This research was done as collaboration between IBM and University of Texas.

The research is applied data mining.
Bug localization to be the most manual intensive activity. Verification engineers continue to look for automatic solutions in this area (left upper chart).

Debugging a failure is expensive because there is a unknown distance between the moment the bug is noticed and the origin of the bug. In order to reach the origin of the bug the verification engineer needs to backtrack through the test's execution, step by step, and to continuously analyze the correctness of that step, until it reaches a decision where there is a contradiction between the expected behavior and the behavior exhibited by the design.

The idea of using data mining to provide support in bug localization is not new. In SW there are success stories. The reason why they cannot be immediately applied to hardware is due to the differences between HW and SW. If we would try to find the SW application which is most similar to a HW design, that application could contain a lot of concurrency and would be time sensitive. These are both conditions which are acknowledged in SW as being a problem for the automatic bug localization. On top of these, it is difficult to map the behavior itself, which we notice in RTL, to the design source code, which we have in a HDL. Most of the SW data mining solution use something similar to source code line coverage for their algorithms, which in our case is not relevant.
We keep the basic idea of using data mining for bug localization, but we adapt it to the HW particular conditions.

First particular condition is that we cannot use code line coverage, which would be closest to the SW solution. We need to look for a different source of data. Due to the fact that our solution targets functional behavior it makes sense to use functional coverage events. A functional coverage event records the moment in which certain functionality was exercised during the simulation of a test.

The basic idea is to record the data for both, passing and failing tests, and to compare it.

The passing tests tell us what is an expected distribution of coverage events. They tell us what is “normal”. The distribution of the events in the failing set would be different. The difference points us to something we call deviant behavior. The deviant behavior shows in which way the failing tests are different from the passing tests. It shows which coverage events are for example more likely to happen in the failing tests than in the passing.

The deviant behavior describes to us the bug, and the events which are more likely to happen in one set than in the other point us to the bug origin and its manifestation.

If we succeed to identify these coverage events, we can provide them to the verification engineer as a hint. It is similar to a colleague telling us where to look first. The advantage of
such hints is that it is a lot easier to check if a bug is in a given location, than to back trace it from its manifestation to the origin. If we succeed to provide valid hints most of the time considering that it is rather cheap to validate them, the savings are significant.

Therefore we need a mechanism that would extract these events by simply comparing the data from the passing and failing tests.
Here is how that mechanism works. We start with the recorded data of all coverage events of all passing/failing tests. With the assumption that the failing tests fail due to the same bug, we keep only the data which we can find in all the failing tests. We apply a data mining algorithm which extracts the “distinguishing events” which are the events which best separate the failing from the passing sets of data. The data mining algorithm is known as an algorithm for building decision trees.

We provide visualization of the bug by using location and timing of these events, in something we call MRIs.
This slide explains why, the process of building a decision tree, is the same process of identifying the distinguishing variables.

On the left side you can see a two dimensional chart. The red colored x represent failing tests, while the blue colored x represent passing tests. Each axis represents how many times the test hit event1 respectively event2.

The process of building a decision tree starts with identifying for each variable, if there is a value for which it best splits between passing and failing tests, for example event1 >50

After choosing it, it goes for each subset into which it split, and tries again to identify the next variable with its threshold. It does so until adding new variables doesn’t bring significant value.

This means that, during this process, we actually select those variables which are the best at “distinguishing” between the failing and the passing tests.
We used the bug localization algorithm as explained above on PowerPC verification via simulation. The verification methodology is as seen in this slide. There is a test case generator which generates tests in assembly. The tests are run via simulation on the device under test. The device contains coverage events. Here these functional coverage events are shown as SystemVerilog cover properties added to the critical areas of the HW functionality. Each time such a property is true during the simulation run of the test, its counter records the event in the Coverage Data base. There are more than 150 000 such functional coverage events and daily there are more than 50 000 such runs.
We run our experiment on 9 different consecutive problems, B1... B9. Even though there was no problem with getting as many passing tests as we wanted, we didn’t have that many failing tests. There is a triage which is not explained here, which ensures that the failing tests are highly probably generated due to the same bug.

For each such bug, we constructed the data mining problem. We added to each data record a new field, recording if the test passed or failed. We cleaned the data with filters, which removed outliers and redundant information. We use that data to build the decision tree. The algorithm does the tree pruning, which automatically decides not only which are the distinguishing variables, but also the number of such variables. As you can see in Table 3 in the article, the number of distinguishing variables varies a lot among the nine bugs. Not only there is a large variation, but also these coverage events can belong to one set and not the other. Normally one would expect that the distinguishing events would belong to the failing set of tests. We have B6 where they belong to the passing tests. This means that, whatever behavior which is normally exhibited by the passing tests, is lacking in the failing test. It is one of the complicated cases to debug, when the problem is a missing behavior rather than an exhibited one.
The problems for which we used this method were consecutive bugs, we didn’t preselect them. Even so, by looking at the area where they happened, we ended up with an interesting set of bugs. We have cache coherency problems, memory consistency, there are quite a few exception related bugs, in which B9 was extremely difficult to debug being an asynchronous interrupt which was not properly gated, and also B3 was a trap instruction, another rather interesting bug to identify.

We saw an address translation, and a live lock. Again, live lock problems are particularly difficult, because they generate a lot of traffic and noise, and back tracing them can be an extensive effort.
We applied our data mining technique on all these bugs. The results are seen in this slide. For 5 bugs the events our method identified were a perfect match to the bug they pointed to. They not only were easy to read and pointed straight to the bug, but looking at other coverage events in the area, they selected the best such events to reflect the bug.

In the case of B3 we have a problem. It was a trap instruction decoding. The problem was that there was really no coverage event in that functional area. As a hint, the distinguishing events were not useful, they were confusing, belonging to macros which didn’t seem to have much in common. On the other hand, after finding the bug, the confusion was easily explained. There was no coverage event in the area where the problem occurred, and the events the algorithm pointed to were the entry and the exit to that area.

In spite of not being helpful as a hint, it still could be used to identify the best test from all failing test, to be debugged (as the test where the problem occurs the earliest in the test) and also the window of cycles in which the origin of the problem happens, which saves a lot of time. It is enough to point to the best test to debug, and to the cycle window, to run the simulation and gather information only once, instead of going through a few iterations.

The interesting situation was when the algorithm pointed to way too many coverage events. This happened to three of our problems. The coverage events didn’t just point to the origin of the problem but to a whole manifestation of the problem. For this we decided to use the MRIs.
An MRI is a visual debugging support (Machine Reasoning Image). The whole MRI shows activity during the simulation of a single test. The x axis shows the simulation cycle at which events happen. Each colored dot shows events happening at that cycle. The y axis groups events together depending on the location in the compiled HW of where the events happen. Same color would mean that the events happened in the same location in the HW.

What such an MRI shows us is the manifestation of a bug. The events show the functionality which is different between passing and failing. In this case we see a repetitive pattern. The names of the coverage events are chosen such that, most of the time, by just reading them we can guess their meaning. In this situation is enough to notice that each such pattern starts with SPR operation and ends with DSI RFID (data storage interrupt – return from interrupt) to understand that the repetitive behavior shows an interrupt. That interrupt, in this case, for B6, has a higher preponderance of happening in the passing tests, and is not seen in the failing tests, which ended up being the bug origin (wrong triggering of DSIs)
Another example of how we can visualize bugs is in B7. The events show a repetitive triggering of the purple event which is count_ucode. That event points to the breaking up of an instruction into microcode.

The bug in this case is in memory consistency, where an atomic store operation was expected, but not delivered.
To summarize the work presented in this paper the idea behind this was the use of data mining algorithm on sets of failing and passing tests to identify the distinguishing events. They are passed to the verification engineer as a hint, to reduce the normal time required by back tracing to a cheaper effort of validating a suggestion that the bug is in a given location. The algorithm also selects from the failing tests the test which should be used for debugging, and provides a cycle windows for the bug origin. The work also introduces a visualization of bugs, the MRIs which use time and location to help understand the behavior which is different in the failing tests compared to passing. The work shows the results of applying this method on 9 consecutive real industry conditions.
The results show that the algorithm correctly identifies the bugs with coverage events, where such events are available.
It provides a faster debugging alternative to simple back tracing. Even in the one case in which the hint was difficult to understand, it increased the debugging speed by identifying the test and the cycle window where to look for the problem.
The MRIs are useful as a way to visualize a bug, which automatically provides very efficiently hints on what the deviant behavior really is.

The savings are considerable. Just consider the simple savings inserted by knowing which test reaches the earliest the bug origin area, and the cycle window for which we need to record the signals to debug, without having to do more than an iteration to gather the information we need to debug.