# On-the-Fly Counterexample Abstraction for Model Checking Invariants

Alma L. Juarez-Dominguez, and Nancy A. Day

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

Technical Report CS-2010-11

July 5, 2010

**Abstract**

Verification of invariants using model checking can find errors, inconsistencies, and contradictions in a model. In some applications, it is beneficial to see at once all paths of the model in which the invariant fails. Thus, instead of the traditional cycle of find bug – fix bug – re-run model checker, we would like to produce *all* counterexamples prior to fixing the bug. However, the set of all counterexamples is often too large to generate or comprehend. We define a series of abstractions that each represent the complete set of counterexamples in a reduced set of equivalence classes. Our abstractions are based on the control states and transitions of an extended finite state machine (EFSM) model. We show how our abstractions can be represented as LTL properties and, therefore, reduce the set of counterexamples on-the-fly during model checking.

## 1 Introduction

Model checking is a powerful technique for finding errors, inconsistencies and contradictions in a model because it searches exhaustively all behaviours of the system [7]. If a model does not satisfy a property describing a desired behaviour, a model checker produces a counterexample, which is a path of the model's behaviour that fails the property. 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. For some applications it is advantageous to find *all* paths that fail the property before fixing the model because the correction may depend on several factors that can only be recognized by looking at all counterexamples. For example, in the detection of feature interactions [13], the resolution often requires a comprehensive view of the problem. Therefore, we would like to have a method to detect all counterexamples prior to modifying the model. However, the set of all counterexamples is often too large to generate and comprehend.

In this paper, we define a series of abstractions that can be used to represent the complete set of counterexamples to an invariant property in a limited and, hopefully, manageable manner. Each abstraction level creates equivalence classes of counterexamples. One representative of each equivalence class is included in the set of counterexample paths presented to the user. Because it is common that models are described using some form of extended finite state machine (EFSM) with control states and transitions, we abstract the counterexamples based on the modelling concepts of control states and transitions. An abstract counterexample is a path of transitions leading to a control state in which the invariant fails. An EFSM can manipulate data values through transition triggers and actions, therefore, a path of transitions is actually

a set of paths of the model with varying data values. If the ratio of control states to complete states (data plus control states) is low, this abstraction provides a significant reduction in the set of counterexamples.

We present a method to generate our abstraction levels on-the-fly during iterations of model checking runs. We execute a cycle of (1) run model checker to find counterexample, (2) abstract counterexample to its equivalence class, and (3) re-run model checker on the same model, ruling out all counterexamples within the same equivalence class. We represent the equivalence class of each counterexample as a property in linear temporal logic (LTL) [24]. By ruling out counterexamples on-the-fly, we greatly reduce the number of model checking iterations needed as compared to a process that generates all counterexamples and then groups them into equivalence classes after the model checking process (*e.g.,* [3] [2] [18]). Because the counterexample abstractions are represented in LTL, our method can be used with many model checkers. We used Cadence SMV [25] to generate our results.

In Section 2, we define the generic EFSM formalism we will use in this paper. In Section 3, we define our series of counterexample abstractions and illustrate their meaning using a small example. In Section 4, we show how the equivalence classes for each level of abstraction can be represented in LTL on-the-fly. We demonstrate the use of our methodology on two automotive feature design models in Section 5. Section 6 compares our contributions to related work. We conclude in Section 7 and discuss future work.

# 2 Extended Finite State Machines

An EFSM is a model with a finite set of control states and transitions, such as a Statechart [15]. We explain our method using a generic flat EFSM, but it can be generalized to models with hierarchical control states and concurrency.

The syntax of an **EFSM** consists of a tuple $\langle CS, InitCS, IV, OV, InitV, T\rangle$ where $CS$ is a set of control states, $InitCS$ is a set of initial control states ($InitCS \subseteq CS$), $IV$ is a set of input variables, $OV$ is a set of output (controlled) variables ($IV \cap OV = \emptyset$), $InitV$ is a set of initial values for input and output variables, and $T$ is a set of transitions. Each transition has a name, a source and destination control state, and a label of the form $(c)/a$, where $(c)$ is a condition on the variables ($IV \cup OV$) called a guard, and $a$ is a set of assignments to output variables. Guards and actions of transitions are optional. Variables have a finite set of possible values, and their initialization is optional. Events, which are often present in EFSMs, can be modelled as Boolean variables with values that do not persist. Figure 1 is an example of a simple EFSM. 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.

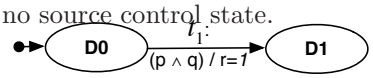


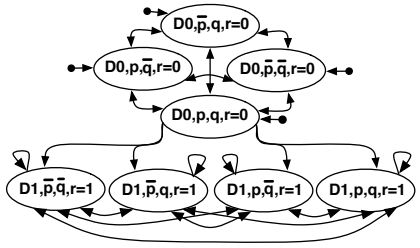

Figure 1: Simple EFSM

The semantics of an EFSM is a set of paths, each of which is a sequence of configurations. Each **configuration** consists of a control state, and a set of values for the variables. A transition is taken in a step when the model is in a configuration consisting of the source control state of the transition and the variables satisfy the guard of the transition. For example, transition $t_1$ is taken when the model is in source control state **D0** and the input variables p and q have the value *true*. When a transition is taken, the model moves to a configuration containing the destination state and the effects of executing the assignments to 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 **D1**, changing the value of the controlled variable r to 1.

Semantically, every control state has a single self-looping transition, which is taken when no guard on any other transition exiting the state is satisfied. These transitions ensure that at every configuration there is a next configuration. We call these transitions **non-contributing** because they do not contribute computationally to creating a counterexample. There are no actions associated with non-contributing transitions, but input changes might occur. $T$ is the set of **contributing** transitions [1].

[1] A self-looping transition in $T$ with no guard or actions implicitly has the guard *true*, unlike non-contributing transitions.

We can describe the meaning of an EFSM in a Kripke structure (KS). In a KS, the control states of an EFSM are typically modelled as a variable taking on the possible state names as values. The other EFSM variables are modelled as KS variables of appropriate types. A KS state represents a configuration of the model with values for each of the KS variables. Figure 2 shows the KS for the EFSM in Figure 1. In our example, the EFSM has two control states, whereas the Kripke structure has 8 KS states. Transition $t_1$ in Figure 1 corresponds to four transitions in the KS. We can see that the control states of the EFMS 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 state space is the reachable set of KS states within the cross product of the possible values of all the KS variables. The KS state space is usually significantly larger than the set of control states of the EFSM.




```
1  MODULE main(p,q,r) {
2   tr :{tn, t1};
3   cs :{D0, D1};
4   init(cs) := D0;
5   init(r) := 0;
6
7  next(tr):=
8   switch(cs){
9    D0: if (p & q) {t1}
10       else {tn};
11   D1: tn;  };
```

```
12 next(cs):=
13  switch(next(tr)){
14    t1: D1;
15    tn: cs;       };
16
17 next(r):=
18  switch(next(tr)){
19    t1: if (p & q) {1}
20        else {r};
21    tn: r;        };
22 }
```

Figure 2: Kripke structure and SMV model for EFSM in Figure 1

Figure 2 also shows an SMV model of the KS of Figure 1. In this paper, we will always use the variable name ***cs*** for the control state variable, which has the value of the current control state of the configuration. In the SMV model, we include a variable, ***tr***, which has the value of the name of the last transition taken in the model. The next value of *tr* is used to update the other variables values [2]. We use the transition value ***tn*** to represent a non-contributing transition, chosen only if no other transition guards are satisfied in a configuration. Because non-contributing transitions have no effect on control states and output variables, and every control state has one such transition, we can use the variable value *tn* as the transition name for all non-contributing transitions in the EFSM. The counterexamples generated by SMV are in terms of KS states; next, we propose a more limited representation by abstracting counterexamples.

# 3 Counterexample Abstraction

In this section, we describe our series of abstraction levels for the counterexamples produced by an EFSM represented as a KS. The intuition behind our counterexample abstractions is that the paths of the KS that go through the same sequence of control states and transitions, but vary in data, should be considered the same from the point of view of the EFSM. Since the user views the model as an EFSM (not as a KS), repeated counterexamples that follow the same path in the EFSM with different data do not add to the user's understanding of the error or inconsistency that the counterexample illustrates. Thus, by grouping the counterexamples based on the path through the EFSM, we create a useful abstraction of the counterexamples. If the ratio of control states to KS states is low, then this abstraction technique greatly reduces the set of counterexamples, potentially making it manageable and useful. For each equivalence class, the user is provided with a representative counterexample including the sequence of KS data values that lead to the error.

A counterexample reported by a model checker may contain loops, therefore the set of all counterexamples can have an infinite number of elements by varying the number of iterations of these loops. Let ***CE*** be the

[2]In theory, transition names can often be implemented using a macro. However, in Cadence SMV, macros do not appear in counterexamples, and we make use of the transition name in our on-the-fly abstraction method presented in Section 4.

set of counterexamples that could be reported initially by model checking an invariant property under no assumptions about the order of counterexample generation. In order to give a measure of how much abstraction we achieve, we define the finite set ***FIPaths*** (failed invariant paths), which contains finite paths that end in a KS state that fails the invariant. Each element in *FIPaths* is generated from a counterexample $c$ in *CE* such that
$FIPaths = \{p \mid \exists c \in CE \bullet p = reduce\_vals(reduceKS(trunc(contribute(c))))\}$ where:

- *contribute(c):* Removes all non-contributing transitions from $c$. Non-contributing transitions allow the environment to change input variables, letting the model make progress, but we are only interested in the inputs that cause the counterexample. Because the transition name included in the configuration is the last transition taken, the inputs in the last KS state of a sequence of non-contributing transitions are copied back to the KS state with the contributing transition just before the sequence begins [3].
- *trunc(c):* Truncates the path $c$ such that there are no configurations beyond the last KS state in the sequence that fails the invariant.
- *reduceKS(c):* Removes loops that reach the same KS state more than once in $c$.
- *reduce_vals(c):* 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. The transition name contains the last transition taken and the inputs to the model are used to calculate the next configuration (not the current configuration) and therefore neither values contribute to the computation that fails the invariant.

One element of *CE* may contain multiple states that fail the invariant. However, each of the subpaths ending at a state that fails the invariant could be another element of *CE*, and therefore creates its own corresponding element in *FIPaths*.

Within the general idea of abstracting counterexamples based on control states and transitions, there are a variety of more precise abstractions that may be useful at different times during the analysis process or in different domains. These abstractions consider which paths of the EFSM should be considered the "same" from an abstract point of view.

We use the EFSM model in Figure 3 to illustrate our counterexample abstraction levels. In the example EFSM, the initial value of output variable s is 0. To demonstrate our method, we have shaded three control states and assume that these states correspond to *some* KS state that fails the invariant. We call these **failed invariant control states (*FICS*)**. As an example of the potential reduction rate of our abstraction method, consider transition $t_6$. It represents 8 transitions in the KS [4]. Table 1 shows the equivalence classes created by each of our abstraction levels for the model of Figure 3. The number beside an equivalence class in Table 1 shows the number of elements of *FIPaths* in the class. The total number of elements in *FIPaths* is 183, which covers all the KS looping and data variations in counterexamples.

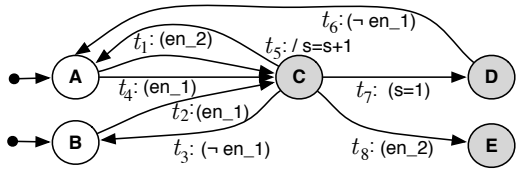



Figure 3: Example EFSM

Next, we present the counterexample abstraction levels from the most detailed to the least detailed. We use the following notation with respect to path, $p$, to describe our counterexample abstraction levels:

- *src(p):* The source control state of the first transition in $p$.
- *dest(p):* The destination control state of the last transition in $p$.
- *last_trans(p):* The last transition taken in $p$.
- *trans_seq(p):* The sequence of transitions taken in $p$.

[3] The definition of *contribute* would be simpler if $tr$ held the value of the next transition to be taken. However, we found the SMV models more intuitive and the description of the levels of abstraction and their representation in LTL simpler using $tr$ as the last transition taken.

[4] The source configuration has values for the input variables of en_2=*true/false*, while the destination configuration has values of en_1=*true/false* and en_2=*true/false*.

| Level 1 | Level 2 | Level 3 | Level 4 |
|---|---|---|---|
| $[\langle t_1 \rangle]$ - 43 | $[t_1]$ - 53 | [**A**,**C**] - 58 | [**C**] - 86 |
| $[\langle t_4 \rangle]$ - 15 | $[t_4]$ - 21 | | |
| $[\langle t_2 \rangle]$ - 28 | $[t_2]$ - 12 | [**B**,**C**] - 28 | |
| $[\langle t_1, t_7 \rangle]$ - 21 | $[t_7]$ - 67 | [**A**,**D**] - 45 | [**D**] - 67 |
| $[\langle t_4, t_7 \rangle]$ - 24 | | | |
| $[\langle t_2, t_7 \rangle]$ - 22 | | [**B**,**D**] - 22 | |
| $[\langle t_1, t_8 \rangle]$ - 12 | $[t_8]$ - 30 | [**A**,**E**] - 18 | [**E**] - 30 |
| $[\langle t_4, t_8 \rangle]$ - 6 | | | |
| $[\langle t_2, t_8 \rangle]$ - 12 | | [**B**,**E**] - 12 | |

Table 1: Summary of series of counterexample abstractions for Figure 3.
The total number of elements of *FIPaths* is 183.

- *reduceEFSM(p):* Removes EFSM loops from $p$. An **EFSM loop** is one that reaches the same control state in $p$ more than once. While EFSM loops may change data and therefore be necessary to fail the invariant, when using *reduceEFSM* we consider all paths with iterations of an EFSM loop to be equivalent.

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 -** All the paths that have the same sequence of transitions after removing any EFSM loops in each path are considered equivalent.

***Definition 1:*** $\forall p \in FIPaths \bullet [p] = \{q \in FIPaths \mid$
$trans\_seq(reduceEFSM(p)) = trans\_seq(reduceEFSM(q))\}$

There are nine equivalence classes at Level 1 for the EFSM in Figure 3. For example, the path $\langle t_2, t_5, t_1, t_8 \rangle$ is part of the equivalence class $[\langle t_2, t_8 \rangle]$ after removing the EFSM loop with respect to control state **C**.

The equivalence classes of Level 1 can also be used to construct the submachine representing **total transition coverage** because every transition of the EFSM that contributes to a counterexample is included in one equivalence class.

**Level 2: Distinct Last Transitions -** All the paths that have the same last transition are considered equivalent.

***Definition 2:*** $\forall t \in T \bullet [t] = \{p \in FIPaths \mid last\_trans(p) = t\}$

An equivalence class for a transition $t$ is empty if that transition is not the last transition in a path within the set of *FIPaths*. There are five non-empty equivalence classes at Level 2 for the EFSM in Figure 3. For example, the path $\langle t_2, t_5, t_1, t_8 \rangle$ is part of the equivalence class $[t_8]$.

**Level 3: Distinct Initial and Final States -** All the paths that have the same initial control state and final control state are considered equivalent.

***Definition 3:*** $\forall i \in InitCS, \forall s \in FICS \bullet [i, s] =$
$\{p \in FIPaths \mid src(p) = i \wedge dest(p) = s\}$

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 six non-empty equivalence classes at Level 3 for the EFSM in Figure 3. For example, the path $\langle t_2, t_5, t_1, t_8 \rangle$ is part of the equivalence class [**B**,**E**].

**Level 4: Distinct Final States -** All the paths that lead to the same final control state are considered equivalent.

***Definition 4:*** $\forall s \in FICS \bullet [s] = \{p \in FIPaths \mid dest(p) = s\}$

There are three equivalence classes at Level 4 for the EFSM in Figure 3. For example, the path $\langle t_2, t_5, t_1, t_8 \rangle$ is part of the equivalence class [**E**].

Figure 4 shows the partial order relationship among the abstraction levels. Two levels are related when an equivalence class of one level is always strictly subdivided into more equivalence classes at the next lower level, meaning that no counterexamples that were previously considered distinct in the abstraction become equivalent at the lower level. Levels 3 (Distinct Initial and Final States) and 2 (Distinct Last Transition) are not linearly related because two paths with distinct initial states that reach the same final control state may have shared suffixes.

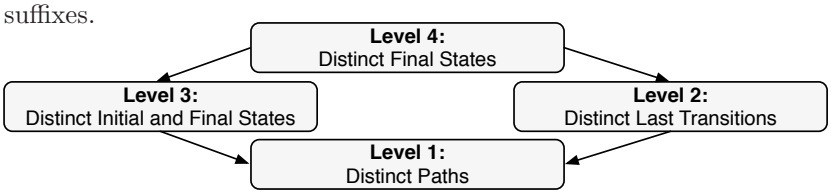



Figure 4: Partial order among counterexample abstraction levels

# 4 On-the-fly LTL Counterexample Abstraction

In this section, we explain how to represent our series of abstractions on-the-fly using LTL properties. In previous work [3] [2] [18], to generate an abstract set of counterexamples, the user first generated all the counterexamples by iterating the model checker and then grouping them into equivalence classes. By ruling out all equivalent counterexamples on-the-fly, we avoid the need to generate all counterexamples, thereby substantially reducing the number of iterations of the model checker. Our LTL properties describe equivalence classes of counterexamples rather than the complete list of counterexamples.

Our on-the-fly abstraction method is illustrated in Figure 5. We assume that the model input to the process contains the following:

**inv**: Macro that specifies the criteria to identify an error, inconsistency or contradiction in the model, *i.e.,* the invariant property.

**contribute**: Macro that specifies $(\neg(tr = tn) \vee \mathsf{final\_states})$, a condition indicating that only contributing transitions are taken unless the model reaches a final control state. Final states are control states that are not source states of any contributing transition.

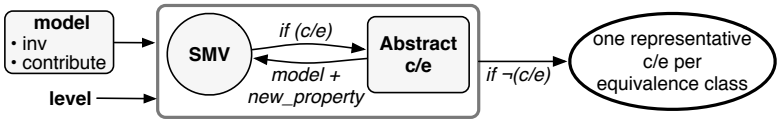



Figure 5: On-the-fly abstraction process

In our method, we iteratively (1) ask SMV to generate a counterexample, (2) abstract the counterexample to its equivalence class for the desired level, (3) represent this abstraction as an LTL expression, (4) create a new property that is the disjunction of this LTL expression with the invariant and the LTL expressions representing previously generated counterexamples, and (5) repeat 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, we disallow the generation of any more counterexamples in that equivalence class. We produce as output one representative counterexample per equivalence class. This iterative process runs automatically via scripts, and it is repeated until no more counterexamples are found.

For every abstraction level, we begin by checking the property

$$\mathrm{G}(\mathsf{contribute}) \rightarrow \mathrm{G}(\mathsf{inv})$$

to get the first counterexample. After this first iteration, we add an LTL expression that represents the equivalence class of this counterexample according to the desired level of abstraction. These LTL expressions are summarized in Table 2, and explained next in order from least complex to most complex (reverse order of Section 3). We use the following operators of LTL:

| Level | LTL property after $m$ iterations |
|---|---|
| **4** | G (contribute) $\rightarrow$ G (inv $\vee$...$\vee$ ($cs$=$FI_1^j$) $\vee$...$\vee$ ($cs$=$FI_m^j$)) |
| **3** | G (contribute) $\rightarrow$ (G (inv) $\vee$<br>(($cs$=$I_1$) $\wedge$ G(inv $\vee$ ($cs$=$FI_1^1$) $\vee$...$\vee$ ($cs$=$FI_1^j$)))<br>$\vee$...$\vee$<br>(($cs$=$I_m$) $\wedge$ G(inv $\vee$ ($cs$=$FI_m^1$) $\vee$...$\vee$ ($cs$=$FI_m^j$)))) |
| **2** | G (contribute) $\rightarrow$ G (inv $\vee$ ($tr$=$t_1^n$) $\vee$...$\vee$ ($tr$=$t_m^n$)) |
| **1** | G (contribute) $\rightarrow$ ((G (inv)) $\vee$<br>(($cs$=$I_1$) $\wedge$ (inv U (($tr$=$t_1^1$) $\wedge$...$\wedge$ ((inv $\vee$ ($cs$=$FI_1^1$) $\vee$...$\vee$ ($cs$=$FI_1^{j-1}$))<br>U (($tr$=$t_1^n$) $\wedge$ G (inv $\vee$ ($cs$=$FI_1^1$) $\vee$...$\vee$ ($cs$=$FI_1^j$)))))))<br>$\vee$...$\vee$<br>(($cs$=$I_m$) $\wedge$ (inv U (($tr$=$t_m^1$) $\wedge$...$\wedge$ ((inv $\vee$ ($cs$=$FI_m^1$) $\vee$...$\vee$ ($cs$=$FI_m^{j-1}$))<br>U (($tr$=$t_m^n$) $\wedge$ G (inv $\vee$ ($cs$=$FI_m^1$) $\vee$...$\vee$ ($cs$=$FI_m^j$))))))))) |

Table 2: Counterexample abstraction properties per level for a set of $m$ elements of *FIPaths*: $\langle I_1, t_1^1, ..., FI_1^1, ..., t_1^n, FI_1^j\rangle$ $\ldots$ $\langle I_m, t_m^1, ..., FI_m^1, ..., t_m^n, FI_m^j\rangle$ where $I_i$ are initial states, $FI_i^j$ are elements of *FICS* and $t_i^k$ are transition names

- Globally: **G** $\psi$ means that $\psi$ must hold on the entire path.
- Until: $\psi$ **U** $\phi$ means that $\phi$ must hold at the current or a future position, and $\psi$ has to hold until that position. From there on, $\psi$ does not need to hold.

A counterexample generated by the model checker may contain more than one KS state that fails the invariant, so we can create multiple elements of *FIPaths* from it. All subpaths of the counterexample that end in a KS state that fails the invariant are elements of *FIPaths* because the shorter paths could have been returned as counterexamples themselves. One equivalence class is generated for each of these paths. Thus, the number of iterations of the cycle can be less than the number of equivalence classes at a level of abstraction. The representative counterexample chosen as output of our method is an element of *FIPaths*.

For Levels 1-4, the first step in the process is to calculate the set $P$, consisting of the elements of *FIPaths* generated from the counterexample returned by the model checker. The macro contribute implements the definition of the function *contribute* in Section 3, ensuring that the counterexample returned by the model checker only contains contributing transitions. The function *trunc* is implemented by creating all subpaths of the counterexample ending in a KS state that fails the invariant. Then, the functions *reduceKS* and *reduce_vals* are applied to each subpath. The result of the application of these functions is the set of paths $P$.

**Level 4: Distinct Final States -** For each path, $q$, in $P$, we add to the property a disjunction with an LTL expression that $cs$ has the value of the last control state in $q$ (which is always an element of *FICS*), forcing the model checker to find another control state where the invariant fails. Because any element of $P$ ends at a KS state in which the invariant fails, effectively we disjunct the LTL expression that $cs$ can have the value of any element of *FICS* found in the counterexample from which $P$ was generated.

**Level 3: Distinct Initial and Final States -** For each path $q$, in $P$, we add to the property a disjunction with an LTL expression describing the path that starts with the initial control state of $q$, and that includes the last control state in $q$ (which is an element of *FICS*). Over multiple paths, we group all final control states with the same initial control state together in a disjunction with the invariant. In the next iteration, the model checker searches for paths that either start with the same initial control state, but end at a different control state that fails the invariant, or start with a different initial control state and end at a control state where the invariant fails.

**Level 2: Distinct Last Transitions -** For each path, $q$, in $P$, we add to the property a disjunction with an LTL expression that $tr$ has the value of the last transition taken in $q$ (which leads to a control state in *FICS*), forcing the model checker to find another transition that leads to a control state in *FICS*.

**Level 1: Distinct Paths -** For each path, $q$, in $P$, we add to the property a disjunction with an LTL expression that makes the model checker accept any path with the same sequence of transitions as $q$, and all EFSM looping variations of this path. For example, for the path $\langle t_1, t_2 \rangle$ of Figure 6, some looping variations are the sequences of control states **A**-**X**-**A**-**B**-**C**, **A**-**B**-**Y**-**B**-**C**, and **A**-**X**-**A**-**B**-**Z**-**A**-**B**-**C**, where **X**,**Y**,**Z** can be a sequence of states. By including the looping variations, the model checker will not report them separately.

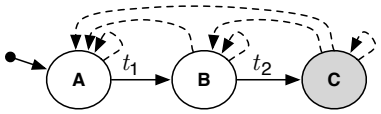



Figure 6: Looping variations of path $\langle t_1, t_2 \rangle$

The LTL expression that recognizes all EFSM looping variants of a path is constructed using the LTL operator Until. For the path $\langle I, t_1, ..., t_n, FI \rangle$, the LTL expression has the form

$((cs{=}I) \wedge (\mathsf{inv}\ \mathrm{U}\ ((tr{=}t_1)\ ...\ \wedge (\mathsf{inv}\ \mathrm{U}\ ((tr{=}t_n) \wedge \mathrm{G}(\mathsf{inv} \vee (cs{=}FI)))))))$

Through the use of the Until operator, we allow paths with loops whose states all satisfy the invariant to be included in this equivalence class.

If the path $q$ includes more than one control state in *FICS*, then the invariant is allowed to be violated by these control states partway through the path. For example, the path $\langle t_1, t_8 \rangle$ in Figure 3 has control states **C** and **E** in *FICS*. The property must accept the occurrence of control state **C** before $t_8$, which is done by disjuncting the constraint that $cs$ can be **C** to the condition that checks that the invariant holds before the next contributing transition, *i.e.,* $\underline{(\mathsf{inv} \vee (cs{=}\mathrm{C}))}$. Then, the LTL expression for this equivalence class is

$((cs{=}\mathrm{A}) \wedge (\mathsf{inv}\ \mathrm{U}\ ((tr{=}t_1) \wedge$
$\underline{((\mathsf{inv} \vee (cs{=}\mathrm{C}))}\ \mathrm{U}\ ((tr{=}t_8) \wedge \mathrm{G}(\mathsf{inv} \vee (cs{=}\mathrm{C}) \vee (cs{=}\mathrm{E}))))))))$

Our method does not rely on the order in which the counterexamples are generated (such as generation of the shortest one first). It may be the case that information discovered from a counterexample generated later in the cycle subsumes information found earlier. For example, when generating the elements of *FIPaths* from a counterexample that contains multiple states that fail the invariant, one of these elements may be a path that was previously discovered. While this may make the LTL expression representing the equivalence classes discovered so far longer than needed, logically, it does not change the resulting set of equivalence classes found. For simplicity of our method, we chose to make the abstraction process for each counterexample independent of any equivalence classes seen in previous cycles.

Table 3 shows the number of cycle iterations, the number of equivalence classes per level (as shown also in Table 1), the maximum BDD size for all cycles, and the total time for all iterations of the analysis of Figure 3.

| **Abstraction Level** | **Iterations** | **Equiv. Classes** | **BDD Nodes** | **Time** |
|---|---|---|---|---|
| **Level 4** | 3 | 3 | 846 | 7.4s |
| **Level 3** | 6 | 6 | 2359 | 7.8s |
| **Level 2** | 5 | 5 | 1653 | 7.6s |
| **Level 1** | 7 | 9 | 18308 | 8.3s |
| | | **# Elements** | | |
| ***FIPaths*** | 102 | 183 | 1905116 | 315m20s |

Table 3: Statistics for the analysis of Figure 3

We counted the number of elements of *FIPaths* for the example in Figure 3 by iteratively adding to the property a disjunction with an LTL expression exactly (*i.e.,* not abstractly) representing the element of

*FIPaths* matching the counterexample. The LTL expression used is similar to that of Level 1, but complete KS configurations are used rather than just control states and transitions. We reduced the number of iterations of this process by finding multiple elements of *FIPaths* from a counterexample when possible (as is done for all the levels).

# 5 Case Studies

In this section, we show how our method is used to check for errors in the functional requirements of two automotive features. We verified two of our non-proprietary feature models, *Collision Avoidance (CA)* and *Emergency Vehicle Avoidance (EVA)* [21], using our counterexample abstraction method. These features are generally known as "Active Safety Systems" because they use sensors, cameras, radar, and sometimes even GPS devices, to help the driver control the vehicle. CA helps to prevent or mitigate collisions when driving forward. EVA pulls the vehicle over when an emergency vehicle needs the road to be cleared. Our automotive features are representative in type and complexity to models that we have seen developed in practice, but do not include failure modes (*e.g.,* fail-safe states for degraded modes of operation).

In previous work [22] [23], we translated these feature models created in MATLAB's Stateflow [1] to SMV. Automotive features are often hierarchical state machines. Thus, our translation creates one state name variable per hierarchy level of the EFSM in the SMV model. In this case, the constraints on *cs* 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. Therefore, Level 3 has the same result as Level 4.

For CA and EVA, we checked the functional requirement that no feature can request braking higher than 60% of the total braking force, *i.e.,* (set_Brake $< 60$). The results of our analysis are summarized in Table 4, showing the number of iterations of the model checker, the maximum BDD nodes for all iterations and the time taken to complete all the cycles. The model checking verification runs were performed on a 2GHz Intel Xeon MP CPU with 8GB of RAM. For CA and EVA, all the counterexamples generated by the model checker contained only one state that fails the invariant. Therefore, in this case, the number of equivalence classes per level corresponds to the number of iterations.

| **Abstraction Level** | **CA** | | | **EVA** | | |
|---|---|---|---|---|---|---|
| | **Iterations** | **BDD Nodes** | **Time** | **Iterations** | **BDD Nodes** | **Time** |
| **Level 4** | 1 | 5658 | 7.5s | 1 | 4557 | 7.5s |
| **Level 3** | 1 | 1043 | 7.6s | 1 | 8477 | 7.6s |
| **Level 2** | 3 | 10656 | 8.0s | 2 | 8543 | 7.8s |
| **Level 1** | 5 | 305495 | 14.2s | 2 | 35874 | 8.3s |

Table 4: Case study results

In this case study, it was not possible to generate the complete set of *FIPaths* because this set is too large and the size of the LTL property became so large that the model checker would not terminate. The complexity of LTL model checking depends on the size of the property, as well as on the size of the model. Thus, unlike in our illustrative example, we cannot provide the number of elements in *FIPaths* that belong to each equivalence class.

Our method allows a feature engineer to have complete information about the set of counterexamples grouped at meaningful levels of abstraction even though all counterexamples cannot be generated. For example, to know in which control states an error is found, a feature engineer would ask for the results from Level 4. There are normally more paths in Level 2 than in Level 4 because each control state usually has many incoming transitions.

# 6 Related Work

Our work differs from approaches that use model checking for debugging in that we find all abstract counterexamples instead of providing only certain classes of witnesses. Copty *et al.* use a model checker engine to generate BDD's that compute counterexamples of a given length [8], while we do not restrict the length of the counterexamples. Other approaches generate all counterexamples and then analyze or abstract them. Groce and Visser use an algorithm to find the differences between counterexamples and traces with no error, by calling a model checker that is able to report all counterexamples [14]. Some model checkers, such as SPIN [17], include the ability to generate all the counterexamples by having the model checking algorithm keep searching the KS state space even when the property fails. No modification of the model or the property is necessary, but the counterexamples generated are in terms of KS states, unlike our method that produces abstract counterexamples. Chechik and Gurfinkel merge the information of all the counterexamples to a property into an abstract one, displaying it as a proof-like tree [6]. Each state of the abstract counterexample path is a summary of the data variations, from all the counterexamples that caused the error, whereas our method only produces one representative counterexample with one set of data values. Several approaches attempt to identify the cause of an error in a counterexample. Ball *et al.* identify the transitions in an error trace that are not part of any correct trace [4]. Their algorithm first calls a model checker; if the property fails, they compute the set of correct transitions, and then iteratively find different error causes by comparing the transitions that are part of the error trace but do not belong to any correct trace. Our work does not attempt to show the cause of an error.

Our method finds equivalence classes on-the-fly, so multiple counterexamples in the same class are never generated, whereas some test case generation approaches apply duplication elimination after all counterexamples are generated [3], [2], [18], or do not apply duplication elimination at all [16]. Test case generation approaches create a test case per property by (1) generating a counterexample with a model checker, and (2) turning the counterexample into a concrete test case. While we do not create concrete tests for simulation or verification, in the future we might integrate the ability to simulate counterexamples generated. Ammann *et al.* use SMV and mutation analysis to create test cases for structural coverage [3], [2]. Gargantini and Heitmeyer construct properties from an SRC specification for structural coverage [11]. Their concept of equivalence of transitions, *i.e.,* coverage of all variable values as defined by the guard in an SCR table, is like our concept of transition names. Their tool has to check if a case is not already covered, which we disallow by adding properties to the model at each iteration. Heimdahl *et al.* use the model's syntax tree to create properties for structural coverage [26], [16]. Engels *et al.* use SPIN for conformance testing given a set of properties written manually [9]. Hong *et al.* uses SMV to find witnesses for control-flow or data-flow criteria [19], [18]. Their use of EFSMs require paths to end at an initial state or at an state marked by the testers, whereas our paths end at a control state in which the invariant fails.

Our work uses model checking tools directly instead of generating a submodel that represents the desired coverage criteria. Benjamin, Geist, Hartman *et al.* developed the tool GOTCHA for state and transition coverage [12], [5], [10]. Their use of projected transitions reduces the model by considering only certain variables, while our concept of transition names does not change the model and groups together paths that have the same set of control states and transitions (but vary in data). Jéron and Morel created a tool, TGV, to construct the subgraph of the system and the specification to perform conformance testing [20]. Zeng *et al.* developed an algorithm for branch coverage [27]. Their algorithm has as input a list of properties to check, whereas we create the properties that disallow classes of counterexamples on-the-fly.

# 7 Conclusion

In this paper, we defined a series of abstractions that represent the complete set of counterexamples to an invariant in a reduced set of equivalence classes, which is easier to generate and comprehend than the whole set of counterexamples. We have shown how to represent these abstractions on-the-fly as LTL properties

to be used by a model checker. The abstractions we defined are based on the modelling abstractions of the control states and transitions of a extended finite state machine (EFSM). Our counterexample abstractions and their representation as LTL properties can be used for any model that is expressed as an EFSM and that would benefit from a more abstract representation of counterexamples. We demonstrated the reduction produced by our counterexample abstractions in the verification of an invariant for two automotive feature design models.

In this paper, our methodology has been explained for a single EFSM, but we are working on generalizing our levels of counterexample abstraction and our proposed on-the-fly LTL abstraction method for multiple concurrent EFSMs without flattening the model. We plan to use our technique for the detection of feature interactions between automotive features. For these large combined models, we expect that even some of our levels of abstraction will generate LTL properties that create a model checking problem that is too large to complete. We are working on techniques that reduce the size of the LTL property needed by using expressions that subsume a set of LTL expressions.

# References


[1] MathWorks Stateflow Documentation, `http://www.mathworks.com/access/helpdesk/help/toolbox/stateflow/`.

[2] P. Ammann, W. Ding, and D. Xu. Using a model checker to test safety properties. In *Proc. 7th IEEE Int'l Conf. on Engineering of Complex Computer Systems*, pages 212–221, 2001.

[3] P. E. Ammann, P. E. Black, and W. Majurski. Using model checking to generate tests from specifications. In *Proc. 2nd IEEE Int'l Conf. on Formal Engineering Methods*, pages 46–54. IEEE Computer Society, 1998.

[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] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets. A study in coverage-driven test generation. In *Proc. 36th ACM/IEEE Conf. on Design Automation*, pages 970–975. ACM, 1999.

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

[7] E. M. Clarke, O. Grumberg, , and D. A. Peled. *Model Checking*. The MIT Press, First edition, 2000.

[8] F. Copty, A. Irron, O. Weissberg, N. P. Kropp, and G. Kamhi. Efficient debugging in a formal verification environment. In *Proc. 11th Conf. on Correct Hardware Design and Verification Methods*, pages 275–292, 2001.

[9] A. Engels, L. M. G. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. In *Proc. 3rd Int'l Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 384–398. Springer-Verlag, 1997.

[10] G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected state machine coverage for software testing. In *Proc. 2002 ACM SIGSOFT Int'l Symposium on Software Testing and Analysis*, pages 134–143. ACM, 2002.

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

[12] D. Geist, M. Farkas, A. Landver, Y. Lichtenstein, S. Ur, and Y. Wolfsthal. Coverage-directed test generation using symbolic techniques. In *Proc. Int'l Conf. on Formal Methods in CAD*, pages 143–158, 1996.

[13] N. D. Griffeth and Y.-J. Lin. Extending telecommunications systems: The feature-interaction problem. *IEEE Computer*, 26(8):14–18, 1993.

[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. *Sci. Comput. 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 *3rd Int'l Worshop on Formal Approaches to Testing of Software*. Springer, 2003.

[17] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, first edition, 2003.

[18] H. S. Hong, S. D. Cha, I. Lee, O. Sokolsky, and H. Ural. Data flow testing as model checking. In *Proc. 25th Int'l Conf. on Software Engineering*, pages 232–243, 2003.

[19] H. S. Hong, I. Lee, O. Sokolsky, and S. D. Cha. Automatic test generation from statecharts using model checking. In *Proc. Workshop on Formal Approaches to Testing of Software*, pages 15–30, 2001.

[20] T. Jéron and P. Morel. Test generation derived from model-checking. In *Proc. 11th Int'l Conf. on Computer Aided Verification*, pages 108–121. Springer-Verlag, 1999.

[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 *Proc. 26th Int'l 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 *Proc. 2008 Int'l Workshop on Modeling in Software Engineering*, pages 45–50. ACM Press, 2008.

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

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

[26] S. Rayadurgam and M. P. Heimdahl. Coverage based test-case generation using model checkers. In *Proc. 8th IEEE Int'l Conf. and Workshop on the Engineering of Computer Based Systems*, pages 83–91. IEEE Computer Society, 2001.

[27] H. Zeng, H. Miao, and J. Liu. Specification-based test generation and optimization using model checking. In *Proc. 1st Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*, pages 349–355. IEEE Computer Society, 2007.