believe that all counterexamples that pass through the same control location to reach the same error (and thus, having the same suffix) match the programmer's understanding of equivalent counterexamples [14].

Level 3: Distinct Initial and Final States - In Level 3, all counterexamples that have the same initial control state and final control state are considered equivalent.

Level 3: $\forall i \in InitCS, \forall s \in FICS \bullet [i, s] = \{p \in FIPaths \mid fst\_cs(p) = i \wedge lst\_cs(p) = s\}$

In this definition, an equivalence class is empty if an initial control state is not the first state on a path that leads to a control state in *FICS*. There are two non-empty equivalence classes at Level 3 for the EFSM in Figure 1. For example, the path $\langle t_1, t_4, t_5, t_4\rangle$ is part of the equivalence class [**OFF**,**ON**].

Level 3 can be a useful preliminary check to examine conditions on the initial control states and variable values that lead to an error in order to find bugs in the specification of possible initial values.

Level 4: Distinct Final States - In Level 4, all counterexamples that lead to the same final control state are considered equivalent.

Level 4: $\forall s \in FICS \bullet [s] = \{p \in FIPaths \mid lst\_cs(p) = s\}$

There are two equivalence classes at Level 4 for the EFSM in Figure 1. For example, the path $\langle t_1, t_4, t_5, t_4\rangle$ is part of the equivalence class [**ON**].

Chechik and Gurfinkel say that a level like this one is important to quickly find errors in initial states when the very first state fails the invariant [7]. For other cases, this level might be useful in choosing a resolution: *e.g.,* a sink/error state could be added to the model from these states.

5 On-the-fly LTL Counterexample Grouping

We have created a method and tool, both called called ***Alfie***[4], to produce a set of diverse counterexamples to an invariant of an EFSM based on our definitions in Section 4 of equivalent counterexamples. *Alfie* automatically iterates the model checking process, modifying the property in each iteration to rule out on-the-fly the generation of any counterexample equivalent to those already produced. At the end of the process, the user will receive one counterexample per equivalence class. *Alfie* uses the Cadence SMV model checker [28] as the model checking component, but any LTL model checker could be used. In our method, by ruling out all equivalent counterexamples on-the-fly, there is no need to generate all counterexamples, which substantially reduces the number of iterations of the model checker compared to related work [10], where all counterexamples are generated and then grouped into equivalence classes.

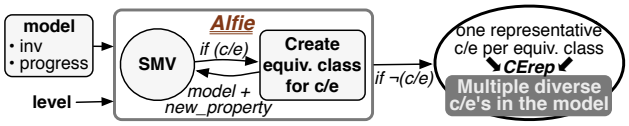


Figure 4: On-the-fly grouping level process

Our on-the-fly grouping method and tool *Alfie* is illustrated in Figure 4. Macro inv specifies the invariant (a property that must be true at every reachable configuration of the model). Our tool *Alfie* iteratively (1) asks SMV to generate a counterexample, (2) deduces the equivalence class of the counterexample for the desired level, (3) represents this equivalence class as an LTL expression, (4) creates a new property that is the disjunction of this LTL expression with the invariant and the LTL expressions representing previously generated counterexamples, and (5) repeats the process by re-running the model checker on the same model with the new property. By disjuncting an LTL expression of the equivalence class with the property, *Alfie* disallows the generation of any more counterexamples in that equivalence class. *Alfie* produces as output one representative counterexample per equivalence class, and runs iteratively until no more equivalence classes of counterexamples are found. We call this set of representative counterexamples ***CErep***. Our method does not rely on the order in which the counterexamples are generated (such as generation of the shortest one first). Our process terminates because the set of *FIPaths* is finite and a different element is generated at each iteration of our method.

Our method uses the following LTL operators:

- Globally ($\mathbf{G}\ \psi$): ψ must hold at every configuration on the path.
- Next ($\mathbf{X}\ \psi$): ψ must hold at the next configuration on the path.
- Strong Until ($\psi\ \mathbf{U}\ \phi$): ϕ must hold at the current or a future configuration on the path, and ψ has to hold until that position. From there on, ψ does not need to hold.

[4]The name *Alfie* is derived from All Failed Invariants.

- $\neg$, $\wedge$, $\vee$ and $\rightarrow$: negation, conjunction, disjunction and logical implication respectively.

To generate the property representing the equivalence class of a counterexample, c, our method first calculates the element of *FIPaths*, q, associated with c. By incorporating part of the definition of *FIPaths* into the LTL property to be checked, we can limit the model checking exploration. Therefore, for every level of equivalence classes, our method begins by running the model checker with the property

$$\textbf{prop}\text{: } \mathrm{G}(\mathsf{progress}) \rightarrow \mathrm{G}(\mathsf{inv})$$

to generate the first counterexample, c. The macro progress implements the definition of the function *progress* in Section 4, ensuring that the counterexample c only contains progressing transitions. The macro progress is defined as $(\neg(\mathrm{X}(tn)) \vee \mathit{finalCS})$ meaning the last transition taken (from the second step of the path onwards) does not have the label tn or the path has reached a final control state, from which there are only non-progressing transitions.

From the counterexample, c, returned, our method applies to c the rest of the definition of *FIPaths*, described in Section 4, thus generating the element q. First, the function *trunc* creates a subpath of c ending in the first configuration that fails the invariant, followed by the application of the functions *reduce_config_loops*, *reduce_init_config* and *reduce_vals* to the subpath returned by *trunc*. For the element q of *FIPaths*, *Alfie* creates an LTL expression $L_{[q]}$ according to the desired level that is added to the invariant as a disjunction, creating the property to check next,

$$\textbf{prop\_L}\text{: } \mathrm{G}(\mathsf{progress}) \rightarrow ((\mathrm{G}(\mathsf{inv})) \vee L_{[q_1]} \vee ... \vee L_{[q_i]}).$$

In each iteration of our process, for the i-th counterexample, c_i, an LTL expression $L_{[q_i]}$ is added as a new disjunction to the consequent.

Because of the temporal operators used in $L_{[q]}$ for some levels, the property **prop_L** may no longer be just an invariant. However, our definition of *FIPaths* works for both counterexamples that are a finite prefix of an infinite path (those produced by safety properties) and counterexamples that include a loop at the end (those produced by liveness properties). Figure 5 illustrates the relationship between counterexamples (*CE*), *FIPaths*, and LTL expressions representing equivalence classes.

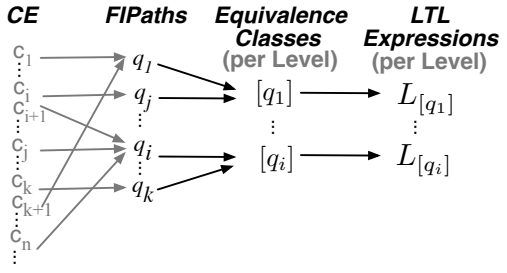


Figure 5: Relationship between *CE*, *FIPaths* and LTL properties representing equivalence classes

Next, we explain in order from the least complex to the most complex (reverse order of Section 4) the LTL expression that represents an equivalence class of counterexamples according to the desired level. The LTL expression checked after one iteration of our process for each level is shown in Table 2 for a counterexample from model AC in Figure 1. For simplicity in the presentation of properties, we use the control state name to mean that the configuration includes this control state, and we use the transition label to mean that the configuration includes this transition as the last transition taken.

Level	LTL property
4	(G (progress)) $\rightarrow$ ((G (inv)) $\vee$ (inv U ($\neg$inv $\wedge$ ON)))
3	(G (progress)) $\rightarrow$ ((G (inv)) $\vee$ (OFF $\wedge$ (inv U ($\neg$inv $\wedge$ ON))))
2	(G (progress)) $\rightarrow$ ((G (inv)) $\vee$ (inv U ($\neg$inv $\wedge$ t_4)))
1	(G (progress)) $\rightarrow$ ((G (inv)) $\vee$ (OFF $\wedge$ (inv U (t_1 $\wedge$ (inv U ($\neg$inv $\wedge$ t_4))))))

Table 2: LTL property per level for counterexample $\langle$OFF-t_1-IDLE-t_4-ON$\rangle$ from the model AC of Figure 1

The number of iterations of our process is equal to the number of equivalence classes. The loop invariant that holds after each iteration is:

Theorem 1 $\quad \forall p \in CE \bullet (p \models \textbf{\textit{prop\_L}} \Leftrightarrow \exists c \in CErep \bullet p \in [c])$

meaning that **prop_L** exactly covers the equivalence classes for the counterexamples in *CErep* seen so far in the process. The justification of *Theorem* 1 is given for Level 4 below, while the justification for the rest of the levels can be found in [20]. The process terminates when no more counterexamples are produced.

Level 4: Distinct Final States - For a path $q \in FIPaths$, in Level 4 *Alfie* adds to the property a disjunction with an LTL expression, $L_{[q]}$, that uses the value of the control state in the last configuration of q (which is an element of *FICS*), generating the following property with each $L_{[q]}$ highlighted:

prop_L4: (G (progress)) $\rightarrow$ ((G (inv)) $\vee$ (inv U ($\neg$inv $\wedge$ *lst_cs*(q)) .

At each iteration, **prop_L4** forces the model checker to find a counterexample in which the first control state to fail inv is different from any $lst\_cs(q_i)$ in the counterexample of *CERep*. The process concludes when all control states in *FICS* have been discovered.

Expression $L_{[q_i]}$ in **prop_L4** includes $\neg$inv as a conjunction with $lst\_cs(q_i)$ because another counterexample may have a prefix with $lst\_cs(q_i)$ in it, but the invariant does not fail in that instance of $lst\_cs(q_i)$, and this counterexample is in a distinct equivalence class, as illustrated by Figure 6.

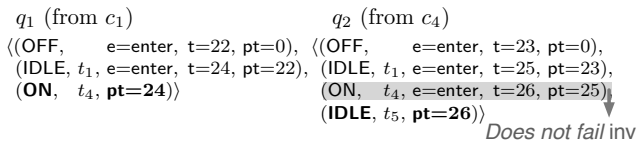


Figure 6: Counterexample c_4 for AC of Figure 1 missed by **prop_L4** if ¬inv is excluded

The justification of *Theorem* 1 with respect to property **prop_L4** for Level 4 is as follows:

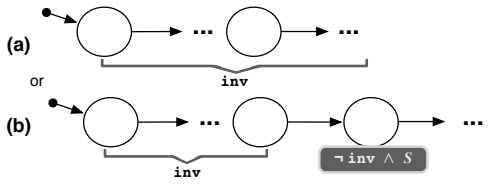
(⇒) $\forall p \in CE \bullet (p \models \textbf{prop\_L4} \;\Rightarrow\; \exists c \in CErep \bullet p \in [c])$

For Level 4, property **prop_L4** has the form

$((\text{G (progress)}) \rightarrow ((\text{G (inv)}) \vee (\text{inv U } (\neg\text{inv} \wedge S_1))$
$\cdots$
$\vee (\text{inv U } (\neg\text{inv} \wedge S_k)))$

where $\{S_1,\cdots,S_k\} =$
$\{lst\_cs(FIPaths(c_1)),\cdots,lst\_cs(FIPaths(c_k)) \quad | \; c_1,..,c_k \in CErep\}$.

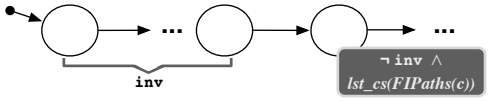
A path p that satisfies **prop_L** must be of one of two forms:



For case **(a)**, p is not in *CE*, so *Theorem* 1 does not apply to this case. For case **(b)**, path p has as the first control state that fails the invariant an element S, returned by $lst\_cs(FIPaths(c))$, for some $c \in CErep$. Therefore, $p \in [c]$ by ***Definition 4***. □

(⇐) $\forall p \in CE \bullet ((\exists c \in CErep \bullet p \in [c]) \Rightarrow p \models \textbf{prop\_L4})$

Let path p be a member of an equivalence class of c in *CErep*. When c was generated by the model checker, property **prop_L4** included the LTL expression (inv U $(\neg\text{inv} \wedge lst\_cs(FIPaths(c)))$), disjuncted with the invariant. If $p \in [c]$, p must have the form:



Therefore, path p satisfies **prop_L4**. □

Level 3: Distinct Initial and Final States - For a path $q \in FIPaths$, in Level 3 *Alfie* adds to the property a disjunction with an LTL expression, $L_{[q]}$, describing the initial control state of q (which must be an element of $InitCS$), and the last control state in q (which must be an element of $FICS$), generating the following property with $L_{[q]}$ highlighted:

prop_L3: (G (progress)) → ((G (inv)) ∨
$(fst\_cs(q) \wedge (\mathsf{inv}\ \mathsf{U}\ (\neg\mathsf{inv} \wedge lst\_cs(q))))$).

prop_L3 forces the model checker to search for a counterexample that either starts with the same initial control state, but ends at a different control state that fails the invariant, or starts with a different initial control state and ends at a control state where the invariant fails. For brevity of the LTL expression over multiple paths, all final control states are grouped with the same initial control state together in a disjunction with the invariant. The process concludes when all combinations of initial control states in $InitCS$ that reach a final control state in $FICS$ have been discovered.

Level 2: Distinct Last Transitions - For a path $q \in FIPaths$, in Level 2 *Alfie* adds to the property a disjunction with an LTL expression $L_{[q]}$ that uses the value of the last transition taken in q (which must lead to a control state in $FICS$ and be part of FIT), generating the following property with $L_{[q]}$ highlighted:

prop_L2: (G (progress)) → ((G (inv)) ∨
$(\mathsf{inv}\ \mathsf{U}\ (\neg\mathsf{inv} \wedge lst\_trans(q)))$).

prop_L2 forces the model checker to find a counterexample in which the transition that leads to the first control state that fails the invariant is different from the one described by $lst\_trans(q)$. The process concludes when all transitions in the set FIT are discovered.

Level 1: Distinct Paths - For a path $q \in FIPaths$, in Level 1 *Alfie* adds to the property a disjunction with an LTL expression $L_{[q]}$ that accepts any path with the same sequence of transitions as q, and all EFSM looping variations of the path before the last transition of q. A looping variant is any path that reaches that same control state two or more times in the path as illustrated in Figure 7. By including the looping variations, the model checker will not report them as distinct counterexamples. The EFSM loops in the looping variations of a path must not contain states that fail the invariant.

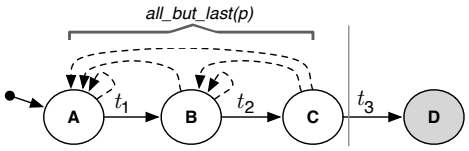


Figure 7: Looping variations of path $\langle t_1, t_2, t_3 \rangle$

Level 1 was the most difficult level to express correctly in LTL. For illustration, consider $q_1 = \langle t_1, t_4 \rangle$, which is the element of $FIPaths$ derived from counterexample c_1 for model AC. The most natural LTL expression to describe

the sequence of transitions in q_1, including looping variants, is (where OFF is the initial control state in the path):

$$\mathsf{OFF} \wedge (\mathrm{F}\ (t_1 \wedge (\mathrm{F}\ t_4))) \qquad (a)$$

However, expression (a) allows looping variants of path q_1 that contain configurations that fail the invariant to be part of the same equivalence class, which is incorrect. Through the use of the Until operator, our LTL expression only includes paths with EFSM loops whose states all satisfy the invariant. The correct LTL expression for q_1 is:

$$\mathsf{OFF} \wedge (\mathsf{inv}\ \mathrm{U}\ (t_1 \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge t_4)))) \qquad (b)$$

and the LTL expression for c_4 (shown in Section 2) is:

$$\mathsf{OFF} \wedge (\mathsf{inv}\ \mathrm{U}\ (t_1 \wedge (\mathsf{inv}\ \mathrm{U}\ (t_4 \wedge (\mathsf{inv}\ \mathrm{U}\ (\neg\mathsf{inv} \wedge t_5)))))) \qquad (c)$$

This LTL expression includes the condition indicating that the invariant does not hold when taking the transition to the configuration that fails the invariant, which is the last transition of the path (t_4 in q_1). Thus, paths that pass through t_4, but do not fail the invariant at t_4, do not satisfy the property (and are not in its equivalence class).

Level	**Equiv. Classes**	**BDD Nodes**	**Time**
4	2	5264	2.32s
3	2	5282	2.34s
2	2	5327	2.32s
1	4	16012	2.79s

Table 3: Statistics for the analysis of model AC in Figure 1

Table 3 shows the number of equivalence classes per level, the maximum BDD size for all iterations of the model checker, and the total time for all iterations of the analysis of AC in Figure 1 (full range of data values). The equivalence classes in Table 3 match those of Table 1 because the full data range model and the reduced data range model both contain the same errors. The model checking verification runs in this paper were performed on a 2.8GHz AMD Opteron CPU with 32GB of RAM.

6 Case Studies

In this section, we show how our method produces diverse counterexamples to help find distinct bugs in an EFSM model of a more complex air conditioning system, an audio player system, and in four automotive feature models. The choice of level depends on how much the user wants to differentiate amongst counterexamples. We seeded the bugs in our case studies. Because all our case studies have only one initial state, our discussions refer only to Levels 1, 2 and 4. Level 4 (distinct final control states) and Level 3 (distinct initial and final control states) both generate the same equivalence classes. Level 1 provides the greatest confidence of finding a set of counterexamples that reveals information

about all distinct bugs, however, it can provide more detail than is needed. Our case studies illustrate this spectrum.

6.1 Air Conditioning System for a Two-story Unit

Consider the air conditioning system (AC2) for a two-story unit shown in Figure 8. The input variable e can take on the values up (a person enters the upper floor), down (a person enters the lower floor) and exit (a person leaves the building), while the input variable t (temperature) and controlled variable pt (previous temperature) range over the values 0..40.

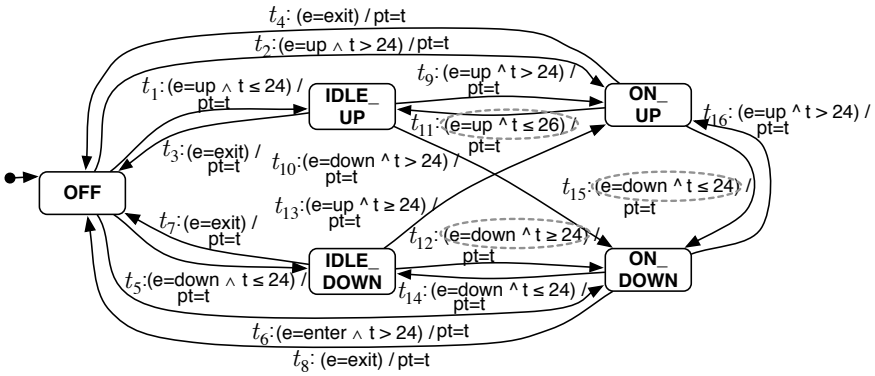


Figure 8: EFSM of a more complex air conditioning system (AC2)

We check a similar invariant to the one for AC in Section 2, stating that model AC2 is either in **ON_UP** or in **ON_DOWN** if and only if the the previous temperature is greater than 24. The results for analysis of all levels are summarized in Table 4, showing the number of equivalence classes, the maximum BDD size over all iterations of the model checker and the total time taken to complete all iterations. In Figure 8, the three actual bugs in AC2 are circled.

Level	Equiv. Classes	BDD Nodes	Time
4	2	12069	2.44s
3	2	12069	2.45s
2	3	11497	2.56s
1	20	6466591	54m32s

Table 4: Statistics for the analysis of model AC2 in Figure 8

For **Level 4**, considering the distinct final control states found, we can learn:

- [(IDLE_UP)]: This counterexample shows that AC2 reaches idle, but with the wrong conditions. The error occurs because the condition on t_{11} checks (t $\leq$ 26) instead of (t $\leq$ 24).
- [(ON_DOWN)]: This counterexample shows that AC2 is on when it is meant to be idle. The error occurs because transition t_{15} checks (t $\leq$ 24) when it should be (t $>$ 24).

However, this level of detail has not isolated all the actual errors in the model.

Level 2 (distinct last transitions) reports 3 distinct counterexamples:

- $[t_{11}]$: Same error as found for equivalence class [(IDLE_UP)] above.
- $[t_{15}]$: Same error as found for equivalence class [(ON_DOWN)] above.
- $[t_{12}]$: The counterexample representing this equivalence class shows that AC2 is on by taking transition t_{12}. The error occurs because transition t_{12} checks (t $\geq$ 24), but it should be (t $>$ 24). This is a distinct error from those found by Levels 3 or 4.

Level 1 generates 20 equivalence classes: each diverse counterexample is a distinct path to a configuration where the invariant fails because the model contains a number of loops. A close examination of the paths lets us identify that all 20 paths are caused by the three transitions (the last one in each path) found by Level 2. In this case, Level 1 did not help to isolate any distinct bugs, and potentially was not worth the time it took. However, Level 1 does show that taking one of the transitions reported by Level 2 does not always lead directly to an error. For example, there are two equivalence classes that end with t_{12}:

$[\langle t_6,t_{14},t_{12}\rangle]$ and $[\langle t_5,t_{12}\rangle]$

indicating that the error is related to t_{12}. But there is an equivalence class where t_{12} does not lead directly to an error:

$[\langle t_5,t_{12},t_{16},t_{15}\rangle]$.

By analyzing these three distinct counterexamples, we obtain more information to modify easily and correctly the condition in t_{12}.

6.2 Audio Player System

Consider an audio player system (AP)[5] shown in Figure 9. When turned on, the system is able to play one of three different audio sources: radio, tape or CD. When a tape or a CD is inserted in the system, that source is selected to play, while when there is no tape or CD, only the radio is available. The input variable e can take on the values next and back (to switch between four radio stations, spool the tape forward or backward, or select the previous or next track of a CD) as well as play (play the tape or the current CD track). The Boolean input variables power, tape_insert, tape_eject, cd_insert and cd_eject correspond to their descriptions, while input variable n (number of tracks) ranges over the values 1..20.

The EFSM model for AP is a hierarchical non-concurrent state machine, represented in the SMV model as one state name variable per hierarchy level. The constraints on control states are expressed over the values of control states

[5]Model is partially based on EFSM by Seifert [30].

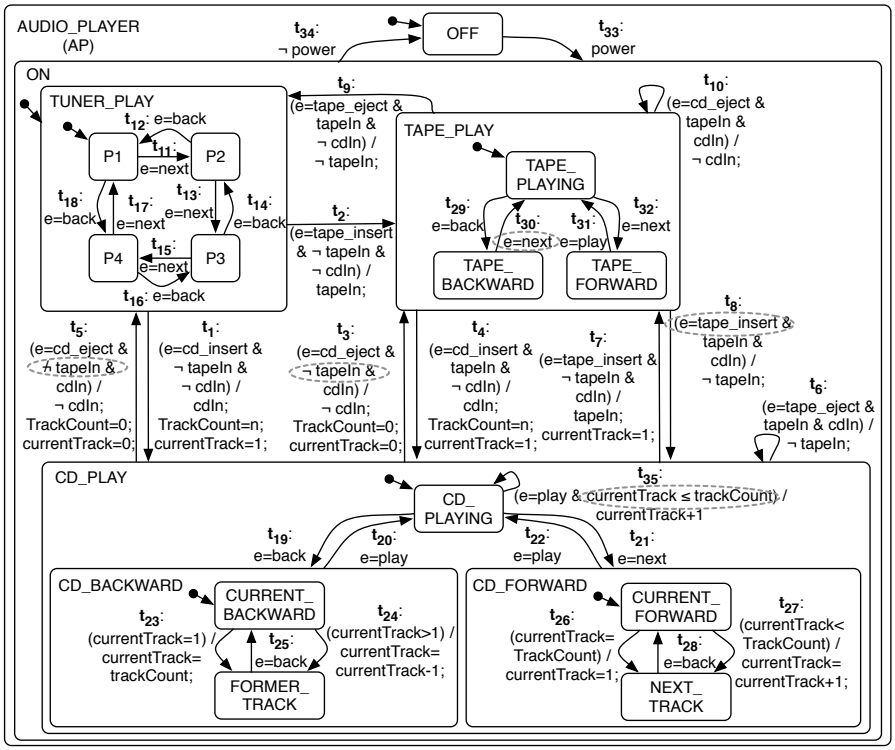


Figure 9: EFSM of an audio player system (AP)

at all levels in the hierarchy. No other changes in the definitions of our levels of counterexample equivalence classes are required. We checked a single large property that the AP model follows precisely its functional requirements. The results for analysis of all levels are summarized in Table 5, showing the number of equivalence classes, the maximum BDD size over all iterations of the model checker and the total time taken to complete all iterations. Figure 9 shows the five actual bugs in model AP circled.

Considering the information provided by **Level 4**, we can learn:

- [(sAP=ON,sON=TUNER_PLAY)]: This counterexample shows that AP reaches the tuner state, but with the wrong conditions. The error occurs because the condition on t_5 checks (¬tapeIn) instead of (tapeIn).
- [(sAP=ON,sON=TAPE_PLAY)]: This counterexample shows that AP is in the tape state when it is meant to be in tuner. The error occurs because transition t_3 checks (tapeIn) when it should be (¬tapeIn).
- [(sAP=ON,sON=CD_PLAY)]: This counterexample shows that AP reaches the CD state with the wrong conditions. The error occurs because transition t_8 checks (e=tape_insert) when it should be (e=tape_eject).

Level	Equiv. Classes	BDD Nodes	Time
4	3	87498	11.77s
3	3	92338	11.83s
2	5	84742	16.33s
1	10	1450210	4m40s

Table 5: Statistics for the analysis of model AP in Figure 9

Once again, **Level 4** has not isolated all the actual errors in the model. **Level 2** reports 5 distinct counterexamples:

- $[t_5]$: Same error as found for equivalence class [(sAP=ON,sON=TUNER_PLAY)] above.
- $[t_3]$: Same error as found for equivalence class [(sAP=ON,sON=TAPE_PLAY)] above.
- $[t_8]$: Same error as found for equivalence class [(sAP=ON,sON=CD_PLAY)] above.
- $[t_{30}]$: The counterexample representing this equivalence class shows that AP is in the state that plays the tape by taking transition t_{30}. The error occurs because transition t_{30} checks (e=next), but it should be (e=play).
- $[t_{35}]$: The counterexample representing this equivalence class shows that AP is in the state that plays the CD by taking transition t_{35}. The error occurs because transition t_{35} checks (currentTrack $\leq$ trackCount), but the current track is out of range.

Level 1 generates 10 equivalence classes: each diverse counterexample is a distinct path to a configuration where the invariant fails because the model contains a number of loops. However, all 10 paths are caused by the five transitions (the last one in each path) found by Level 2 to have errors.

6.3 Active Safety Automotive Features

Next, we considered the analysis of a set of non-proprietary automotive features designed in MATLAB's Stateflow [1] that we created previously [21]: *Collision Avoidance (CA)*, *Emergency Vehicle Avoidance (EVA)*, *Parking Space Centering (PSC)*, and *Reversing Assistance (RA)*. These features are known as "Active Safety Systems" because they use sensors, cameras, and radar to help the driver control the vehicle. CA helps to prevent or mitigate collisions when driving forward. EVA pulls the car over when an emergency vehicle needs the road to be cleared. PSC assists during perpendicular parking. RA helps prevent or mitigate collisions when reversing. Our automotive features are representative in type and complexity of models that we have seen developed in industrial practice[6], but do not include concurrency or failure modes (*e.g.,* fail-safe states for degraded modes of operation).

[6]Observed during Alma Juarez's visits to General Motors (GM) Research and Development as part of the requirements of her NSERC Industrial Postgraduate Scholarship.

	Reachable State Space	# Trans.	# Vars.	Max. Vars. range	Basic Control States
CA	8.24241e+07	26	25	100	9
EVA	1.64848e+08	19	34	100	8
PSC	2.74266e+10	18	34	100	12
RA	6.18181e+07	18	24	100	8

Table 6: Sizes of automotive feature models CA, EVA, PSC and RA

Level	CA			EVA		
	# Equiv. Classes	BDD Nodes	Time	# Equiv. Classes	BDD Nodes	Time
4	2	452147	4.4s	2	36053	3.4s
3	2	452147	4.4s	2	36053	3.4s
2	2	453186	4.6s	2	39991	3.5s
1	2	465399	5.1s	2	17857	3.8s
Level	**PSC**			**RA**		
	# Equiv. Classes	BDD Nodes	Time	# Equiv. Classes	BDD Nodes	Time
4	2	24220	3.4s	2	452178	5.0s
3	2	24220	3.4s	2	452178	5.0s
2	2	39329	3.5s	2	453022	4.5s
1	2	31507	4.6s	2	465797	4.9s

Table 7: Case study results per level of equivalence classes for automotive feature models

In previous work [22] [23], we created a translator from a subset of Stateflow to SMV and used it to translate these feature design models to SMV. Stateflow is used extensively in the automotive and avionics industries. In Stateflow, features are hierarchical state machines[7]. Thus, our translation creates one state name variable per hierarchy level of the EFSM in the SMV model, and the constraints on control states are expressed over the values of control states at all levels in the hierarchy. Also, automotive features only have one initial state, because Stateflow does not allow multiple initial states. Table 6 contains information on the size of the translated models in SMV.

For CA, EVA, PSC and RA, we checked the property that a feature remains disengaged when intended, as specified in each feature's functional requirements. The results of our analysis are summarized in Table 7, showing the number of equivalence classes, the maximum BDD nodes used over all iterations of the model checker and the time taken to complete all iterations of the analysis.

[7]The features analyzed in this work do not use Stateflow concurrency, although it is supported by our translator.

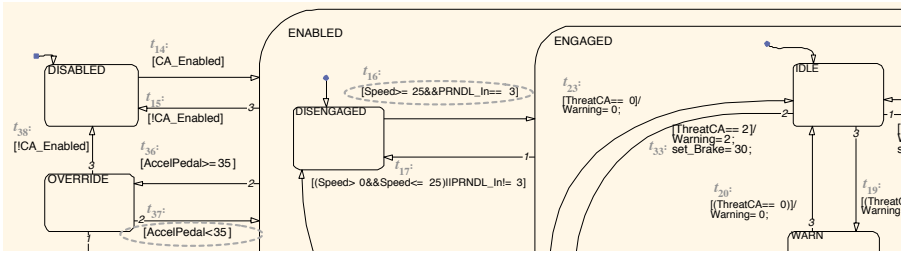


Figure 10: Excerpt of STATEFLOW design feature CA showing the two bugs uncovered by **Alfie**

Figure 10 circles the two actual bugs in CA. *Alfie* generated two distinct counterexamples at all levels exactly matching these two bugs. Consider the information provided by Level 1 (distinct paths):

- $[\langle t_{14}, t_{16} \rangle]$: This counterexample shows the sequence of transitions leading CA to become engaged when it should be disengaged. The error occurs because the condition on t_{16} is (Speed ≥ 25), although the condition should be (Speed > 25).
- $[\langle t_{14}, t_{36}, t_{37} \rangle]$: This counterexample shows the sequence of transitions that lead CA to be disengaged when it should not be. The error occurs because transition t_{36} can be taken regardless of the enabledness of CA. A correction is to have t_{37} check if variable CA_Enabled is *true*, which is a necessary condition for CA to be disengaged.

While all the levels provided the same number of distinct counterexamples in this case study, Level 1 gives the highest confidence that we have isolated the distinct bugs in the model. Level 1 is the only level that differentiates errors that are not in the last transition. For example, for the bug isolated by the second counterexample above, an alternative correction might be to change t_{36}. If there were multiple transitions besides t_{36} entering state OVERRIDE, there could be multiple distinct bugs to isolate, which would be captured by Level 1.

Overall, our case studies show our method provides a useful representation of the complete set of counterexamples without having to generate all counterexamples, a process that may not be possible, or if possible, would take a long time. To illustrate the reduction that we accomplish with our method, we explain how many data variant paths are represented by the equivalence class p= $[\langle t_{14}, t_{16} \rangle]$ reported by Level 1 for model CA, and illustrated in Figure 11. The input variables used as guards in transitions restrict the values that these variables can take. However, the input variables in CA have very large ranges, *e.g.,* Speed, ranging from 0 to 100. Therefore, the number of ways the input values can vary is extremely large, as shown in the numbers on top of the circles in Figure 11. These numbers denote the data variations allowed at each step of path p. The total number of data variant counterexample paths for the equivalence class p is 7.822×10^{21}, which does not include looping variants of

path p. Even though some model checkers are able to detect irrelevant variables using techniques such as cone of influence reduction [24], having variables such as Speed as in this example, would still produce a great number of counterexamples.

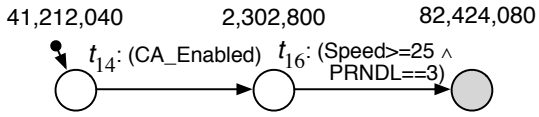


Figure 11: Data variant paths in equiv. class $[\langle t_{14}, t_{16} \rangle]$

7 Discussion

Our definitions of equivalent counterexamples may not match exactly the set of distinct changes that must be made to fix a model. However, as illustrated in our case studies, we believe our definitions offer a good balance between reduction and detail in creating a useful set of distinct counterexamples for the modeller to study. We anticipate that a user is likely to either focus on Level 1 to gain as much information as possible about the complete set of bugs or incrementally progress through the levels to see if a higher level produces any additional useful information. Our approach would work well with an approach that animates counterexamples or extracts a sub-model that contains errors.

In our analysis, we chose to concentrate on the first configuration where the invariant fails in a path. It might be possible to extend our approach to distinguish counterexamples that have a second configuration where the invariant fails, however, it is not clear that it is worth considering these second failures since the system has already failed the invariant.

So far, because of the reduction achieved by our approach we have not experienced scalability problems in our case studies. However, a danger with our approach is that when there are many equivalence classes, the growing size of the property becomes an inhibiting factor in model checking because LTL model checking scalability depends on the size of the property as well as the size of the model [29]. We are investigating promising methods for partitioning the property to deal with larger size models and alternative model checkers.

8 Related Work

To the best of our knowledge, our approach is the first to generate and summarize the set of all counterexamples on-the-fly by modifying the property. Some approaches to generating multiple counterexamples use a modified version of a model checking algorithm to generate all counterexamples. In addition to SPIN [17], which can generate all counterexamples by continuing the state space search after the property fails, Copty *et al.* create a model checking engine that

generates a BDD representing all counterexamples of a given length, and in a post-processing step, annotate these counterexamples to help diagnose and fix a reported failure [10].

Many approaches do not necessarily generate all counterexamples while attempting to isolate the cause of an error. Jin *et al.* created a model checking algorithm variation that creates annotated counterexamples with events describing fate (inevitability towards the error) or free will (attempt to avoid the error) [19]. Groce and Visser describe an algorithm to find traces that are data variations of a counterexample for Java programs, then process this set of traces to find differences between counterexamples and traces with no error [14]. Sharygina and Peled use a testing tool to generate traces that are related to a counterexample for a software program, where the generated traces or *neighbourhood* of a counterexample might help understand the cause of the error [31]. The neighbourhood of a counterexample may contain traces with no error as well as other counterexamples, but no automatic analysis is done to group or classify counterexamples (if more than one exist). Chechik and Gurfinkel use a modified model checker that generates multiple counterexamples to a property and then in post-processing create a proof-like tree to summarize the data variations from the counterexamples generated [7]. Beer *et al.* describe an algorithm to detect a set of causes for the first failure to a property by evaluating sub-formulas of the property on the given counterexample trace [5].

Ball *et al.* modify the model by removing the transition in the counterexample that does not appear in any correct trace so far and then looking for another counterexample [4]. Their method uses explicit state model checking and makes the assumption that the cause of a failure is a single transition. Therefore, all counterexamples that include this transition are in the same equivalence class, even ones that result in a different error, and the equivalence class of a counterexample is not precisely defined because it depends on the order in which the traces are explored. By changing the model, they eliminate the possibility of finding a different counterexample that includes the removed transition.

Our work bares some resemblance to the use of model checking to generate test cases of a model that satisfy certain coverage criteria (*e.g.,* [3], [16], [11], [18]). However, in these approaches, one witness (test case) is found for each property and then a new property is created to generate another witness until the coverage criteria is satisfied. Some testing approaches use a structural coverage criteria for EFSM-based models. For example, Geist, Hartman *et al.* developed the tool GOTCHA for state and transition coverage [6], [12], while Gargantini and Heitmeyer construct properties from an SCR specification for structural coverage based on guards of a transition [13].

Compared to all these approaches, by focusing on EFSM models, we have been able to create an automatic method that produces a set of diverse counterexamples on-the-fly to help the modeller isolate the cause of bugs efficiently.

9 Conclusion

In this paper, we defined a series of levels of equivalence classes that each create a set of distinct counterexamples to an invariant. This reduced set is easier to generate and comprehend than the whole set of counterexamples. We have shown how to represent these equivalence classes on-the-fly as LTL properties to be used by a model checker so that all counterexamples are never generated. Our approach can be used with any LTL model checker independent of the order in which it produces counterexamples. We believe that our definition of equivalent counterexamples is a very useful one for analysis of EFSM-based languages because of the fundamental role that control states and transitions play in how people represent systems in these languages. We demonstrated the reduction produced by our equivalence classes of counterexamples in the verification for several case studies including four automotive feature design models. The weakness of our approach is that the model checker will repeat work again as it analyzes the model in each iteration of our cycle, however the number of iterations is often quite small because each of our LTL properties represents a set of counterexamples.

In this paper, our methodology has been explained for a single EFSM, but we are working on generalizing it for multiple concurrent EFSMs without flattening the model. We plan to use our technique for the detection of feature interactions between automotive features. In this application, it is quite important to have a comprehensive view of all feature interactions prior to recommending changes to the features models or creating a manager that resolves interactions at run-time since a feature interaction is not an error in the model. We anticipate that these examples will exercise our ability to partition the LTL properties.

Acknowledgments

This work was partially funded by General Motors (GM) Canada, Critical Systems Labs Inc. and the NSERC Automotive Partnership Canada grant.

References

[1] MathWorks Documentation.

[2] *ITU-T Recommendation Z.100: Specification and Description Language (SDL)*. ITU, 2007.

[3] P. Ammann, W. Ding, and D. Xu. Using a model checker to test safety properties. In *Int. Conference on Engineering of Complex Computer Systems*, pages 212–221, 2001.

[4] T. Ball, M. Naik, and S. K. Rajamani. From symptom to cause: Localizing errors in counterexample traces. *SIGPLAN Notices*, 38(1):97–105, 2003.

[5] I. Beer, S. Ben-David, H. Chockler, A. Orni, and R. Trefler. Explaining Counterexamples Using Causality. In *Proc. Int'l Conf. on Computer Aided Verification*, pages 94–108. Springer-Verlag, 2009.

[6] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets. A study in coverage-driven test generation. In *Conference on Design Automation*, pages 970–975. ACM, 1999.

[7] M. Chechik and A. Gurfinkel. A framework for counterexample generation and exploration. *Int. J. Softw. Tools Technol. Transf.*, 9(5):429–445, 2007.

[8] K. T. Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Int. Design Automation Conference*, pages 86–91. ACM, 1993.

[9] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 2000.

[10] F. Copty, A. Irron, O. Weissberg, N. P. Kropp, and G. Kamhi. Efficient debugging in a formal verification environment. In *Conference on Correct Hardware Design and Verification Methods*, pages 275–292, 2001.

[11] A. Engels, L. M. G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 384–398. Springer-Verlag, 1997.

[12] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *Int. Symposium on Software Testing and Analysis*, pages 134–143. ACM, 2002.

[13] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. *SIGSOFT Software Engineering Notes*, 24(6):146–162, 1999.

[14] A. Groce and W. Visser. What went wrong: Explaining counterexamples. In *SPIN Workshop on Model Checking of Software*, pages 121–135. Springer, 2003.

[15] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computing Programming*, 8(3):231–274, June 1987.

[16] M. P. Heimdahl, S. Rayadurgam, W. Visser, G. Devaraj, and J. Gao. Auto-generating test sequences using model checkers: A case study. In *Int. Worshop on Formal Approaches to Testing of Software*. Springer, 2003.

[17] G. J. Holzmann. *The Spin Model Checker*. Addison-Wesley, 2003.

[18] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *Int. Conference on Software Engineering*, pages 232–243, 2003.

[19] H. Jin, K. Ravi, and F. Somenzi. Fate and free will in error traces. In *Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 445–459. Springer-Verlag, 2002.

[20] A. L. Juarez Dominguez. *Detection of Feature Interactions for Automatic Active Safety Features*. PhD Thesis, University of Waterloo, 2012.

[21] A. L. Juarez Dominguez, N. A. Day, and R. T. Fanson. A preliminary report on tool support and methodology for feature interaction detection. Technical Report CS-2007-44, University of Waterloo, 2007.

[22] A. L. Juarez Dominguez, N. A. Day, and R. T. Fanson. Translating Models of Automotive Features in MATLAB's Stateflow to SMV to Detect Feature Interactions. In *Int. Systems Safety Conference*. System Safety Society, 2008.

[23] A. L. Juarez Dominguez, N. A. Day, and J. J. Joyce. Modelling Feature Interactions in the Automotive Domain. In *Int. Workshop on Modeling in Software Engineering*, pages 45–50. ACM Press, 2008.

[24] R. P. Kurshan. *Computer-aided verification of coordinating processes: the automata-theoretic approach*. Princeton University Press, 1994.

[25] N. Leveson, M. Heimdahl, H. Hildreth, and J. Reese. Requirements specification for process-control systems. *IEEE Transactions on Software Engineering*, 20(9):684–707, 1994.

[26] N. G. Leveson. *Safeware: System Safety and Computers*. Addison Wesley, 2001.

[27] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, 1st edition, 1992.

[28] K. L. McMillan. *Symbolic model checking*. Kluwer Academic, 1993.

[29] D. Peled. *Software Reliability Methods*. Springer, 2001.

[30] D. Seifert. Test Case Generation from UML State Machines. Technical Report inria-00268864, DEDALE - LORIA, 2008.

[31] N. Sharygina and D. Peled. A combined testing and verification approach for software reliability. In *Int. Symposium of Formal Methods Europe*, pages 611–628. Springer-Verlag, 2001.