Efficient Debugging in a Formal Verification Environment

Fady Copty, Amitai Irron, Osnat Weissberg, Nathan Kropp*, and Gila Kamhi

Logic and Validation Technology, Intel Corporation, Haifa, Israel
Microprocessor Group*, Intel Corporation, USA
gila.kamhi@intel.com

Abstract. In this paper, we emphasize the importance of efficient debugging in formal verification and present capabilities that we have developed in order to augment debugging in Intel's Formal Verification Environment. We have given the name the *"counter-example wizard"* to the bundle of capabilities that we have developed to address the needs of the verification engineer in context of counter-example diagnosis and rectification. The novel features of the counter-example wizard are the *"multi-value counter-example annotation," "multiple root cause detection,"* and *"constraint-based debugging"* mechanisms. Our experience with the verification of real-life Intel designs shows that these capabilities complement one another and can considerably help the verification engineer diagnose and fix a reported failure. We use real-life verification cases to illustrate how our system solution can significantly reduce the time spent in the loop of model checking, specification and design modification.

1 Introduction

Verification is increasingly becoming the bottleneck in the design flow of electronic systems. Simulation of designs is very expensive in terms of time, and exhaustive simulation is virtually impossible. As a result, designers have turned to formal methods for verification.

Formal verification guarantees full coverage of the entire state space of the design under test, thus providing high confidence in its correctness. The more automated and therefore the more popular formal verification technique is *symbolic model checking* [2]. While gaining success as a valuable method for verifying commercial sequential designs, it is still limited with respect to the size of the verifiable designs.

The capacity problem manifests itself in an additional obstacle—low productivity. A lot of effort is spent decomposing the proofs into simpler proof obligations on the modules of the design. A global property to be verified is usually deduced from local properties verified on the manually created abstraction [10] of the environment for each module. The abstractions and assumptions needed to get a verification case through increase the chance of getting false failure reports. Further

T. Margaria and T. Melham (Eds.): CHARME 2001, LNCS 2144, pp. 275-292, 2001.

more, in case of valid failure reports, back-tracing to the root cause is especially difficult.

Given the inherent capacity and productivity limitation of symbolic model checking, we emphasize in this paper the importance of efficient debugging capabilities in formal verification, and we present capabilities that we have developed in order to augment debugging in a commercial formal verification setting. Our solution provides debugging capabilities needed at three major stages of verification.

- **Specification Debugging Stage.** In the early stages of verification, most of the bugs found are not real design errors, but rather holes in the specification. In this stage, which we call *specification debugging* [4,5], the verifier is in the loop of model checking and specification modification based on the feedback from the symbolic model checker. The turn-around time of the model checker becomes then very critical to the productivity of the verification engineer.
- **Model Debugging Stage.** In order to find the root cause of a bug reported by a formal verification system, one needs intimate knowledge of the design behavior. In order to check whether a bug is spurious or not, one must understand in detail the effect of pruning and environmental assumptions made to verify the design under test. Additionally, once a fix has been made as a result of a *counter-example* report, one must ensure that the previously reported counter-example does not hold any more in the fixed design.
- **Quality Assurance Stage.** A significant problem in industrial-size projects is ensuring that the process of fixing one design error does not introduce another one. In the context of conventional testing this is checked through regression testing[7]. If consecutive test suites check several properties, a failure in one property may require re-testing all the previous suites, once the failure has been rectified. Efficient regression testing clearly requires techniques useful for debugging.

In the case of a failing verification, the output of a formal verification system is a counter-example—a trace illustrating that the design under test does not satisfy a given property. It is especially difficult to diagnose a verification failure reported as a counter-example. On one hand, the verification engineer suffers from too much data and the difficulty to distinguish the relevant data (i.e., the signal values that cause the failure). On the other hand, he has too little information. Rectification of the error displayed by a single counter-example trace does not guarantee the overall correction of the failure. A trace is just one of the many witnesses that demonstrate that the model does not satisfy the given property. In that sense, a counter-example has too little information. Trace analysis is even more difficult in formal verification, where obscure corner cases can produce complex failures and consequently complex counter-examples to debug.

We have given the name *"counter-example wizard*" to the bundle of capabilities that we have developed to address the needs of the verification engineer in context of counter-example diagnosis and rectification during the above stages of verification. The novel features of the counter-example wizard are the "*multi-value counter-example annotation," "multiple root cause detection,"* and *"constraint-based debugging"* mechanisms. Our experience with the verification of real-life Intel designs

shows that these capabilities complement one another and can considerably help the verification engineer diagnose and fix a reported failure.

The "multi-value" nature of the counter-example annotation mechanism enables the concise reporting of all the failures (i.e., counter-examples) as a result of one model checking run. Hence, the understanding of more than one root cause of a failing verification facilitates the rectification of the failure. The ability to fix more than one root cause can reduce the number of model checking runs needed to get to a passing verification run. Most importantly, multi-value counter-example reports enable the user to pinpoint the pertinent signal values causing the failure and aid in detecting how to change the values to correct the failure.

"Constraint-based debugging" allows the verification engineer to restrict the set of failures (i.e., counter-example traces) to only those that satisfy a specific sequential constraint. If this subset is empty for a given sequential constraint, this means that the constraint is sufficient to eliminate all counter examples found so far. However, the model checker must still be run again to find out if the constraint is sufficient to resolve all counter-examples of all lengths.

The system solution that we provide reduces the time spent in the loop of model checking, specification and design modification. The usage flow consists of running the model checker, dumping all the model checking data needed to compute all the counter-examples of a given length, and then debugging in an interactive environment by loading the pre-dumped model checking data. The fact that we have taken the model checker out of the "check-analyze-fix" loop reduces the debugging loop to "analyze-fix" and consequently improves the time spent in debugging considerably. The effective usage of secondary memory allows the verification engineer to post-process model checking data and debug without the need to add the model checking run to the verification loop.

This paper is organized as follows. In Sect. 2, we present an overview of the formal verification system with enhanced debugging capabilities. Section 3 depicts in detail the capabilities of the "counter-example wizard" Section 4 explains the algorithms underlying our system solution. In Sect. 5, we illustrate the efficiency of these techniques through verification case studies on Intel's real-life designs. We summarize our conclusions in Sect. 6.

2 System Overview

The formal verification system with the counter-example wizard consists of three major components:

1. A state-of-the-art symbolic model checker which accepts an LTL-based formal specification language
2. An interactive formal verification environment which enables access to all the model checking facilities
3. A graphical user interface which allows the user to display annotated counter-example traces and access interactive model checking capabilities

The usage flow consists of two major stages:

1. **Model Checking.** The model checker is run with the option to dump the relevant model checking data and counter-example information to secondary memory.
2. **Interactive Debugging.** The user loads and interacts with the model checking data to access different counter-examples and perform "what-if analysis" on the existence of counter-examples under specific conditions.

The easy storage and loading of relevant model checking data is due to the "*data persistency mechanism*" of the model checker. At any point of the model checking run, the data persistency mechanism can represent all the relevant, computed model checking information in a well-defined ASCII format which later can be loaded in an interactive model checking environment and analyzed through the usage of a functional language.

The fact that the model checker can dump the relevant information for debugging at any point, enables easy integration of this mechanism into regression testing. When regression test suites are run with the "counter-example data dump" facility enabled, the analysis of the failing verification test cases can be done without rerunning the model checker and regenerating the failing traces, which can be computationally expensive. The computational benefit of the system is also witnessed in the specification , model verification, and modification loop.

3 Counter-Example Wizard

Traditional symbolic model checkers provide a single counter-example as the output of a failing verification. To diagnose a verification failure reported as a counter-example is difficult. On one hand, we suffer from too much data and the difficulty identifying the relevant data. On the other hand, we often do not have sufficient information to find the root cause of the failure and rectify it. In this section, we present the "counter-example wizard" that addresses the counter-example analysis and rectification problem and has three novel capabilities: multi-value counter-example annotation, multiple root cause detection, and constraint-based debugging.

3.1 Multi-value Counter-Example Annotation

We address the counter-example diagnosis problem by introducing a concise and intuitive counter-example data representation. We call this advanced representation "multi-value counter-example annotation." This novel annotation relies on the exhaustive nature of the symbolic model checking and hence the ability to represent all the counter-examples of a given length[1]. The enhanced counter-example reporting classifies signal values at a specific phase of a counter-example trace into three types:

[1] The model checker generates counter-examples of the shortest path length. The underlying symbolic model checking algorithms that enable "multi-value counter-example annotation" will be explained in detail in Section 4.

- **Strong 0/1** indicates that in all possible counter-examples that demonstrate the failure, the value of the signal at the given phase of the trace is 0 or 1, respectively.
- **Weak 0/1** indicates that although the value of the signal at the given phase is 0 or 1 respectively for this counter-example, the value of the signal at this phase can be different for another counter-example illustrating the failure. Even though the model checker has some leeway in the choice of a value for this signal, this signal must preserve some relation with other signals at this phase.
- **Weaker 0/1** is similar to "weak" designation, except that weaker values are basically arbitrary, and have little or no influence on the generation of a failure.

The strong values provide the most insight on the pertinent signals causing a failure. For example, if the value of a signal at a certain phase of a counter-example is a strong zero, this means correcting the design so that the value of the signal will be one at that phase will often correct the failure. Hence, the error rectification problem is often reduced to determining how to cause a strong-valued signal to take on a different value.

The counter-example wizard can make use of a waveform display to represent the multi-value counter-example annotation. Figure 2 illustrates a screen shot of the multi-value annotated counter-example graphical display. (Later we will show the use of a text-based display.) The counter-example in the figure demonstrates the violation of the specification *"if X is high, W will be high a cycle later."* The value of Y is clearly not relevant (i.e., weaker); therefore its waveform is shadowed out. Furthermore, the values of X and Z in the second cycle do not affect the failure, so their waveforms in that cycle are shadowed out as well. Examination of the waveform reveals that the cause for the failure is the value of the signal Z, which causes the output of the AND gate to be low.

Our experience shows that strong values alone sometimes provide sufficient information to figure out the root cause of a failure and speed up the debugging. Nevertheless, we have also witnessed many verification cases where the answer to the root cause of failure lay in the weak values (as seen in Fig. 3). The debug of traces with weak values is facilitated by the sequential constraint-based debugging capability which is the second major feature of the counter-example wizard.

3.2 Constraint-Based Debugging

"Sequential constraint-based counter-examples" are traces displaying the failure while obeying some temporal constraint. A sequential constraint in the debugging wizard is described as an associated list of pairs of a Boolean condition and a corresponding phase in which the Boolean condition must hold. In other words, the user specifies a function over the signals in the design and a point in time when the function should hold. The counter-example wizard then looks for traces that satisfy the constraint, and recalculates all weak and strong signal values relative to this subset of traces.

Constraint-based debugging facilitates the rectification and the diagnosis of the root cause of a failure. If no counter-example that holds the constraint is found, then the constraint describes a condition sufficient to make the erroneous design or specifi-

cation correct. Consequently, "sequential constraint-based debugging" can help significantly in the correction of design or specification errors.

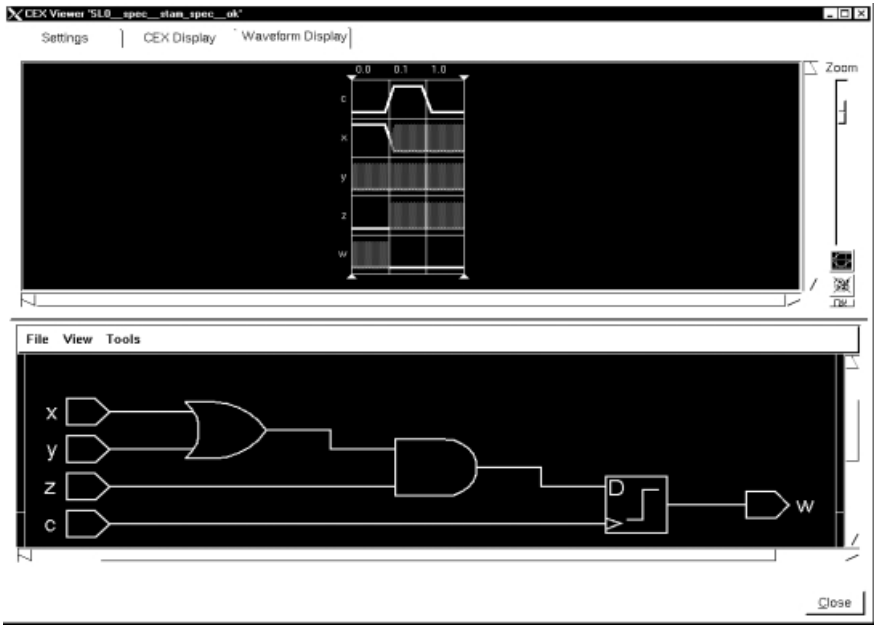

Fig. 2. Graphical counter-example display using multi-value annotation. In the waveform (top), the strong (i.e., significant) signal values are represented by bold lines, while the weaker values are represented by gray boxes

For example, let us assume that some input vector *foo* should be one-hot encoded (exactly one of the bits in the vector is high and the rest are low), but in the counter-example presented it is not encoded as one-hot. In the absence of constraint-based debugging, the user would have to add an environmental assumption that *foo* is one-hot and rerun the model checker to see if the erroneous encoding is indeed the root cause of the failure—a task that can take hours. With constraint-based traces the user can write the environmental assumption as a sequential constraint, and if there are no counter-examples that satisfy the constraint, then the user knows that assumption is sufficient to eliminate all current counter-examples. Thus the user is able to check whether an assumption will cure the current failure without rerunning the model checker.

Additionally, constraint-based debugging allows "what-if analysis" and the ability to investigate the relationships between signals. For example, setting the value of a signal to a constant value at a specific phase and observing other signal values that have consequently become strong, helps the user to understand the relationships between signals over time. Thus, constraint-based debugging refines the information that weak values provide.

Figure 3 illustrates how the usage of two different constraints help debug a failing verification task. The task is to check that the model illustrated in the lower half of the figure satisfies the specification "*if Z is high, W will be high a cycle later.*" On the upper half of the figure, three multi-value annotated counter-example traces are illustrated. The leftmost trace shows all the counter-examples of length three violating this specification. Viewing this trace, we observe that only W, Z and the clock C get strong values in the annotated trace. The signals X and Y have weak values for the first phase and weaker values for all the rest of the phases (indicating that the values of these signals in second and third phases are irrelevant to the failure). In this case, the strong values do not provide enough information; therefore we analyze the weak values (i.e., the relationship between X and Y in the first phase). The middle trace and the rightmost trace demonstrate all the counter-examples of length three, under two constrained values of the signal X in the first phase (one and zero, respectively). When the value of X is high, we can ignore the value of Y. Therefore, the value of Y becomes weaker under this constraint as illustrated in the middle annotated trace. The second constraint, as illustrated by the rightmost trace, assigns X a low value in the first phase. Under this constraint, the signal Y gets a strong one value. Therefore, our conclusion from this constraint-based debugging session is that Y must be high in the first phase to get a violation. Furthermore, to rectify the violation both X and Y need to get a low value in the first phase.

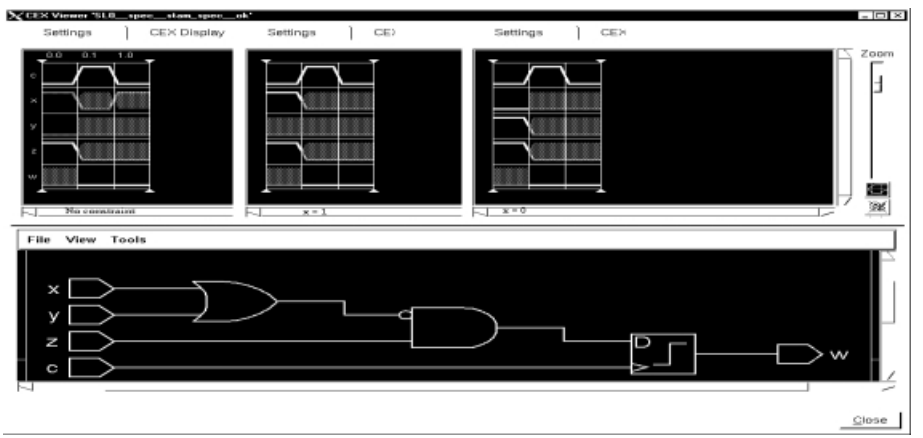

Fig. 3. An illustration of the usage of constraint based debugging. The significant (i.e., strong) signal values are highlighted. The weak values are shadowed out, and weaker values are displayed as gray boxes

3.3 Multiple Root Cause Detection

The user can request a predetermined number of interesting traces that may give clues to several root causes of a failure. The underlying algorithm chooses traces that are unique, and do not share the same prefix and usually share the same postfix. The selection uses a heuristic that selects states to be as different from one another as possible. The purpose is to maximize the parity between states returned by the selection procedure. Usually the design under test has some initialization sequence; therefore, when generating a set of interesting traces, traces with the same prefix but different postfix are chosen. The intuition is that traces with different postfix will give hints on more root causes for the violation.

This mechanism works naturally under the constraint-based debugging mechanism. By request of the user all the interesting traces that comply with a given constraint are produced.

4 Underlying Algorithms

A finite state machine is an abstract model describing the behavior of a sequential circuit. A completely specified FSM M is a 5-tuple $M = (I,S,\delta,\lambda, S_o)$, where I is the input alphabet, S is the state space, δ is the transition relation contained in S x I x S, and $S_o \subseteq S$ is the initial state set. BDDs [2] are used to represent and manipulate functions and state sets, by means of their *characteristic functions*. In the rest of paper, we make no distinction between BDDs and a set of states.

4.1 Background

A common verification problem for hardware designs is to determine if every state reachable from a designated set of initial states lies within a specified set of "good states" (referred to as the *invariant*). This problem is variously known as *invariant verification*,[2] or *assertion checking* [4,5,6].

According to the direction of the traversal, invariant checking can be based on either *forward* or *backward analysis*. Given an invariant G and an initial set of states A, *forward analysis* starts with the BDD for A, and uses BDD functions to iterate up to a fixed point, which is the set of states reachable from A, using the *Img* operator. Similarly, in *backward analysis,* the *PreImg* operator is iteratively applied to compute all states from which it is possible to reach the complement of the invariant. The BDDs encountered at each iteration are commonly referred as *frontiers.*

The *counter-example generation algorithm* [8] is a combination of backward and forward analysis. In Figure 4, we see that counter-example generation starts with the BDD for all the states that do not satisfy the invariant (i.e., F_o) and the *PreImg* operator is applied until a fixed point or a non-empty intersection (i.e, $F_{i+1} \cap S_o \neq \emptyset$) of the last backward frontier with the initial states is reached. A counter-example is then constructed by

[2] Although the "debugging wizard" is a valid tool both for invariant and *liveness* property verification, for the sake of simplicity, in this section we will explain the underlying algorithms of the debugging wizard in the context of invariant verification.

applying forward image computation to a state selected from the states in the intersection (i.e., all the states from which there is a path to the states that complement the invariant). Again by iteratively intersecting the forward frontiers with the corresponding backward frontiers and choosing a state from each intersection as a representative for the corresponding phase in the counter-example, the counter-example trace is built.

Target frontiers [4] are the sets of states obtained during backward analysis, starting from the *error states* (states that violate the invariant) and iteratively applying the *PreImg* operator till reaching an intersection with the initial states. More precisely, we define the nth *target frontier,* $F_{n,}$ as the set of states from which one can reach an error state in *n* (and no less than *n*) steps.

$$F_0 = \neg\, Invariant \tag{1}$$

$$F_{n+1} = \mathrm{PreImg}(F_n) - \bigcup_{i=1}^{n} F_i$$

In what follows, we denote by *N* the index of the last target frontier before the fixed-point, such that $Target\ frontier_{N+1} = Target\ frontier_N$. The target frontiers are disjoint, and their union, which we denote as *Target* represents all the states from which a state violating the invariant can be reached:

$$Target = \bigcup_{i=1}^{N} F_i \tag{2}$$

As can be seen in the counter-example generation algorithm depicted on the left-hand side of Figure 4, the frontiers F_o, F_1, …, F_N represent the target frontiers. In order to obtain reachable target frontiers, we need to filter out the states unreachable from $F_N \cap S_0$ in each frontier. The filtering is done by applying forward analysis starting from the states in $F_N \cap S_0$ as seen in the right-hand side of Figure 4. From here on, we will refer to this version of target frontiers as "reachable target frontiers".

The underlying data structure for all the algorithms of the counter-example wizard is the *reachable target frontiers*. Two major characteristics of reachable target frontiers make them very useful for the computations needed for debugging.

- Any trace through the frontiers is guaranteed to be a counter-example.

All possible counter-examples in this verification of length N (when N is the number of frontiers) are included in the frontiers.

Therefore, these frontiers store all the information needed for the querying and extraction of counter-examples in a very concise and flexible way. In our system, the model checker dumps the BDDs representing the reachable target frontiers to secondary memory. Interactive debugging is done by restoring the target frontiers in the interactive environment and querying them through the functional language.

```
COUNTER-EXAMPLE (δ, S_o, p )
i = 0;
Target = F_0 = {pi | pi does not satisfy p};
// F_0 - all the states that do not satisfy p
C = {} // Counter-example initialized, an empty list
While (F_i ≠ ∅) {
  F_i+1 = PRE-IMG(δ,F_i);
  if (F_i+1 ∩ S_0 ≠ ∅) {
    // the property is violated
    F_i+1 = F_i+1 ∩ S_0;
    j = i;
    while (j >= 0) {
      s = SELECT_STATE(F_j+1);
      F_j = Img(δ, s) ∩ F_j; j= j – 1;
      C = PUSH(s,C);
    }
    return REVERSE(C);
    // return a counter-example
  } //end the property is violated
  i = i + 1;
  F_i = F_i - Target;
  Target= Target U F_i ;
} // the property is satisfied
return C; // return an empty list
```

```
TARGET-FRONTIERS(δ, S_o, p )
i = 0;
Target = F_0 = {pi | pi does not satisfy p};
// F_0 - all the states that do not satisfy p
C = {} // Target frontiers initialized, an empty list
while (F_i ≠ ∅) {
  F_i+1 = PRE-IMG(δ,F_i);
  if (F_i+1 ∩ S_0 ≠ ∅) {
   // the property is violated
    F_i+1 = F_i+1 ∩ S_0;
    j = i ;
   while (j >= 0) {

      F_j = Img(δ, F_j+1) ∩ F_j; j= j – 1;
      C = PUSH(F_j,C);
    }
    return REVERSE(C);
    // return Target Frontiers
  } //end the property is violated
  i = i +1;
  F_i = F_i – Target;
  Target = Target U F_i ;
} // the property is satisfied
return C; // return an empty list
```

Fig. 5. Classic counter-example generation and target frontiers calculation algorithms

4.2 Annotating the Counter-Example Values

Annotating the counter-example values becomes rather simple, once we have the "reachable target frontiers" at hand. The value of signal *x* at phase *i* is

- **Strong**, if $F_{i \mid x=0} = 0$ or $F_{i \mid x=1} = 1$
- **Weak,** if $F_{i \mid x=0} \neq F_{i \mid x=1}$
- **Weaker,** if $F_{i \mid x=0} = F_{i \mid x=1}$ (i.e., x is not in the support of F_i)

when Fi is the reachable target frontier corresponding to phase i. The value of the signal is chosen according to the specific trace at hand.

4.3 Constraint-Based Debugging

As described above, a sequential constraint in the counter-example wizard is described as an associated list of pairs of a Boolean condition and a corresponding phase in which the Boolean condition must hold. In other words, the user speci-

fies a function over the signals in the design and a point in time when the function should hold. The counter-example wizard looks for a trace that leads to a failure and satisfies the constraint, and recalculates all the weak and strong signal values relative to this subset of traces.

Constraints are internally represented as BDDs, and each phase constraint afterwards is intersected with the corresponding reachable target frontier. When a condition is applied to the target frontiers, not every possible trace through the target frontiers is a counter-example any more. States in a frontier that do not comply with the constraint are thrown out leaving some of the traces through the frontiers dangling (i.e., they are not of length N, when N is the number of target frontiers). We remedy the target frontiers by performing an N-step forward propagation followed by an N-step backward propagation through all the frontiers.

The task of calculating a new trace under the constraint now becomes simply finding any trace through the *newly* calculated target frontiers. The multi-value annotation is applied to the new set of target frontiers.

4.4 Computation Penalty

In this section, we present data that compares the CPU time that took our system to compute all the data needed for multi-value annotation and single-value annotation. As seen in the numbers reported in Table 1. based on typical Intel verification cases, multiple-value annotation computation is more costly than single-value annotation computation. Table 1. supports the fact that for average size test cases the multi-value annotation can take 2-3 X more time than single-value annotation. On average the number of single-value counter-example sessions needed to root-cause a failure is more than three. Thus, the computation penalty paid in multi-value annotation is less than the single-value annotation. Additionally each session with a single-value counter-example requires analysis time of the verification engineer. Therefore, our conclusion based on computation data and the utility's deployment in Intel is that although multi-value annotation computation takes more time than single-value annotation, it reduces the overall verification time significantly.

Table 1. Experimental results comparing the model checking time required to compute counter-examples with multi-value annotation and single value annotation making use of eight typical Intel verification test cases

Test case	Multi-value annotation	Single-value annotation
Real 1	31.0 s	21.0 s
Real 2	4.3 s	1.8 s
Real 3	10.7 s	5.2 s
Real 4	292.7 s	89.9 s
Real 5	129.7 s	44.6 s
Real 6	173.0 s	64.2 s
Real 7	8091.8 s	7250.2 s
Real 8	2118.4 s	1421.1 s

5 Experimental Results from the Deployment of the "Counter-Example Wizard"

The counter-example wizard has been used to help debug numerous real-life Intel verification cases. The tool has been found to be beneficial during proof development in reducing the time spent analyzing counter-examples and reducing the number of "check-analyze-fix" iterations, consequently speeding up convergence to a proof. The main advantages of the wizard have been observed to be in determining the root cause of a set of counter-examples and identifying a resolution to the failure.

In this section, we illustrate the benefits of the counter-example wizard through real-life examples. Our productivity claim of the counter-example wizard relies on our experience in its deployment at Intel.

5.1 Determining a Root Cause

5.1.1 Strong versus Weak Values

From our experience, the most important benefit of counter-example wizard has been the ability to home in quickly on the root cause of a counter-example by pinpointing pertinent signal values. The distinction between weak and strong values often indicates which signal values are responsible for a failure and which signal values do not affect the failure (i.e., not related to the root cause). For example, if a signal has weak values in all but one phase, this is a good indication that the one strong phase is the only important phase for this signal. Similarly, if only one of the inputs to a logic gate has a strong value, this is often an indication that the other input signal values can be ignored as being unrelated to the root cause of the failure.

Multi-value counter-example annotation, by reducing the number of relevant signals and phases, reduces the amount of data to be comprehended in the counter-example report. Instead of being distracted by insignificant information, the verification engineer can concentrate on pertinent signals, phases, and values and ignore all the rest. Consequently, it is easier to comprehend the essential behavior of the model, and identify values related to the root cause of a counter-example. In practical use, this characteristic of the counter-example wizard has been its most direct and immediate advantage.

To illustrate how multi-value annotation helps pinpoint the root cause of a failure, we present an example from our experience of the verification of a property involving ten pipe stages of control signals. In one failure report, we observed that almost all the signals in the model had weak values. One signal stood out, however, with a single strong value assignment in a particular phase. This gave us immediate indication that the failure was related to the value of this signal and phase. By tracing the staging of the signal through its several pipe stages, we shortly discovered that at one point the signal was not getting flopped into the next stage (see Fig. 5). Note that the value of the pipelined signal *Reg_Rd* changes from a strong 1 to a strong 0 at cycle 4. A simple examination of the flip-flop revealed that its clock was not toggling during the phase in which *Reg_Rd_s05* took on a strong zero value, illustrating the root cause of the failure. This is a simple case that we could have

debugged even with traditional single-value counter-example reporting. Nevertheless, multi-value signal annotation allowed us to debug much more quickly and without unnecessary trial and error. This is exemplary of the more complex counter-examples with which the wizard has proved helpful.

5.1.2 Sequential Constraints

The "sequential constraint" capability has furthermore helped in root cause determination in our verification problems. There are several cases of when constraints can be useful. First, signals relevant to a counter-example could be related but not constant. For example, if two signals are equivalent in the design under test, but their equality is not guaranteed in the verification, and the verification failure is due to this lack of an equivalence guarantee, then the counter-example report would show both signals having weak values. To see if there is a relationship between the signals, the value of one of the signals would need to be constrained. In this example, if the value of one of the signals is constrained to 0, then the other signal will take on a strong 1 value. When this happens, we know there must be some relationship between these signals with respect to this set of counter-examples (namely, that they should be equivalent but are not in the verified model). The advantage that the wizard has provided us in cases like this is the ability to identify relationships between signals in the counter-examples and thus help direct us to the root cause.

```
@ @1 @@ @@ @@ @@ @@ @@ @@ @@   RegRd_s02
! !! 11 @@ @@ @@ @@ @@ @@ @@   RegRd_s03
@ @@ @@ 11 @@ @@ @@ @@ @@ @@   RegRd_s04
@ @@ @@ @@ @0 @@ @@ @@ @@ @@   RegRd_s05
@ @@ @@ @@ @@ 00 @@ @@ @@ @@   RegRd_s06
----------------------------
0  1  2  3  4  5  6  7  8  9
```

Fig. 6. In this text-based multi-value annotated counter-example trace, the columns represent phases, and the rows represent the values of each of the signals at each phase. The '@' and '!' symbols represent weak 0 and weak 1 values, respectively, whereas '0' and '1' represent the strong values

Another case for which constraints are useful is when the set of counter-examples is due to multiple root causes. Constraints can be used to partition the set of counter-examples to identify the different root causes. This partitioning is not explicit but happens naturally during analysis while experimenting with constraints and searching for a root cause. Once one root cause is identified, analysis is then switched to the cases not covered by that root cause. This is repeated until the set of counter-examples has been fully covered. The set is thus partitioned according to the various root causes.

Multiple root causes usually cannot be identified when working with a single counter-example at a time. A given counter-example may be due to only one of the root causes; therefore only one root cause could be found from that counter-example. Even when a single counter-example is due to multiple root causes, it is unlikely that

all would be identified: When one apparent root cause is found, the verification engineer typically assumes that this is the sole cause. Therefore when multiple root causes exist, the counter-example wizard provides an additional advantage over traditional single-counter-example debugging.

5.2 Identifying a Solution

The counter-example wizard has also been helpful in identifying solutions to a set of counter-examples. In general, once the root cause of a failure is found, the root cause is usually eliminated either by expanding the model to include necessary guaranteeing logic, or by making an environmental assumption about the behavior of signals. The model is then re-verified after the necessary changes have been made. The expectation is that the changes will eliminate any counter-examples, yielding a successfully completed proof.

Analyzing counter-examples one at a time is often a process of trial and error. In a typical verification workflow, a verification run generates a single counter-example, the verification engineer tries to determine the root cause of the failure, a solution is implemented, the model is rebuilt, and the verification is run again. Unfortunately, the root cause may or may not have been correctly identified. Often it results in another counter-example report. When working with only a single counter-example at a time, there is no way to avoid this trial and error process.

The counter-example wizard can eliminate some of this trial and error. Potential solutions can be tested to see if they really do resolve the current set of counter-examples. As noted above, a solution usually takes the form of either an expansion of the model to include necessary guaranteeing logic, or an environmental assumption. Therefore, a solution is essentially a restriction on signal behavior that disallows the behavior observed in the counter-example. To check whether the proposed solution will work, this restriction is given as a sequential constraint to the counter-example wizard. If the wizard determines that there are no counter-examples that satisfy this constraint, then the proposed solution successfully resolves all counter-examples of the given trace length for this verification.

If the wizard does find counter-examples that satisfy the constraint, then the proposed solution does not completely resolve the current set of counter-examples. In this case a different solution can be tried, or the remaining counter-examples can be analyzed to determine why the proposed solution did not resolve them. Possible solutions to verification failures can thus be tested without implementing the solution and rerunning the verification. This can significantly reduce the time spent in the trial and error loop. The wizard gives feedback within seconds, instead of the minutes or sometimes hours it takes to rerun the verification.

In the example in Section 5.1 concerning the flip-flop whose clock failed to toggle, we could quickly test our guess that this was the root cause by specifying to the wizard the constraint that the clock should be high in the phase in which it failed to toggle. We then searched for counter-examples under this constraint, and when none were found, we knew that getting the clock to toggle during the phase

in question would be sufficient to resolve all counter-examples of this trace length.

5.3 Evaluating Counter-Example Wizard Capabilities through an Example

In this section, we present the benefits of counter-example wizard by comparing multi-value annotation versus single-value through an example from a real-life formal property verification case.

Let us first briefly explain the inputs of the verification case: the model, the property to be verified, and the design assumptions.

- Property: If a request is killed, then the register that contains the request gets cleared. More specifically, if this "Active Register" holds a request that receives a kill, then it will be clear for the next two cycles.
- Model Behavior:
  - Eventually a request is received and is retained in the Holding Register.
  - When the Active Register becomes free, the request moves from the Holding Register into the Active Register.
  - When the request is finished being serviced, it is cleared out of the Active Register.
- Micro-architectural Assumption :

The same request cannot be made twice.

```
0 00 00 00 00 00 00 00 00 00 01 00 01 00 00  Valid request
0 00 00 00 00 00 00 00 00 00 00 00 01 00 00  Kill request

0 00 00 00 00 00 00 00 00 00 00 11 00 11 00  Holding reg. Valid
0 00 00 00 00 00 00 00 00 00 00 00 11 00 11  Active reg. Valid

0 00 00 00 00 00 00 00 00 00 11 00 11 00 00  Holding reg. Write enable
0 11 11 11 11 11 11 11 11 11 11 11 11 11 11  Holding reg.
                                             written/reset
0 00 00 00 00 00 00 00 00 00 00 11 00 11 00  Holding -> Active

0 00 00 00 00 00 00 00 00 00 00 00 00 01 00  Holding - special bit
0 00 00 00 00 00 00 00 00 00 00 00 00 00 00  Active - special bit

-----------------------------------------------
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
```

Fig. 6. Traditional single counter-example trace. The columns represent phases, and the rows represent the values of each of the signals at each phase

```
@ @@ @@ @@ @@ @@ @@ @@ @@ @! @@ @1 @@ @@  Valid request
@ @@ @@ @@ @@ @@ @@ @@ @@ @@ @@ @1 @@ @@  Kill request

@ 00 00 00 00 00 00 00 @@ @@ !! @@ 11 @@  Holding reg. Valid
@ @@ @@ @@ @@ @@ @@ @@ @@ @@ @@ 11 00 11  Active reg. Valid

@ @@ @@ @@ @@ @@ @@ @@ @@ !! @@ !1 @@ @@  Holding reg. Write enable
@ 11 11 11 11 11 11 11 11 !! !! !! 11 11  Holding reg.
                                          written/reset
@ 00 00 00 00 00 00 00 @@ @@ !! 00 !1 00  Holding -> Active

@ @@ @@ @@ @@ @@ @@ @@ @@ @@ @@ @@ @1 @@  Holding - special bit
@ @@ @@ @@ @@ @@ @@ @@ @@ @@ @@ @@ @@ @0  Active - special bit

-------------------------------------------
0  1  2  3  4  5  6  7  8  9  10 11 12 13 14
```

Fig. 7. Multi-value counter-example trace. The columns represent phases, and the rows represent the values of each of the signals at each phase. The '@' and '!' symbols represent weak 0 and weak 1 values, respectively, whereas '0' and '1' represent the strong values

In the above traces two requests (Valid request) arrive one cycle apart. For the first request, the Write Enable for the Holding Register goes high in cycle 10, causing the Holding Register to be valid in the next cycle (cycle 11). Also in cycle 11, the Holding→Active signal goes high, causing the request to move from the Holding Register into the Active Register. Consequently, the Active Register is valid in cycle 12. Furthermore, the trace shows a Kill request arriving in cycle 12. (This should kill the request by clearing the Active Register, which it does: the Active Register is not valid in cycle 13.)

However, the property states that the Active Register should remain clear for two cycles, yet we see that it becomes valid again in the next cycle (cycle 14). This occurs due to the second request, which moves into the Active Register by the same process as the first request. Thus the Active Register becomes valid again after only one cycle, rather than the two cycles specified by the property. Hence the violation of the property and the failure of the proof.

Let us now illustrate how multi-value annotation helps the verification engineer to find the root cause of the failure.

- Multi-value annotation helps narrow the scope of the search for the root cause.
  - More strong values are associated with the second request than with the first. Therefore, the focus should be on the second request.
- Multi-value annotation helps identify which logic has to be guaranteed to resolve all counter-examples of the given length.
  - Despite the assumption that the same request cannot be made more than once, the first and second requests arrive two cycles apart from one another. A closer examination (not seen in the above trace) of the second request shows that it is indeed identical to the first request but with one exception: its "Special" bit. Multi-value annotation is helpful in identifying this conclusion. With single counter-example debugging, there is no indication that the Special bit is the only exception; it sim-

ply receives a zero or one, just like every other bit that comprises the request. However, with multi-value annotation the Special bit takes on strong values, so it is certain that here the Special bit is the only important component of the request.

– Now that the second request, in particular its Special bit, has been identified as important, the Special bit can be followed from the request through the registers, as shown in the above trace. The Special bit is not getting passed from the Holding Register to the Active Register in cycles 13-14, even though the request is getting passed. Since the Special bit takes on strong values, we know that if the passing of the Special bit from the Holding to the Active Register can be guaranteed, then we will have resolved all counter-examples of this length.

This debugging session helped the verification engineer pinpoint the missing logic that guarantees the transfer of the Special bit. Once that logic was included in the model, the proof succeeded.

The verification case just described is derived from an actual debugging session from our verification work. We have indeed encountered much more complex counter-examples than the one just demonstrated. They include some with multiple root causes that could be identified all at once using multiple counter-example capabilities, but which single counter-example debugging could identify only one at a time. Such examples are quite complex and are beyond the scope of this paper.

6 Conclusions

In this paper we have introduced a novel formal verification debugging aid, the "counter-example wizard." The novelty of the wizard is in its multi-value counter-example annotation, sequential constraint-based debugging, and multiple root cause detection mechanisms. The benefits of counter-example wizard have been observed in an industrial formal verification setting in verifying real-life Intel designs.

The demonstrated advantages of the formal verification system augmented with the counter-example wizard are a shorter debugging loop and unique help in diagnosing and resolving failures. The time saved was due to faster determination of the root cause of a set of counter-examples, and the ability to identify and resolve multiple root causes in a single proof iteration. Furthermore, the wizard allows the verification engineer to test solutions to verification failures and observe if they really do resolve the apparent root cause.

References

[1] R. Bryant, "Graph-based Algorithms for Boolean Function Manipulations",, IEEE Transactions on Computers,C-35:677-691, August 1986.

[2] K.L. McMillan. "Symbolic Model Checking", Kluwer Academics, 1993.

[3] K. Ravi, F. Somenzi, "Efficient Fixpoint Computation for Invariant Checking", In Proceedings of ICCD'99, pp. 467-474.

[4] R. Fraer, G. Kamhi, L. Fix, M. Vardi. “Evaluating Semi-Exhaustive Verification Techniques for Bug-Hunting” in Proceedings of SMC’99.
[5] R. Fraer, G. Kamhi, B.Ziv, M. Vardi, L. Fix. “Prioritized Traversal: Efficient Reachability Computation for Verification and Falsification”, in Proceedings of CAV’00,Chicago,IL.
[6] I. Beer, S. Ben-Davis, A. Landver. “On-the-Fly Model Checking” of RCTL Formulas”, in Proceedings of CAV’98.
[7] R.H. Hardin, R. P. Kurshan, K.L. McMillan, J.A. Reeds and N.J.A. Sloane, “Efficient Regression Verification”, Int'l Workshop on Discrete Event Systems (WODES '96)
[8] E. Clarke, O. Grumberg, K. McMillan, X. Zhao, ``Efficient generation of counterexamples and witnesses in symbolic model checking", in the proceeding of DAC'95.
[9] B. Kurshan, “Formal Verification in a Commercial Setting”, In Proceedings of DAC’97.
[10] J. Jang, S.Quader, M. Kaufmann, C. Pixley, “Formal Verification of FIRE: A Case Study”, in Proceedings of Design Automation Conference, 1997, Anaheim, CA
[11] R.K. Brayton, G.D. Hachtel, A. Sangiovanni-Vincentelli, F.Somenzi, A.Aziz, S.T.Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S.Qadeer, R.K. Ranjan, S. Sarwary, T.R. Shiple, G. Swamy, T. Villa, “VIS: A system for Verification and Synthesis”, in Proc. of DAC’94.
[12] I. Beer, S. Ben-David, C. Eisner, A. Landver. “RuleBase: An industry-oriented formal verification tool”. In Proc. of Design Automation Conference 1996 (DAC’96)