Fate and Free Will in Error Traces^{*}

HoonSang Jin¹, Kavita Ravi², and Fabio Somenzi¹

¹ University of Colorado at Boulder

{Jinh,Fabio}@Colorado.EDU

² Cadence Design Systems

kravi@cadence.com

Abstract. The ability to generate counterexamples for failing properties is often cited as one of the strengths of model checking. However, it is often difficult to interpret long error traces in which many variables appear. Further, a traditional error trace presents only one possible behavior of the system causing the failure, with no further annotation. Our objective is to “capture more of the error” in an error trace to make debugging easier. We present an enhanced error trace in an alternation of *fated* (forced) and *free* segments. The fated segments show unavoidable progress towards the error while the free segments show choices that, if avoided, may have prevented the error. This segmentation raises the questions of whether the fated segment should indeed be inevitable and whether the free segments are critical in causing the error. Addressing these questions may help the user analyze the error better.

1 Introduction

The algorithmic nature of model checking results in an important debugging capability—the generation of *counterexamples*. A counterexample provides evidence of failure of a *universal* property. Dually, it is possible to provide a witness to an *existential* property. Both counterexamples and witnesses, henceforth referred to as *traces*, are useful in debugging or exploring a design. Counterexamples that are easy to analyze help a developer diagnose problems quickly.

A counterexample may be a finite or infinite trace. The failure that it demonstrates may indicate a flaw in the design being verified, may point to a mistake in modelling its environment, or may be an artifact of the abstraction applied to obtain a simple model. State-of-the-art model checkers provide only one trace. Oftentimes, however, users of model checkers find a single error trace difficult to relate to the exact bug in the design. This may be due to a combination of reasons:

- A trace of a large design often involves tens of state and input variables over tens of cycles. Good presentation in the form of waveforms or diagrams helps manage this complexity, but presentation alone cannot help in the understanding of the problem.
- Counterexamples to invariants are usually guaranteed to be of minimum length. However, for liveness properties, the length of the trace is only heuristically reduced. More importantly, the algorithms that generate the error traces do not attempt to

^{*} This work was supported in part by SRC contract 2001-TJ-920 and NSF grant CCR-99-71195.

minimize the number of variables appearing along the trace or the changes in these variables (e.g., the number of events). Oftentimes, irrelevant events may occur and distract or mislead the user as to the cause of the failure.

- An error trace provides only one possible path to the failure. The set of all counterexamples is one way to characterize the error completely but producing all counterexamples is computationally hard, hence practically infeasible.
- Often, not all events in the error trace have the same importance, and it is difficult to spot the *crucial events*. In the case of a failing invariant, the last state of the trace is a so-called *bad state*—one in which the invariant does not hold. However, the cause of the failure may have to be looked for several cycles earlier, when an event took place that made the catastrophe inevitable, or at least very likely.

What is most useful in debugging are behaviors of the design most germane to the failure, perhaps a counterexample marked up with important events. Our approach focuses on one such possible property—“inevitability” in the design’s behavior towards the failure. We produce an error trace that is annotated with the inevitable segments. Identifying the foregone portions of a counterexample provides a classification of the events in the trace. The user may then analyze the error in terms of whether the foregone portions are correct or whether the remaining events were critically responsible for the error. We explain this in the following example.

Example 1. Consider a circuit for prioritized transfer of requests of two kinds—high priority (HI) and low priority (LO). The transfer is accomplished with two FIFOs, one for each kind of request. Requests arrive one at a time and are sent out in order within each priority. The transfer is controlled by a fast input clock and a slow output clock. A HI request is given priority over a LO request unless the LO request matches the address field of a HI request. In this case, the HI request is not issued until the matching LO request is sent out. Such designs are common in cache interfaces to main memory in processor design. In our implementation of this priority scheme, an unconstrained source of LO requests may cause starvation of the HI requests, as evidenced by the failure of the following CTL property.

$$\text{AG}(\text{match} \rightarrow \text{AF sav} \neq \text{hh}) .$$

In the property, *match* signals that the address field of a request at the head of the HI FIFO is the same as a LO FIFO request. The value of the head pointer to the HI FIFO (*hh*) is saved in *sav* when a match is detected; the variables differ when the HI request is issued. If the matched HI request is eventually issued, the property is satisfied.

An error trace is shown in Fig. 1. Each Column represents a variable of the design. All variables to the left of *inaddr* are state variables, and those to the right including *inaddr* are inputs. The error trace consists of a stem from the initial state to a state where a match is detected, and of a cycle back to that state, along which the HI request is not issued. Requests are indicated by *validin* that signals a valid request, incoming address *inaddr*, and type of request *hl*. The head and tail pointers respectively track the emptying and filling of the LO (*lt* and *lh* respectively) and HI FIFOs (*ht* and *hh* respectively). The first two requests are HI requests (*hl*=1) for address 0. This is followed by a LO request for the same address. This causes a match (*match*=1) two input clock

periods later, when the slow output clock (sloClk=1) ticks. Along the cycle, the LO FIFO is alternately filled and emptied (notice the values of lt and lh). Since there is always an entry with address 0 in the LO FIFO, the HI request starves. The vertical bars

	Stem----				Cycle----->																
Time	0	1	2	3	4	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
sloClk	0		1	0		0			1	0			1	0			1	0			1
match	0			1		1															
hh	0					0															
ht	0	1	2			2															
sav	0					0															
lh	0					0				1			2					3			
lt	0			1		1	2	3		0				1							
inaddr	0					0															
hl	1		0			0															
validin	1	0	1			1												0			
FFW	f	f		f		f	f			f				f							

Fig. 1. Counterexample for prioritized requests

and the f label on the last line of Fig. 1 are annotations of the error trace. The annotation at Column i pertains to the transition from Column i to Column $i + 1$. Details will be discussed in Sect. 4. The bars highlight the inevitable segments of the counterexample. The HI FIFO and the LO FIFO receive a request for the same address 0. As long as the LO requests with address 0 continue to enter the LO FIFO (free choices), the rest is inevitable. The annotated segments capture a high level picture of why the error occurred by showing that the LO requests starve the HI requests.

We propose an algorithm to generate an error trace divided into *fated* and *free* segments, so that the eventual demise of a system—as signaled by the failure of a property—can be interpreted as the joint accomplishment of Fate and Free Will. In our approach, Fate can be endowed with varying degrees of power, reflecting the various degrees to which it controls the environment of the system. We argue that this is desirable, and in Sect. 4 we present an approach to determine the extent of Fate’s sway.

The paper is organized as follows: Sect. 2 presents the background. Sect. 3 presents the main Fate and Free Will approach and two algorithms that implement it. They are followed by a comparison to related work (Sect. 5) and conclusions.

2 Preliminaries

Let $V = \{v_1, \dots, v_n\}$ be a set. We designate by V' the set $\{v'_1, \dots, v'_n\}$ consisting of the primed version of the elements of V . We define an *open system* as a 5-tuple $\langle V, W, I, R, \mathcal{F} \rangle$, where V is the set of (current) state variables, W is the set of input variables, $I(V)$ is the initial predicate, $R(V, W, V')$ is the transition relation, and $\mathcal{F} =$

$\{F_i(V)\}$ is the set of Büchi fairness conditions. The variables in V' are the next state variables. All sets are finite, and all variables range over finite domains. We assume that $R(V, W, V')$ is *deterministic*.

An open system defines a fair labeled transition structure in the usual way, with states Q_Ω corresponding to the valuations of the variables in V , and transition labels corresponding to the valuations of the variables in W . Conversely, a set of states $T \subseteq Q_\Omega$ corresponds to a predicate $T(V)$ or $T(V')$. Predicate $T(V)$ ($T(V')$) is the characteristic function of T expressed in terms of the current (next) state variables.

The positive (negative) cofactor of predicate $P(V)$ with respect to variable $v \in V$ is denoted by P_v ($P_{\neg v}$) and is obtained by letting $v = 1$ ($v = 0$) in $P(V)$. A *cube* is a predicate that consists of the conjunction of variables and their negations.

By existentially quantifying the input variables from R , one obtains a Kripke structure to which model checking algorithms for various specification mechanisms can be applied [Kur94, CGP99]. We assume that properties are specified in a temporal logic like CTL* [EH86], which conveniently subsumes both CTL [CE81] and LTL [WVS83, LP85]. For ω -regular properties [Kur94], we assume that model checking is reduced to language emptiness check on the composition of the given system and the Büchi automaton for the negation of the property.

In computing the set of states that reach a given target, a symbolic model checker stores the sets of states at a given distance from the target, called *onion rings*. The onion rings are used to produce shortest counterexamples for failing invariants. If forward search from the initial states is used to check an invariant, then a set of rings around the initial states is stored. The shortest path is traced backwards from a bad state t closest to the initial states (from the innermost onion ring). In each step, the predecessors of t are intersected with the onion ring immediately inside the one of t and a state is chosen from the intersection. The process is repeated until an initial state is reached.

An error trace for a liveness property consists of a path leading from an initial state to a fair cycle. The fair cycle consists of paths connecting states from all fairness constraints. Each sub-path is constructed with the help of a set of onion rings around the destination states. Hence, each sub-path is of minimal length, though the overall counterexample may not be minimal. A popular algorithm is described in [CGMZ95], and a comparison of various techniques can be found in [RBS00].

We will use standard future and past tense CTL notation, such as EX, its past-tense dual EY, EU, its past-tense dual ES and EG, with their usual semantics in the rest of this paper.

3 Fate and Free Will in Error Traces

An error trace is composed of *parts*, each of which is made up of *segments*. Dividing an error trace into segments characterized by Fate and Free Will is the main approach in this paper. A counterexample to an invariant consists of one part only: a path from an initial state to a bad state; a counterexample to a liveness property consists of several parts: the stem plus the parts that make up the fair cycle by connecting states satisfying different fairness constraints. In the following we consider dividing one part into *fated* and *free* segments. The method we develop can be applied to each part in turn.

3.1 The Fatal Game

The set of *controlling variables* of an open system $\Omega = \langle V, W, I, R, \mathcal{F} \rangle$, written $\Gamma(\Omega)$, is a subset of W . When Ω is understood, we shall simply write Γ . Informally, Γ consists of the variables that a hostile environment of Ω uses to force the behavior of the system, dictating its *fate*. The remaining variables in W are considered *non-controlling*. They are folded into the system, allowing the system *free-will* in their choice.

Given Ω , Γ , and a target predicate $T(V)$, a two-player concurrent reachability game [EJ91] can be defined as follows. The positions of the game correspond to the states of Ω . The two players are the (hostile) *environment* and the *system*. From position $\tilde{V}$, the environment chooses values for the variables in Γ and simultaneously the system chooses values for variables in $W \setminus \Gamma$. Let $\tilde{W}$ be the resulting valuation of W . The new position is computed as the unique $\tilde{V}'$ satisfying $R(\tilde{V}, \tilde{W}, \tilde{V}')$. The goal of the environment is to reach a state that satisfies $T(V)$ in spite of the system's opposition.

A (memoryless) strategy for the environment is a function that maps each state of Ω to one valuation of the variables in Γ . Likewise, a strategy for the system is a function that maps each state of Ω to one valuation of the variables in $W \setminus \Gamma$. A position $\tilde{V}$ is a winning position for the environment if there exists an environment strategy such that, for all system strategies, $T(V)$ is eventually satisfied. A position $\tilde{V}$ is a winning strategy for the system if $T(V)$ is never satisfied.

In Sect. 4 we address the automatic identification of variables that may be profitably added to a given (possibly empty) Γ . In the remainder of this section we assume that the controlling variables are given, together with the open system and the (failing) property.

We are interested in showing whether a hostile environment, controlling only variables in Γ , can force the system into a failing run to enhance the information provided by the error trace. Obviously, if Ω is deterministic and $\Gamma = W$, the environment can force the system along any error trace. This is not very interesting as it provides no further illumination on the error when compared to the traditional error trace. Conversely, if $\Gamma = \emptyset$, the environment can force an error only when it becomes inevitable.

The more interesting cases occur when we can use the assumption that the hostile environment can control only some of the input variables, to help explain an error trace. For example, in the case of an asynchronous system under the interleaving model of concurrency, the variables that control the scheduling of the processes are natural candidates for inclusion in Γ . In languages like SMV [McM94], the scheduler variables are easily identified.¹ With languages in which the scheduling must be modeled explicitly, the user can easily identify the relevant variables as Γ . In hardware designs, the system control variables could be a reasonable guess for Γ and the data variables for $W \setminus \Gamma$.

Given an open system Ω , a property φ for which a nonempty set $E(\varphi)$ of error traces exists, and a set of controlling variables $\Gamma(\Omega)$, we can formulate the winning positions of the hostile environment as follows.

Definition 1. *Given a set of states $T(V')$ of Ω , we denote the set of states that the environment can control to T in one step using Γ by*

$$\text{MX}_{\Omega, \Gamma} T = \exists \Gamma. \forall (W \setminus \Gamma). \exists V'. [R(V, W, V') \wedge T(V')] .$$

¹ The scheduler variables have names `<proc>.running` in SMV. They are not input variables, but they can be treated as such.

The dependence on Ω and Γ is usually understood, and hence the subscript to MX omitted. We similarly define MSUT as the set of states that the environment can control to T via paths of arbitrary finite length confined in S . That is,

$$\text{MSUT} = \mu Z. T \vee [S \wedge \text{MX} Z] .$$

MSUT computes exactly those states from which a hostile environment has a winning strategy. The i -th iteration of the fixpoint computation corresponding to MSUT represents the states from which the environment has a winning strategy with i moves.

The operators of Definition 1 are related to the operators of CTL, and indeed are specialized versions of the operators of Alternating-time Temporal Logic [AHK97]. MX differs from EX only in the type and position of quantification of the $W \setminus \Gamma$ variables. The environment and the system are assumed to choose simultaneously the values for the inputs they control. This is reflected in the order in which the input variables are quantified. Since the environment must be able to control the system to some state in T without knowledge of the values of the variables in $W \setminus \Gamma$, the variables in Γ appear in the outermost quantification. On the other hand, the exact state in T to which the system is controlled is immaterial; hence, the next state variables appear in the innermost quantification. MU differs from EU only in the use of MX instead of EX .

The definition of $\text{MX}T$ is illustrated in Fig. 2, where a fragment of a system is shown for which $W = \{g, w\}$, $\Gamma = \{g\}$, $T = \{3, 4\}$, and $\text{MX}T = \{1, 2\}$. Transitions out of States 0, 3, 4, and 7 are not shown to avoid clutter; they are all to states not in T . The

Fig. 2. Illustration of the definition of $\text{MX}T$

environment can force State 2 to State 4 by choosing $g = 0$. Conversely, State 6 cannot be forced to states in T because the system can control State 6 to State 7 by choosing $w = 0$. State 1 can be controlled to a state in T by choosing $g = 1$. (The exact state reached depends on the choice of the system.) However, the environment does not have a choice for the value of g that will work from State 5 regardless of the value of w .

3.2 General Fate and Free Will Algorithm

For a given choice of Γ , the environment may be able to force the system along parts of the error trace, but not the entire trace. For example, in hardware designs where Γ maps to the control variables and $W \setminus \Gamma$ maps to the data variables, a specific data value may be important at some point to trigger the failure, but for the rest, appropriate choices of control variables could force the failure to happen. The choice of the data values can

be likened to acts of volition of the system (free will), whereas progress toward the bad states under the control of the environment are akin to the accomplishment of Fate.

In terms of games, the environment may not have a winning strategy from the initial states; the system may have to collaborate at some points along the trace by making the “wrong” choices (mistakes). Consequently, we can formulate the selection of an error trace in terms of a sequence of games between the environment and the system. Each game is an instance of the one described in Section 3.1, generating the fated segments. The sequence of games are interleaved with the (free-will) mistakes of the system. (Henceforth, mistake, free choice and free segment will be used interchangeably.)

Our objective is to find an error trace $e \in E(\varphi)$ that consists of few, short segments. Such an error trace can be generated by answering the following question.

Question 1. What are the states from which the environment can force the system to T if the system makes exactly i mistakes?

For $i = 0$, these are states computed by the game in Section 3.1. For $i = 1$, these are states from which the system has a winning strategy (the environment has no winning strategy), but with one mistake, the system reaches states from which the environment can force the system to T . For any $i > 0$, these are states from which the system has a winning strategy, and with one mistake reaches states from which the environment can force the system to T with $i - 1$ mistakes of the system. In other words, for any $i > 0$, the system still has a winning strategy for fewer than i mistakes but unavoidably reaches T if it commits i mistakes. An error trace with few segments can be constructed from the initial state that requires the fewest mistakes to reach the target state T .

To compute these states, we first classify states into those from which the system must progress towards the error and states from which the system can avoid reaching the error. We consider a general partition that divides the state space into *layers*, $\{S_i\}$. Each layer S_i has two types of states—boundary states T_i and interior states $S_i \setminus T_i$. From the boundary states, the system can avoid reaching the error and has to make a free choice at these states to move out of the layer. At the interior states, the system can be forced by the environment towards the boundary states of the corresponding layer, and consequently towards the error. With this definition a trace will be annotated by a fated segment (perhaps empty) within each layer, and a free choice at the boundary states (precisely those at which the system can make mistakes).

Our layering scheme starts with the target as the initial set of boundary states and forms concentric layers around the target. The i -th concentric layer contains states that must make i mistakes to reach the target. A trace moving through these layers with decreasing index eventually reaches the target. At the boundary states of each layer, the system can make a free choice into states in the previous layer (closer to the target). The interior states for each boundary is then computed to complete the layer.

Each layer is closed, that is, all states outside of previously computed layers at which the system can be forced to the current boundary are in the current layer. Hence, once a layer is computed, a new set of boundary states is determined. This ensures that fated and free segments alternate in our algorithms.

Consider the state graph in Fig. 3, State 1 is the initial state and State 5 is the target. Suppose $W = \{w\}$ and $\Gamma = \emptyset$ for this graph. The layer set A is an example of the layers defined above. The layers are ordered by increasing index. The index indicates

Fig. 3. Example showing set of layers A

the number of wrong choices that the system has to make to progress towards T . For instance, State 1 is forced to reach either State 3 or 4 (boundary states T_1), therefore it is in S_1 . States 3 and 4 can make a free choice to go to 5.

Fig. 4 implements this approach to computing the layers $\{S_i\}$. Its arguments are the target states T and the states that reach T and are reachable from the initial states, H . A list L is instantiated to store the layers. In each iteration of the loop, the boundary states are computed as T_i in Line 9 and the layer is completed as S_i in Line 5. Since the layers start at T , one of the concentric layers is formed in each iteration. Previously computed states are removed from T_i . The while-loop stops when there are no more states.

```

1  FATEANDFREEWILLLAYERS ( $T, H$ ) {
2     $L := ()$ ;
3     $i := 0; T_0 = T$ ;
4    while ( $H \neq \emptyset$ ) {
5       $S_i := M H \cup T_i$ ;
6       $L := \text{CONCATENATE}(S_i, L)$ ;
7       $H := H \setminus S_i$ ;
8       $i := i + 1$ ;
9       $T_i := (\text{EX } S_i) \cap H$ ; }
10   return  $L$ ; }
```

Fig. 4. General Fate and Free Will algorithm

Some observations on the layers computed in Fig. 4 are listed below.

Observation 1. For each i, j , $S_i \cap S_j = \emptyset$.

Consider the sequence of onion-rings R around T restricted to the set H . The rings are computed using a backward search with $R_0 = T$. and form a partition of H .

Lemma 1. For each i , $R_i \subseteq \bigcup \{S_j \mid j \leq i\}$.

By the construction of the layers, a trace passing through S_i has to make a free choice out of T_i . Due to Lemma 1, no free choices are possible from S_i to S_j for $j < i - 1$.

Observation 2. *The lower bound on the number of free choices from a state in S_i to T is i .*

Theorem 1. *The algorithm in Fig. 4 terminates.*

Corollary 1. *By Theorem 1 and Observation 1, the set of layers $\{S_i\}$ form a partition of the states in H .*

Theorem 2. *It is possible to construct a trace from a state in S_n to T by making exactly one free choice out of every layer S_i to layer S_{i-1} for $i \leq n$.*

The algorithm in Fig. 4 requires linear (in the number of states) MX and EX steps but is more expensive than generating the traditional trace. MX computations may be more expensive than EX computations when using BDDs (in terms of BDD sizes) due to the difference in quantification of input variables. The I variables cannot be quantified before $W \setminus I$ and the $W \setminus I$ variables cannot be quantified before V' since existential and universal quantifications do not commute. This may cause several BDD variables to be alive over most of the MX computation, which in turn may cause the BDDs to be very large. The extra computational burden may affect performance of this algorithm.

```

1  COUNTEREXAMPLE ( $S, T, L, R$ ) { //  $R$  is the set of rings around  $T$ 
2    if ( $T \cap S \neq \emptyset$ ) return (PICKSTATE( $T \cap S$ ));
3     $L_p :=$  layer  $L_i$  of least index that intersects  $S$ ;
4     $Q := S \cap L_p$ ;
5     $p := \text{INDEX}(L_p)$ ;  $\pi := ()$ ;
6    do {
7       $R_q :=$  ring  $R_i$  of least index that intersects  $Q$ ;
8       $P := \text{PICKSTATE}(Q \cap R_q)$ ;
9       $\pi := \text{APPEND}(\pi, P)$ ;
10      $Q := \text{EY}(P)$ ;
11     if ( $p > 0$  and  $Q \cap L_{p-1} \neq \emptyset$ ) {
12        $Q := Q \cap L_{p-1}$ ; // make a free choice out of  $L_p$ 
13       mark  $P$ ;
14        $p := p - 1$ ; }
15   } while ( $P \cap T \neq \emptyset$ )
16   return  $\pi$ ; }
```

Fig. 5. Counterexample algorithm based on the General Fate and Free Will algorithm in Fig. 4

Based on Theorem 2, we propose an algorithm to construct a trace between S and T given the sequence of layers L computed by Fig. 4. Fig. 5 shows a trace constructed from the layers in L using the fewest free choices. The algorithm takes the following arguments: S initial states, T target states, L sequence of layers and R rings around

T . (States in ring R_i have a path of length i to T .) The algorithm starts with the initial state in the least layer (closest to T) in Line 3 and builds an error trace in a sequence of alternating fated and free segments down the layers in L .

The while-loop terminates when the trace includes a target state. A free choice is made at the earliest opportunity to progress towards T (Line 12). Each state at which the system makes a free choice is marked to annotate the error presented to the user. The rings around T are used to minimize the length of the trace to T (Lines 7–8).

Theorem 2 ensures the correctness of this algorithm. The cost of generating such a trace is comparable to the cost of the traditional method to generating error traces.

A trace generated by Fig. 5 is explained to the user as follows: For each fated segment, from the source state, the system is bound to reach a target set, one state of which is presented as the destination of the fated segment. Each free choice is a step taken by the system towards the error. It would have been possible at these states to avoid reaching the error.

Theorem 3. *The trace produced by the algorithm in Fig. 5 contains the least free segments (markings).*

While algorithm in Fig. 5 computes a trace with fewest free segments, it does not guarantee a counterexample of minimum length.

3.3 Restricted Algorithm

The algorithms of Fig. 4 and Fig. 5 provide a set of layers and a counterexample annotated with the fated and free segments. The general algorithm has the disadvantage of being expensive to compute. Also, since only one state is represented from the boundary, it may be hard to comprehend the fated segment. However, the layers are too large to present to the user. In this section, we present a restricted Fate and Free Will algorithm that has two important advantages—it is less expensive than the general algorithm and the error trace is more self-contained in the information it provides.

This main restriction is that the computed layers in this algorithm are not complete with respect to H . This is achieved by setting the boundary to one state of the T_i computed in Line 9 of Fig. 4 instead of the entire T_i . This modification is likely to make the MU computation less expensive due to a smaller T_i . Additionally, now the error trace presents a complete picture in the fated segment since the fated segment is exactly to the state presented in the error trace.

The restricted algorithm is described in Fig. 6. The input to the algorithm is a set of initial states, S , a set of target states, T , such that $S \cap T = \emptyset$, and a set H of states such that the paths from S to T are confined within H . We can assume that H consists of states reachable from S , and that can reach T . In fact, we only need to compute the states reachable from S , because backward search from T never touches states that cannot reach T . We assume that H is divided into *onion rings* $\{h_i\}$ according to the distance of a state from S .

The output of the FATEANDFREEWILL algorithm is a counterexample C in the form of a list of segments. The segments are constructed backwards from T . The last segment to T is always fated. Fated and free segments alternate, though the fated segments may be empty. A fated segment computation always follows a free segment computation to

```

1  FATEANDFREEWILL ( $S, T, (h_0, \dots, h_m)$ ) {
2     $prev := free; p = m;$ 
3    if ( $T \cap S \neq \emptyset$ ) return ((PICKSTATE( $T \cap S$ )));
4     $h_p :=$  ring  $h_i$  of least index that intersects  $T$ ; // closest to S
5     $T :=$  CLOSESTSTATE( $T \cap h_p, S$ );
6     $C := ((T));$  // initialize counterexample trace,

7    // find segments from violation to the initial states
8    while ( $T \cap S \neq \emptyset$ ) { // done, if current target intersects initial states

9      // alternate fated and free segments
10     if ( $prev = free$ ) {
11        $prev := fated;$  // flip the flag
12        $H := h_0 \cup h_1 \cup \dots \cup h_p;$ 
13        $S_1 := M H U T;$  // compute states that can be forced to T
14       if ( $S_1 = T$ ) continue; // failed to find non-trivial fated segment,
                               // now find the free segment
15       // non-trivial fated segment exists!
16       // compute onion-rings to trace shortest path for fated segment
17        $h_p :=$  ring  $h_i$  of least index that intersects  $S_1$ ; // closest to S
18        $R := (\rho_0, \dots, \rho_q) :=$  RINGS( $S_1, h_p \cap S_1, T$ );
                               // rings of  $E S_1 S (h_p \cap S_1)$  up to  $T$ 
19     } else { // set the variables for free segment construction
20        $prev := free;$  // flip the flag
21        $h_p :=$  ring  $h_i$  of least index that intersects  $EX T$ ;
22        $R := (h_p \cap EX T, T);$  // rings for free segment
23     }

24     // construct current segment, add it to counterexample
25      $\sigma :=$  BUILDSHORTESTPATH(LAST( $R$ ),  $T, R$ );
26      $C :=$  CONCATENATE( $(\sigma), C$ );

27      $T :=$  FIRST( $\sigma$ ); // destination of next segment
28   }
29   return ( $C$ );
30 }

```

Fig. 6. Fate and Free Will algorithm

Fig. 7. The rings built during the computation of $M H U T$ do not guarantee shortest paths. State 3 is added to the fixpoint only after two iterations, even though it has a path of length 1 to a state in T . All states are in H . Variable w is controlled by the system

greedily minimize the number of free segments in C . Two consecutive fated segment computations are avoided since the MU computations are closed.

The procedure starts by picking a state in T closest to S in Line 5. The function `CLOSESTSTATE` tries to pick a state from a set that is closest to a given state in terms of the Hamming distance of the encodings. C is initialized to T . Each iteration of the while-loop computes one segment, either fated or free. The alternation is imposed by the *prev* flag that records whether the current computation is fated or free, and if-else statement (Lines 10–23) performs the other based on the previous value of *prev*. The first segment is set to a fated computation, by setting *prev* initially to *free*.

For a fated computation (Lines 10–18, Lines 25–26), Line 13 computes the states in H that can be forced to the current target T . If the computation was trivial (only states in T result), then the algorithm goes back to the beginning of the while-loop to try free segment computation. Notice that the state sets of the iterations of $MHUT$ do not provide a means to trace shortest paths in general. (See Fig. 7.) Hence if the MU computation is non-trivial, onion rings are set up (Lines 17–18) from the state closest to the initial state in the fated set in order to be able to build one short fated segment. For a free segment computation (Lines 19–23, Lines 25–26), the states that can make a free choice into T and the onion rings for this choice are set up in Lines 21–22. The distance from the initial state is guaranteed to decrease since the index p for the onion rings decreases. This is critical for the termination of the algorithm as it ensures that fewer rings are considered every time a free computation is performed.

Lines 25–26 build the actual free or fated segment to be added to the counterexample using the onion rings R . The procedure `BUILDSHORTESTPATH` traces a minimum-length path from a state in $\text{LAST}(R)$ to a state in T using the onion rings R around T as described in Sect. 2. The rings aid in building short segments. Finally, in Line 27, the target T for the next segment is set to the first state in the counterexample constructed thus far.

This algorithm is not guaranteed to produce a counterexample trace of minimum length or least free segments.

Example 2. We have implemented the algorithms in Fig. 6. The counterexample in Fig. 1 for Example 1 is generated using the restricted algorithm in Fig. 6. In this example, T is set to `validin` since it is a control input. The counterexample is a liveness property and is generated in parts. Each part is generated with the algorithm in Fig. 6. The first part is the stem that shows a path from the initial state to a state such that there is a match, more precisely to a state that satisfies the formula $(\text{match} \wedge \text{EG}(\text{sav} = \text{hh}))$. The next part builds a cycle around the state with the match such that the HI request saved in *sav* never gets issued in the cycle.

In the first part, the state in Column 4 is picked as the target. This state indicates a match, with the LO FIFO containing the matched entry. The stem is built backward from the initial state. The HI FIFO gets two requests for address 0 (in Columns 2 and 3, but only one is required) and the LO FIFO gets a request for address 0 causing the match.

In the next part, a cycle is traced through State 4 by setting State 4 as the target T and a successor of State 4 (State 5) as the initial state S . Since the cycle is traced backward, the first fated segment is indicated in Columns 9–15–0. A free choice is made into State 9 with *inaddr* = 0 and *h1* = 0. The other fated segments of the cycle (Columns 5–8 and Columns 2–4) are constructed similarly.

The counterexample is read left to right. The free choices allow LO requests to enter the LO FIFO with `inaddr = 0`. The fated segments inevitably issue these requests as the `slcClk` ticks. The free choices ensure starvation of the HI request corresponding to the entry in `sav`. This raises the question as whether these free choices are indeed allowed. Avoiding starvation may lie in the answer to this question.

4 The Consequences of Progress

Various choices of controlling variables are possible for a given system. These choices may reflect the control-data dichotomy, or the distinction between external and internal nondeterminism. In this section we discuss an algorithmic criterion for the selection of controlling variables that is based on the notion of *progress* towards the bad states.

Example 3. Consider a system with a master reset input r . Assertion of r causes the system to make a transition to a specified “home” state. Unless the home state is a bad state for the property being verified, r should remain deasserted throughout the error trace, and, unless r is included in Γ , no fated segments can be found.

In cases like that of Example 3, we would like to automatically detect variables playing a role similar to r , and assign them to Γ . We observe that a trace from some initial states must eventually cross the boundaries between adjacent rings built around the target. If we impose the requirement that at each step some progress be made towards the target, we define a set of transitions that can be used by the error traces. Let $P(V, W, V')$ be the characteristic function of the transitions that leave the current endpoint of the trace and cross the boundary into the next onion ring; let $w \in W$ be an input variable. If $P_w = 0$ or $P_{\neg w} = 0$, the requirement of progress dictates the value of w . We add w to Γ if it is often critical to making progress.

Extending this principle, if a variable w is such that $\forall w. P = P_w \wedge P_{\neg w} = 0$, we consider it for inclusion in Γ . If $\forall w. P = 0$, variable w divides the transitions into disjoint sets. In adding a few such variables to Γ , one hopes that a succinct explanation of the error trace will emerge.

Example 4. Analysis of the progress requirements for the system of Example 1 reveals that sometime input `validin` must have a prescribed value for the system to evolve towards the bad states. In addition, it is always the case that the universal quantification of `validin` from P yields 0. Hence, we choose $\Gamma = \{\text{validin}\}$. This choice leads to the counterexample of Fig. 1.

5 Related Work

Not much has been written about the explanation of counterexamples produced by model checkers. The works closest to ours are [dAHM00] and [AHK97], though the points of contact are more in the formulation than in the purpose. Specifically, the game-theoretical formulation we have adopted for the computation of fated segments is related to the approach of [dAHM00]. However, we are interested in explaining counterexamples, as

opposed to solving the model checking problem. This leads to a reversal of the roles between environment and system. Another distinguishing aspect is that we may generate multiple fated segments along a trace, and the last of these segments may not reach the target (bad) states. Finally, in our context we can explore scenarios defined by different choices of controlling variables I .

The work on vacuity detection [BBER97,KV99] deals with the notion of *interesting witness* to a passing formula. A witness is interesting for φ if it is a witness to all (important) subformulae of φ . Since producing an interesting witness to φ amounts to finding a counterexample for a formula derived from φ , our techniques can be directly applied to the derivation of interesting witnesses that are easier to understand.

6 Experiments

We have implemented the algorithm in Fig. 6 in the model checking tool VIS. The generation of the annotated trace in Fig. 1 took 2.1 seconds on an IBM IntelliStation with a 400 MHz Pentium 2 CPU and 1 GB of RAM running Linux. For comparison, the standard counterexample generation algorithm took 5.7 seconds. We applied this algorithm to two other circuits. The first is a Digital Audio Input/Output Receiver circuit. The property tested failed and the counterexample produced one forced segment of length 31 in 4 seconds as compared to 1.9 seconds in the traditional trace computation. The third example is a buggy implementation of the bakery protocol. The mutual exclusion property failed, producing a counterexample with four fated segments of lengths 13, 6, 4, and 5 respectively. Generating the counterexample took 1.2 seconds, versus 0.1 seconds with the traditional approach.

7 Conclusions

We have discussed the explanation of counterexamples produced by model checkers. The separation into segments and the pinpointing that some sequences are fated while others are free all contribute to explaining the error. We have presented an example with a trace demarcated into fated and free segment to explain our method. The segments help analyze the failure better. We have also presented experimental results for other circuits.

Many extensions and variants of the techniques we have proposed remain to be explored. One direction of interest is the formulation of liveness counterexamples as Büchi games. We are also researching counterexamples for existential properties.

Another area of investigation is to extend an error trace to a bundle of error traces while keeping them readable. We are exploring the idea of extending each state to a set of states represented as a cube in the counterexample. The motivation for this is to reduce the number of relevant variables appearing in the counterexample so it is easier to debug. Towards the same goal is the detection of dependent variables so as to isolate a small basis for variables in the error traces. Also, in case of hierarchical descriptions, we would like to restrict the traces to variables that appear in as few modules as possible.

References

- [AHK97] R. Alur, T. A. Henzinger, and O. Kupferman. Alternating-time temporal logic. In *Proceedings of the IEEE Symposium on the Foundations of Computer Science*, pages 100–109, 1997.
- [BBER97] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. Efficient detection of vacuity in ACTL formulas. In O. Grumberg, editor, *Ninth Conference on Computer Aided Verification (CAV'97)*, pages 279–290. Springer-Verlag, Berlin, 1997. LNCS 1254.
- [CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Proceedings Workshop on Logics of Programs*, pages 52–71, Berlin, 1981. Springer-Verlag. LNCS 131.
- [CGMZ95] E. Clarke, O. Grumberg, K. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Proceedings of the Design Automation Conference*, pages 427–432, San Francisco, CA, June 1995.
- [CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
- [dAHM00] L. de Alfaro, T. A. Henzinger, and F. Y. C. Mang. Detecting errors before reaching them. In E. A. Emerson and A. P. Sistla, editors, *Twelfth Conference on Computer Aided Verification (CAV'00)*, pages 186–201. Springer-Verlag, Berlin, July 2000. LNCS 1855.
- [EH86] E. A. Emerson and J. Y. Halpern. ‘Sometimes’ and ‘not never’ revisited: On branching time versus linear time temporal logic. *Journal of the Association for Computing Machinery*, 33(1):151–178, 1986.
- [EJ91] E. A. Emerson and C. S. Jutla. Tree automata, mu-calculus and determinacy. In *Proc. 32nd IEEE Symposium on Foundations of Computer Science*, pages 368–377, October 1991.
- [Kur94] R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.
- [KV99] O. Kupferman and M. Y. Vardi. Vacuity detection in temporal model checking. In *Correct Hardware Design and Verification Methods (CHARME'99)*, pages 82–96, Berlin, September 1999. Springer-Verlag. LNCS 1703.
- [LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specification. In *Proceedings of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 97–107, New Orleans, January 1985.
- [McM94] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, MA, 1994.
- [RBS00] K. Ravi, R. Bloem, and F. Somenzi. A comparative study of symbolic algorithms for the computation of fair cycles. In W. A. Hunt, Jr. and S. D. Johnson, editors, *Formal Methods in Computer Aided Design*, pages 143–160. Springer-Verlag, November 2000. LNCS 1954.
- [WVS83] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths. In *Proceedings of the 24th IEEE Symposium on Foundations of Computer Science*, pages 185–194, 1983.