# An Experience of Complex Design Validation: How to Make Semiformal Verification Work

Sabih Agbaria, Dan Carmi, Orly Cohen,
Dmitry Korchemny, Michael Lifshits, and Alexander Nadel
Intel Corporation
P.O. Box 1659
Haifa 31015 Israel
{sabih.agbaria,dan.carmi,orly.cohen,dmitry.korchemny,
michael.lifshits,alexander.nadel}@intel.com

## ABSTRACT

There are two main techniques used for RTL validation: simulation and formal verification. The main drawback of simulation is its inability to provide satisfactory design coverage when the number of important scenarios is very large. Formal verification provides exhaustive coverage, but its capacity is insufficient for realistic designs. In this paper we describe our experience with semiformal verification (SFV) techniques used to validate two CPU design blocks each of which included novel features carrying high risk to the project. On the one hand, the number of different scenarios in these blocks was enormous, and thus simulation could not provide satisfactory coverage. On the other hand, these blocks were too complex to be formally verified. Applying the proposed method to these designs, believed to be mature after many weeks of intensive dynamic and traditional formal validation, revealed bugs in both the design and validation collateral, some of them critical. The results obtained show that SFV has good potential for RTL validation, and that it can save a substantial amount of the effort required to cover important scenarios in simulation or to manually build an abstraction model for formal verification. Our semiformal algorithm uses formal engines only (and runs only on the formal verification model) to explore scenarios requiring many clock cycles to execute, and it has an important advantage over most other approaches (which combine formal engines with simulation) – it circumvents the consistency problems between the simulation and formal verification models of the design.

## Categories and Subject Descriptors

B.6.3 [**Logic Design**]: Design Aids—*Verification*

## Keywords

Semifotmal Verification, Model Checking

## 1. INTRODUCTION



The complexity of contemporary hardware (HW) designs imposes a heavy burden on their validation. Validation has become a bottleneck of HW projects, such that many important new features are dropped because of the inability to verify them in a reasonable time. Though raising the design abstraction level would certainly provide a significant validation efficiency boost, there is still a long way to go to the enforcement of high level modeling and verification methodology in big HW design projects. In this article we focus on Register Transfer Logic (RTL) level design description.

There are two main approaches to RTL validation — dynamic simulation and formal verification (FV). FV is exhaustive, but its complexity grows exponentially as the size of the design increases. There is a common belief that simulation scales well since its complexity relates linearly to the size of the design. However, this is not correct, as simulation is not used per se, but rather to ensure at least some minimal coverage of the design space to get confidence in design correctness. Of course, no practical number of simulation runs can provide exhaustive verification of the design. The number of important scenarios to be exercised in simulation grows exponentially with design complexity; thus the required number of simulation runs also grows exponentially [13]. Since in practice the total number of simulation tests cannot grow that fast, the validation gap keeps increasing.

One promising direction for bridging the verification gap is to explore semiformal verification technologies that combine simulation and FV algorithms. Semiformal verification methods do not provide the exhaustiveness of purely FV methods, but they do achieve much larger coverage of the design space than simulation methods. On the other hand, the capacity of SFV is far superior to that of FV.

Since part of SFV is FV, in order to run SFV, a FV environment is needed. The FV environment requires specifying assumptions which constrain the inputs of the device under test (DUT) to model the DUT environment. This is relatively simple to do for big blocks with well-defined functionality, since their interface protocol is also well defined. To fit FV capacity limitations, it is usually required to verify only a part of the DUT, whose functionality is described only in a larger context. In this case, creating an FV environment becomes painful and time consuming. Since the capacity of SFV is usually significantly higher, it can often run on the original blocks, so that the effort of building the environment in SFV may be much smaller than in the case of FV. This is a huge advantage of SFV.

In this paper we describe our experience applying SFV to the verification of several key modules of leading Intel CPU designs, among them Resource Manager and Request Tracker. Our experience clearly shows the added value of SFV. Using SFV we were able to find several important functional bugs that could not

be revealed during many weeks of intensive dynamic simulation. Neither could these bugs have been revealed by FV tools, as running exhaustive FV was unfeasible, and bounded model checking (BMC) could not get to the deep bounds necessary to reach these bugs.

The notion of SFV is not new (see, for instance, [17]), and there exist several industrial and academic tools [2,7,9,12] implementing different SFV verification algorithms. However, we were unable to run our design on any third party tool available to us because of the numerous limitations of these tools in handling our design methodology and in SystemVerilog support. Adapting the existing design to meet the tool limitations turned out to be impractical, and instead of adapting the design to existing tools we had to create our own SFV tool. The SFV verification algorithm we used shares the main idea described in [11], but the area of application is different. The algorithm described in [11] was applied to post-silicon debug, while our primary area of application is pre-silicon validation. In addition, we implemented many important enhancements, the most significant of them being multiple witness generation, as described in Section 3.3.

Our design was written in SystemVerilog, and we used SystemVerilog Assertions (SVA) as our assertion specification language [14]. All assertion examples in this paper are also written in SVA. For clarity and conciseness we omit explicit clock and reset specification in assertions, and assume that some default values are used.

The rest of this paper is organized as follows. Section 2 provides a short overview of SFV methods relevant for our work. Section 3 describes our SFV algorithm. Section 4 reviews the expertise required to apply the suggested SFV solution, and identifies design areas where SFV is most beneficial. Section 5 contains high-level description of the design blocks to which SFV was applied. Section 6 presents the results obtained. Conclusions follow in Section 7.

# 2. SEMIFORMAL VERIFICATION METHODS

There is a large variety of SFV methods, and an exhaustive overview and detailed taxonomy of them are beyond the scope of this paper. We refer the reader to the survey of Bhadra et al. [3] for a detailed description. In this section we mention only those features that are important for the justification of our choice of implementation.

One of the basic ideas of SFV suggested by Yang and Dill [8] is the idea of *waypoints*, or *guideposts*. Waypoints are predefined points or areas in the state space of the model. Instead of trying to formally prove an assertion directly, the SFV tools try to hit some subsets of the waypoints in the order defined by a specific search policy. The assertion to be verified is checked not directly from the initial state, but from one of the waypoints. If this waypoint is close enough to an assertion failure state, then it will be easy for the FV tool to hit this failure state. This method can be used for bug hunting only; the inability to discover a bug does not guarantee that the design is correct. However, the coverage of the design space achieved by the described SFV framework is much higher than that of dynamic simulation, and even in the absence of bugs, the confidence in design maturity when using SFV is higher.

We limit our discussion to SFV methods using the idea of waypoints. The waypoint-based SFV methods may be classified using the following parameters:

- Waypoint definition.
- Waypoint traversal policy.
- Propagation policy.
- Formal verification engine.
- Number of search threads.

This taxonomy is not perfect since its classification parameters are not independent, and it is by no means complete. However, we will stick with it as it serves our needs well, and covers major EDA SFV tools such as Synopsys Magellan [18] and Mentor 0-in Dynamic Formal Verification [10].

## 2.1 Waypoint Definition

Waypoints may be defined explicitly by the user [9,11], or generated automatically by a SFV tool [5,15,20]. Users have key knowledge about the behavior of the DUT, and they can provide a small group of highly efficient guideposts. For example, if we want to check for a queue overflow, the waypoints could be: "queue is $1/4$ full", "queue is half full", and "queue is $3/4$ full".

The advantage of automatically generated waypoints is that they do not require user intervention and manual waypoint specification. Automatic waypoints are usually generated based on proximity metrics. For example, for property
`a |=> b ##1 c` the automatically generated waypoints could be states where `a` is true, and states where `b` is true such that they have a predecessor where `a` is true. Other methods might include, for example, waypoint selection among states of an abstracted version of the design space at a specific distance from the assertion failure state [6].

The main disadvantage of automatically generated waypoints is their large number and the fact that many of them are inefficient. Both these factors may significantly increase verification time or even make verification unfeasible, as reaching each and every waypoint is computationally expensive.

Some tools use a mixed strategy, where some waypoints are provided by the user and others are generated automatically by the tool.

## 2.2 Traversal Policy

The waypoint traversal policy defines the order in which the waypoints are traversed when the failure state of the assertion being verified is searched.

The simplest traversal policy is waypoint traversal in a specific order. More sophisticated policies may include hitting the closest waypoint, hitting the closest waypoint among those that are closer to the assertion failure area, etc. The latter policies usually also include retreat and restart strategies: if there is no significant progress detected for a long period, the search is restarted from an earlier point and the latter part of the path is inserted into a "black" list in order not to repeat the same path twice.

## 2.3 Propagation Policy

The main policies for reaching new waypoints are dynamic simulation, random simulation, and FV-based propagation. In a dynamic simulation policy some existing simulation test is run and the waypoints are selected on the simulation trace either manually or automatically. From each of these waypoints an assertion violation condition is searched for using FV methods, such as symbolic simulation or BMC [10]. These FV runs are usually shallow, and their goal is to look for assertion violations in the proximity of the simulation trace. Since the simulation trace effectively becomes "thicker" from the coverage point of view, this method can be referred to as *simulation trace amplification*. The big advantage of this method is that it explores real-life scenarios. It has, however, two important drawbacks: it requires the availability of a simulation environment and simulation tests *in addition* to the

FV environment, and it requires that the FV and simulation models be consistent. The latter drawback is very serious, as in practice simulation and FV models may differ significantly. For example, the FV model may have a smaller memory than the dynamic verification model or several parts of original logic may be replaced with shortcut logic in order to fit the capacity of the FV tools. This circumstance makes the use of a dynamic simulation policy problematic for complex HW designs.

In a random simulation policy, the model is simulated randomly when trying to hit the next waypoint (e.g. this method is included in Ketchum [12]). This policy does not necessarily require creation of a simulation environment and tests, when the external inputs are constrained with SVA (and treated as assumptions), but its main challenge is the necessity to respect these constraints. To be able to conduct FV, it is crucial to specify assumptions constraining the behavior of the environment in order to prevent false assertion failures. Since FV engines are part of the SFV flow, random simulation should take into account all constraints imposed by assumptions when generating random stimuli. Though random simulation is very popular in SFV, assumption handling is the Achilles' heel of this policy. While random simulation can handle simple assumptions on inputs rather effectively, it is difficult to resolve complex temporal assumptions, especially to generate high quality random stimuli. Some kinds of assumptions cannot be resolved in principle, such as those that constrain input behavior that depends on output behavior, for example "Output `ready` cannot be asserted without previous assertion of input `req`". These issues limit the applicability of random simulation.

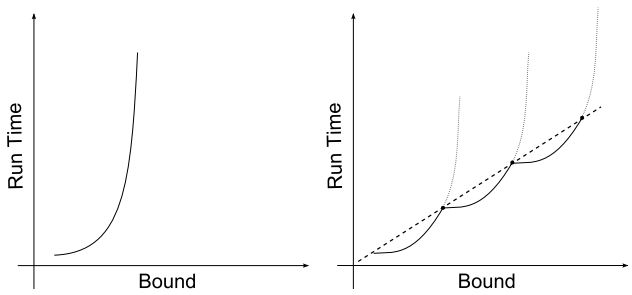
In FV-based propagation, waypoints are treated similarly to assertions. For example, the waypoint "queue is half full" may be represented as an assertion "queue is never half-full". The FV engine tries to violate the latter assertion, and if it succeeds, it generates a counterexample telling what stimuli should be applied to make the queue half-full. This is exactly a path to the waypoint, also called a waypoint *witness*. FV-based propagation does not require any additional environment except for the regular FV environment which is needed in any case to conduct FV and which is always part of SFV methods. Unlike random simulation policy, FV-based propagation does not impose any limitations on the form of the assumptions. In other words, a big advantage of FV-based propagation is that the verification environment required for it is exactly the same as in the case of classical FV, except, of course, for waypoint specification, if explicit waypoint definition is used (Section 2.1). The drawback of this policy is the requirement that consecutive waypoints be relatively close to one another in order to fit the capacity of the FV engine.

## 2.4 Formal Verification Engine

SFV methods differ depending on the FV engine used: symbolic simulation [16], BDD-based model checkers [16], BMC [4], etc. In the early days of SFV, BDD-based model checkers were mostly used, but at the present, BMC methods have became more popular, as they posses much higher capacity than other FV engines.

## 2.5 Number of Search Threads

Depending on the number of search threads, SFV methods can be subdivided into single-threaded methods and multi-threaded methods. Here "thread" is not necessarily a thread in the programmatic sense; an SFV-application exercising multiple search threads may be implemented as one single-threaded process, though such implementations are rare. In single-threaded methods, only one path through waypoints is sought, while in multi-threaded methods many paths through waypoints are considered. Multi-threaded methods provide better coverage of the design space than single-threaded methods, but this comes at the price of verification performance, as multi-threaded methods require much more computational resources than do the single-threaded methods.



(a) Formal verification (b) Semiformal verification

**Figure 1: Verification time reduction**

# 3. BMC-BASED SEMIFORMAL VERIFICATION

In this section we describe our new SFV method. Our method is purely BMC-based, i.e. no simulation is involved, and thus no synchronization between dynamic and formal verification models is required.

## 3.1 Basic Algorithm

BMC is a powerful (and the most commonly used) FV technique that verifies the behavior of the DUT for input sequences of bounded length. It starts from the initial state of the DUT and searches for a run of one clock cycle that violates an assertion. If no assertion violation is found, the number of run cycles is iteratively increased. The number of different scenarios grows very quickly with the length of the run. The proposed BMC-based semiformal algorithm executes multiple shallow BMC runs, trading the exhaustiveness of a search for speed. The user provides an ordered set of *waypoints* which direct the search engine towards the desired deep design state. The algorithm searches for a path from one waypoint to the next, starting from the initial state; the BMC engine is restarted at each waypoint in order to avoid exponential blowup. This idea is illustrated in Fig. 1. The time needed to reach a deep design state is reduced from exponential to approximately linear in the number of clock cycles, which makes it possible in practice to get to states that would never have been reached with traditional BMC.

As described in Section 2.2, some SFV methods automate the waypoint search and use various heuristics to guide the tool. In our approach, however, we let users provide high-level direction for the semiformal search towards the desired area by encoding the waypoints with SVA cover points. Our experience shows that, being familiar with design behavior, most users define these high-level directions quite naturally. For example, consider a queue that requires 200 clock cycles to be filled. To validate the queue control logic in a stress *"full queue"* state, possible waypoints could be *"1/4 full queue"*, *"1/2 full queue"*, and *"3/4 full queue"*, each waypoint being easily reached by the tool. We did implement an automation of the traversal, (see Section 3.4 for details), but our experience shows that most users prefer provide high level direction for the tool manually.

The stages of the SFV flow are outlined in Fig. 2. First, the user defines a series of high-level waypoints modeled with SVA cover

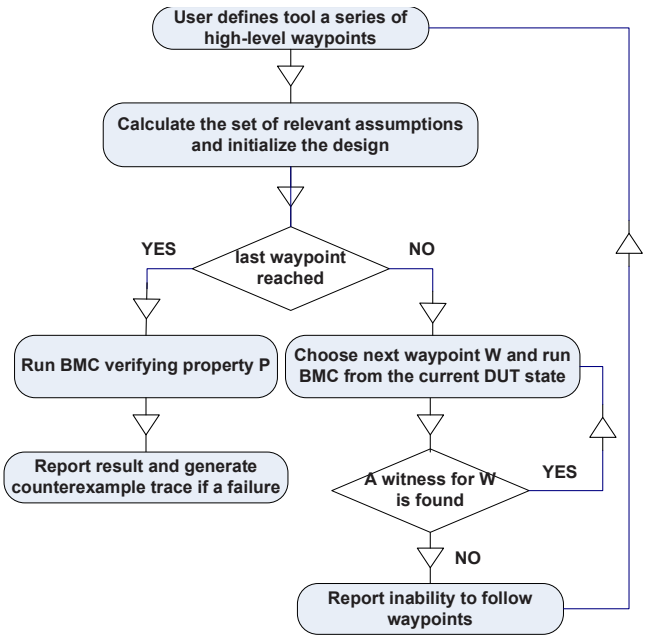



**Figure 2: Semiformal verification flow**

points $c_1, c_2, \ldots, c_n$ and the assertions to verify $p_1, p_2, \ldots, p_n$. SVA is part of SystemVerilog – it can be embedded in the DUT code or written in a separate bounded module using the same compilation command. Design engineers use a familiar language quite easily to embed SVA assertions/waypoints in their code. It is recommended to capture sophisticated, multi-cycle SVA expressions required by a project in a library using functional templates. Given the assertions, the set of relevant user-specified assumptions for all $p_i$ is calculated statically by traversing their cones of influence. It is important to use the whole set of assumptions in detecting of each cover point's witness (in addition to the assumptions relevant for the specific cover point), because otherwise, when a property failure is found and the witnesses of all waypoints are concatenated to form the counterexample trace, some part of the trace may not comply with the assumptions and may make the whole trace spurious. Two actions are performed for each waypoint: BMC verification to find the witness for the cover point or a counterexample for an assertion failure, and simulation[1] to properly initialize the data for the next BMC run (see details in Section 3.2). These actions are repeated, targeting subsequent waypoints $c_2, \ldots, c_n$ thus analyzing deepening design behaviors. If a witness is not found for some $c_i$, an indeterminate result is reported. If there is an assertion failure, its counterexample is appended to the concatenation of witnesses $c_1, \ldots, c_n$. If a timeout or required BMC bound is reached, a lack of failure is reported. Note that this is one of the two traversal policies we implemented; we review both policies in detail in Section 3.4.

## 3.2 Calculation of New Initial States

Calculation of a new initial state when the engine starts from some intermediate waypoint should take into account the recent state of the DUT (the values of all the sequential elements in the DUT) and the recent state of the properties: assertions and assumptions. We use the same initialization method for both the DUT and the properties due to the fact that the properties may be represented as finite automata [19] and that the automaton of a safety property can be synthesized into RTL [1]. We use a conventional RTL simulator to simulate both the DUT and the RTL synthesized from the properties on the waypoint witness. As an example, consider the following assumption

```
m1: assume property (a |-> ##4 b);
```

The RTL representing its automaton is shown in Fig. 3.

```
bit q1, q2, q3, q4;
initial {q1, q2, q3, q4} = '0;

// posedge clk is the assumption clock
always @(posedge clk) begin
  q1 <= a; q2 <= q1; q3 <= q2; q4 <= q3;
end
wire fail = !b && q4;
```

**Figure 3: RTL for assumption $m1$**

If the value $\texttt{a} = 1$ in the witness appears in the next to last step, the initial state of the next BMC run should have $\texttt{q2} = 1$. Simulating the property automaton is important: blindly reusing the initial property condition `initial q1, q2, q3, q4 = '0;` would have led to a discontinuity of the adjacent BMC runs, and potentially to false negatives and to bogus witnesses and counterexamples.

## 3.3 Using Multiple Witnesses to Enhance Coverage

Our experiments described in Section 6 show that the proposed basic single-threaded algorithm will likely miss corner-case bugs. The reason for this is that a randomly chosen path, constructed from a series of witnesses each of which satisfies the corresponding intermediate waypoint, does not exhibit sufficient coverage of the design space.

To achieve better design space coverage we implemented a multi-threaded search algorithm that advances towards the desired deep states along multiple paths in parallel. For each intermediate waypoint, a random set of witnesses is calculated instead of a single witness, and for each such witness a separate verification process towards the next waypoint is launched. Fig. 4 illustrates a scenario where using two witnesses for the waypoints resulted in bug detection, whereas the chances of detecting the bug would have been much smaller otherwise.

As described in Section 3.1 our basic algorithm is BMC-based. In BMC the reachability condition from one waypoint to the next is coded as a Boolean formula which is then solved with a SAT solver to produce a waypoint witness. To produce multiple instances we modified our SAT solver to provide several satisfying assignments instead of one. The main challenge was to generate several satisfying assignments to the same formula which were as different as possible in order to minimize overlap.

Our results (Section 6) show that the quality of the multi-threaded algorithm is far superior to the quality of the single-threaded algorithm. However, the number of verification threads grows exponentially with the number of the waypoints, since at each waypoint a fixed number of new threads is generated. This may be remedied by limiting the total number of threads by a predefined upper bound.

## 3.4 Manual and Automatic Traversal Policies

[1]In this case the FV model is simulated, which is different than in the case of dynamic formal verification, where the DV model is simulated, a fact which requires synchronization between the models

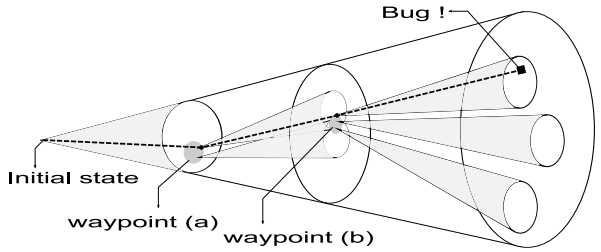



**Figure 4: Multiple witnesses**

**Algorithm 1** Automatic usage mode

1: Try to find a failure for any of the assertions from the initial state of the model. This is a regular BMC run
2: Try to reach each cover point in the model starting from the model initial state
3: When a witness of any cover point is found, the above steps are repeated starting from the initial state specified with the newly reached cover point

We identified that there are two major usage modes for the SFV algorithm. In the first, the "manual single scenario" usage mode described in 3.1, the user verifies a set of properties that are expected to fail in a specific design scenario. This mode employs FV-based propagation (see 2.3). The user indicates a series of events (waypoints) that guide the way through the scenario and lead to a potential property failure. Consider, for example, a block that stores the incoming requests in a queue. The correct behavior of the control logic (e.g. calculation of the STALL condition) should be validated in appropriate stress scenarios, e.g. when the queue is full. In this case, FV-based propagation following the waypoints specified by the user and verifying the properties once a stress condition is reached will achieve the highest confidence in the correctness of the properties.

In the second, the "automatic multiple scenarios" usage mode, the user verifies a set of properties that may be violated in a wide range of design scenarios. In this case, especially if the user is not very familiar with the design, it is difficult to specify a large number of scenarios with waypoints. Therefore, because it lacks user guidance (a series of events leading to a potential property failure), the algorithm's traversal policy should uniformly cover as much of the reachable design space as possible to achieve the highest confidence in the correctness of the properties. For this, the algorithm uses the set of available cover points in the model that were specified for some other purpose, e.g. to qualify the modeling (the reachability of these cover points is verified to make sure that no overrestricting assumptions added to the model interface are masking important behaviors). Algorithm 1 aims to verify all the assertions from as many initial states as possible. These algorithm steps are done automatically, and various verification runs are simultaneously executed using multiple threads. This traversal policy does not require a good familiarity with the model, but it can be very computationally intensive, as the maximal number of verifications can reach the factorial of the number of cover points in the model. Users generally limit the maximum number of parallel verifications. Our experience shows that most users prefer the single scenario mode, where they manually provide high-level direction for the tool.

# 4. APPLICATION GUIDELINES

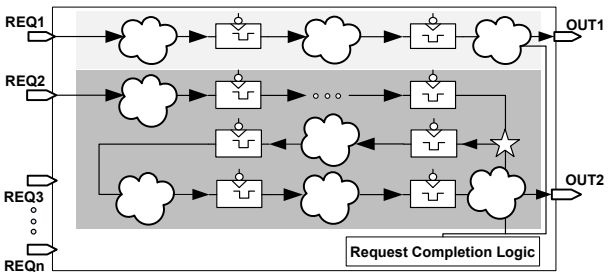



**Figure 5: Request Tracker**

In this section we review the expertise required to apply the suggested SFV solution. Additionally we specify design areas where we think the application of SFV would be most beneficial.

To effectively run the flow using FV-based propagation (see 3.4), users should be familiar with the micro-architectural specification of the DUT as well as with the property specification language, i.e. SVA. This is required for specifying the appropriate cover points and for matching them to the corresponding assertions (usually resource limitations prevent checking every assertion from every cover point). Users should also be familiar with FPV tools and methodology to determine that SFV is the right solution to overcome FV complexity problems, considering other traditional solutions, such as DUT reduction by abstraction, black-boxing, and pruning among others.

We found that large DUTs, where classic FPV gets into complexity problems and is unable to achieve sufficient confidence, are the most fruitful candidates for the application of SFV. Such DUTs usually include complex logic comprising mixed control [2]/datapath [3] logic and involving a big coverage space. For example, long flows and/or protocols with deep pipelining and queues and/or counters require high BMC bounds to reach and validate stress scenarios.

# 5. TEST CASES

We implemented the algorithm in a proprietary formal verification tool and applied it to various CPU design blocks. The RTL was written in SystemVerilog and included novel features carrying high risk. A significant multi-month effort had already been invested in the simulation and FV of most of these blocks. The FV confidence level was not high enough in all cases, as the BMC bound reached by the traditional BMC approach was insufficient. In most cases, after reducing the models using both manual and automatic techniques, the full cone of influence of a typical assertion was of a size FV engines could handle — 1K inputs, 5K state elements, and 75K gates. Still, design scenarios requiring more than forty clock cycles could not be addressed by FV. Thanks to the intensive FV work that had already been done, the environment included hundreds of assumptions, assertions, and cover points captured in SVA. As a result, the SFV flow could be instantly applied.

The first block, Request Tracker, is responsible for managing various request types and ensuring the correct execution order between requests, giving preference to high-priority requests while not starving the low-priority ones. The high-level diagram of Request Tracker is shown in Fig. 5.

The different request types vary in the time needed to process them, e.g. request $REQ1$ (path $REQ1$ — $OUT1$) requires con-

[2]producing the values of control signals based on data values and states

[3]functions and registers operating on data based on the value of the control signals

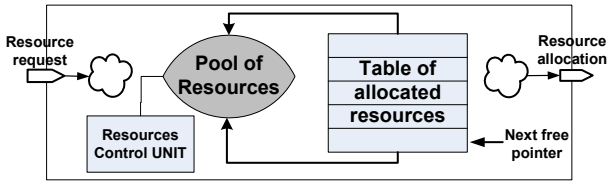



**Figure 6: Resource Manager**

siderably fewer clock cycles than request $REQ2$ (path $REQ2$ — $OUT2$). The requests come from different sources, and each request has a unique ID. We chose to experiment with $REQ2$, which could not be properly addressed in FV due to BMC bound limitations.

The second block, Resource Manager, is responsible for controlling resources and ensuring that no resource is allocated twice and that none are lost. The resources are kept in the pool and allocations/deallocations are recorded using a cyclic table. See Fig. 6 for the high-level diagram of the block. We will not describe other blocks in detail.

# 6. RESULTS

In this section we describe results illustrating the efficiency of our SFV tool and compare it with that of pure BMC. Section 6.1 describes the results of detecting artificial bugs manually inserted into the design blocks. Section 6.2 describes real bugs detected by the SFV tool.

## 6.1 Testing the Ability to Adequately Cover Design Behaviors

To verify the SFV algorithm efficiency we inserted artificial bugs into the designs and allowed choosing waypoints from *existing* cover points previously defined by validation engineers for other purposes. We did this in order not to explicitly guide the tool towards the known bug. We made sure that the bugs were of a "good quality", i.e., that they could not be revealed in FV or by existing simulation tests.

The artificial corner-case bug we inserted in the Request Completion Logic sub-block of the Request Tracker (see Fig. 5) caused a failure when multiple requests of type $REQ2$ arrived from particular, unusual sources in a specific order. The bug resulted in one of the requests being incorrectly marked as completed. We used 9 different waypoints modeling several $REQ2$ requests in various pipe stages on a path $REQ2$ — $OUT2$; for example, the one marked by a star in Fig. 5. For each waypoint, we calculated 5 witnesses, targeting each of 12 assertions, starting verification from 9×5=45 different initial states (defined by waypoints' witnesses). The cover points occurred at bounds 64–70, and verification took 1406–3379 seconds (on a machine with 4Gb memory and two Intel Xeon CPU 3.60 processors). A failure was detected by one of the twelve assertions from only one initial state, whereas runs from the other 44 initial states missed the problematic scenario. It occurred at bound 34 after 14707 seconds, starting from a waypoint with a 70 clock phase long witness and resulting in a 104 (70+34) clock phase long combined counterexample starting from the original initial state.

The artificial corner-case bug we inserted into the Resource Manager was related to control logic calculating the condition for next request $STALL$. This caused *Next free pointer* to wrap around early, due to illegal allocation, thereby running over other resources in the table. The general cover points used as waypoints asserted that various table lines were allocated, and the table was incrementally filled up in the tested scenario until it it became full and the wraparound occurred. We ran traditional BMC and SFV with single as well as multiple witnesses. The assertion verified that "resources are not being lost in the system". In all cases a timeout of 20 hours was used. Results are summarized in Table 1.

The cyclic allocation table size is 20, and thus a wrap around happens after the 19th table line is allocated. BMC could not get beyond the allocation of line 8, whereas the SFV algorithm easily reached the required stress scenario. The multiple witness approach was necessary in order to come across the problematic combination of resource requests. The total number of verification runs was $3(\text{witnesses})^{6(\text{waypoints})}$=729. Note that the SFV algorithm does not necessarily produce the shortest counterexample — "line 8" was reached with bound 71 using BMC whereas using SFV it was reached with bound 71 to 83.

## 6.2 Corner-Case Bugs in Mature Design

The most important result achieved by applying the proposed SFV tool to the validation of the blocks described in Section 5 is the exposure of three corner-case bugs in the Resource Manager design, two of them critical:

- Incorrect $STALL$ calculation in a very specific combination of allocation requests (a real version of the artificial bug we described earlier), which caused resources to be lost.
- A bug in recovery/restart event handling, which results in not all of the allocated resources being correctly sent back to the resource pool.
- Corruption of a mechanism which validates resource integrity in the Resource Control Unit, in case of extremely high allocation traffic.

Neither FV nor simulation could reveal these bugs, even after many weeks of continuous, exhaustive testing.

## 6.3 Other Results

The proposed SFV tool was able to reproduce a number of bugs previously found in simulation, as well as many bugs in the FV environment modeling. The tool also made it possible to root-cause a fatal post-silicon bug after a two-month manual effort to reproduce it on RTL using other techniques proved futile. The post-silicon debugging activity was performed in a way similar to that described in [11].

In addition, the proposed SFV tool enabled FV activities on large DUTs with deep stress scenarios non-addressable with traditional FV. The relatively low BMC bound achievable by FV on these units was sufficient to hop between waypoints towards the stress scenarios. Usually, a considerable manual effort is required to create an artificial model sufficiently reduced so as to enable achieving significant FV confidence on a DUT. Such a modeling effort can constitute up to 50 percent of the validation work. In some cases, this effort was saved. In other cases, which otherwise would have been dropped, the SFV tool enabled FV.

# 7. CONCLUSION AND RECOMMENDATIONS

The summary of our findings is as follows.

- The proposed SFV solution, in spite of its relative simplicity, was proved to be efficient. It was able to detect, with reasonable effort, 5 artificially inserted corner-case bugs that were missed by FV (due to bound limitations) and simulation (due to coverage limitations).

**Table 1: Resource Manager Verification Results**

| Cover/Assert. | BMC | | Semiformal, single | | Semiformal, multiple | |
|---|---|---|---|---|---|---|
| | **Result** | **Bound** | **Result** | **Bound** | **Result** | **Bound** |
| Line 4 | covered | 69 | covered | 69 | covered | 69 |
| Line 8 | covered | 71 | covered | 71 | covered | 71..83 |
| Line 12 | uncovered | 24 | covered | 89 | covered | 85..95 |
| Line 16 | uncovered | 26 | covered | 99 | covered | 93..107 |
| Line 19 | uncovered | 26 | covered | 113 | covered | 99..119 |
| Line 0 | N/A | N/A | covered | 129 | covered | 107..133 |
| Assert. | TO | 38 | TO | 42 | failed | 142 |

- The proposed SFV solution detected a number of actual, hard to expose RTL bugs very close to project release.
- It is feasible to build a custom SFV flow on top of existing FV tools in a project. Two main ideas should be kept in mind: 1) no simulation is required for the calculation of the new initial states if only boolean properties are being used (see Section 3.2 for details) 2) The FV tool should support multiple witnesses generation to enhance SFV coverage.
- To run the SVF tool effectively, users should be familiar with the micro-architectural specification of the DUT, the property specification language (i.e. SVA), as well as FPV tools and methodology. Applying the SFV solution is most worthwhile on large DUTs where classic FPV runs into complexity.

In our opinion, the suggested method for pure SAT-based SFV is very simple to grasp and straightforward to implement, yet it exhibits superior abilities to achieve good design coverage and to detect deep, corner-case bugs in industrial-scale designs. The encouraging results described earlier were achieved with a relatively small amount of work on the part of the validation engineers. In addition, the suggested method boosts the productivity of the validation team by rendering unnecessary the substantial validation effort required to reduce design size to fit the capacity limitations of FV tools.

## Acknowledgment

The authors would like to thank Paul Inbar, Roy Frank, Tamir Salus, Zurab Khasidashvili, Asi Sapir, Haim Kerem, Dani Even-Haim, Amit Palti, Eli Singerman, and Alon Flaisher for their valuable suggestions, ideas, experimental results, reviews, and support of our work.

## 8. REFERENCES

[1] R. Armoni, S. Egorov, R. Fraer, D. Korchemny, and M. Vardi. Efficient LTL compilation for SAT-based model checking. In *ICCAD*, 2005.

[2] A. Aziz, J. Kukula, and T. Shiple. Hybrid verification using saturated simulation. In *DAC*, pages 615–618, 1998.

[3] J. Bhadra, M. S. Abadir, L.-C. Wang, and S. Ray. A survey of hybrid techniques for functional verification. In *IEEE Design and Test of Computers*, volume 24, pages 112–123, 2007.

[4] A. Biere, A. Cimatti, E. Clarke, O. Strichman, and Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.

[5] P. Bjesse and J. Kukula. Using counter example guided abstraction refinement to find complex bugs. In *DATE*, page 10156, 2004.

[6] F. M. De Paula and A. J. Hu. An effective guidance strategy for abstraction-guided simulation. In *DAC '07*, pages 63–68.

[7] D. L. Dill. What's between simulation and formal verification? (extended abstract). In *DAC '98: Proceedings of the 35th annual Design Automation Conference*, pages 328–329, New York, NY, USA, 1998. ACM.

[8] D. L. Dill and C. H. Yang. Validation with guided search of the state space. In *DAC*, pages 599–604, 1998.

[9] M. Ganai, P. Yalagandula, A. Aziz, A. Kuehlmann, and V. Singhal. Siva: A system for coverage-directed state space search. *J. Electron. Test.*, 17(1):11–27, 2001.

[10] M. Graphics. 0-in formal verification. http://www.mentor.com/products/fv/.

[11] C. R. Ho, M. Theobald, B. Batson, J. Grossman, S. C. Wang, J. Gagliardo, M. M. Deneroff, R. O. Dror, and D. E. Shaw. Post-silicon debug using formal verification waypoints. In *DVCon*, 2009.

[12] P. H. Ho, T. Shiple, K. Harer, J. Kukula, R. Damiano, V. Bertacco, J. Taylor, and J. Long. Smart simulation using collaborative formal and simulation engines. In *ICCAD*, pages 120–126, 2000.

[13] A. J. Hu. Simulation vs. formal: Absorb what is useful; reject what is useless. In *Hardware and Software: Verification and Testing*, volume 4899, pages 1–7, 2008.

[14] IEEE. *IEEE Standard for SystemVerilog – Unified Hardware Design, Specification, and Verification Language*, 2005.

[15] A. Kuehlmann, K. L. McMillan, and R. K. Brayton. Probabilistic state space search. In *ICCAD*, pages 574–579, 1999.

[16] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell, MA, USA, 1993.

[17] K. Ravi and F. Somenzi. High density reachability analysis. In *ICCAD*, 1995.

[18] Synopsys. Magellan. http://www.synopsys.com/TOOLS/VERIFICATION/FUNCTIONALVERIFICATION/Pages/Magellan.aspx.

[19] M. Y. Vardi. An automata-theoretic approach to linear temporal logic. In *Proceedings of the VIII Banff Higher order workshop conference on Logics for concurrency: structure versus automata*, pages 238–266, 1996.

[20] P. Yalagandula, V. Singhal, and A. Aziz. Automatic lighthouse generation for directed state space search. In *DATE*, pages 237–242, 2000.