2022
DESIGN AND VERIFICATION™
DVCON
CONFERENCE AND EXHIBITION
EUROPE
MUNICH, GERMANY
DECEMBER 6 - 7, 2022

An Efficient Methodology for Mutation-Coverage-Collection of Formal-Property-Checking

Holger Busch, Infineon Technologies AG, Neubiberg, Germany (*holger.busch@infineon.com*)

***Abstract*—This paper discusses the significance of mutation-coverage-based approaches for assessing and attaining completeness of formal-property sets and fulfilling sign-off criteria, highlights difficulties to which formal-verification is more exposed than simulation, but also shows how specific strengths of a formal tool are exploited in an efficient automated coverage collection methodology which handles even biggest designs verified by formal-property-checking.**

Keywords*— *formal property checking;mutation coverage;completeness;ISO26262

I. Introduction

A. Structural Coverage

While *functional coverage* is related to verification in verification plans derived from requirements and specifications, *structural coverage* refers to the code implementing a specification, without considering whether the specification has been captured correctly and completely. Thus even 100% structural coverage does not guarantee that the design exposes intended behavior in all circumstances in the field. On the other hand, if regions of the design have not been touched at all by the verification, these have a high risk to contain unknown bugs if they are not anyway redundant or unreachable, i.e. could be removed without impact on design behavior.

Classically, structural coverage is computed by checking which lines, block entering or other conditions are activated at least once by a set of simulation traces, which we call *control* or *activation coverage* [1]. This principle not only applies for simulative verification, but formal property-checkers are also able to compute witness traces.

Mutation coverage, also called *observation coverage,* has been devised in order to not just record activation of code locations, but also to examine whether the functional effect of executing code is checked by a verification suite. Thus, mutation coverage is much deeper than activation coverage. For instance, if a signal assignment is mutated to an inverted value, the line coverage rate will not be affected, but a test-case or formal property which checks function depending on the regular assignment will flag the wrong behavior. If no such test-case is contained in the verification suite, but only ones which just activate the line without checking its execution, the mutation coverage goal for this line is not met. If no testcase even touches the mutated line, this verification hole will already become evident by control coverage measurement. Hence, mutation coverage subsumes control coverage.

In the standard mutation-coverage determination procedure, one pre-selected mutation is enabled at a time in each detection run of that mutation. A fault is classified as detected, if a previously passing simulation test-case or formal property fails with the fault being enabled. Given that each testcase or property only detects few out of all faults, the overall number of qualification runs critically depends on a favorable selection of appropriate pairs of fault * testcase/property in order to minimize the total number of runs without coverage contribution.

B. Control Coverage of Formal Property Sets

While a simulation regression suite yields, either by directed testing or randomization, a collection of explicitly executed traces which can be evaluated with regard to line and mutation coverage, the situation is principally different for formal-property-checking. Once a property has been proven, the verification result is equivalent to a potentially huge number of witness traces, which however normally remain implicit except for one single trace which can be generated on demand by the property checker. While it would be possible to devise a procedure to generate several diverse witnesses, similarly to constrained randomization in simulation, such an approach would be quite inefficient, since the generation of each witness by a formal-property checker is more CPU-intensive than running a comparable test-case scenario with a given input stimulus in a simulator.

C. *Observation Coverage of Formal Property Sets*

A formal property-checker is equipped with powerful algorithms for searching the state space for a sequence refuting the current property's commitments. It collects all mutation coverage of a given originally proven property set from failed formal proofs using the regular highly optimized proof machinery without explicit witness simulation. If a property is proven despite the enabled fault, no witness of it could detect that fault.

Following example illustrates that for formal verification, mutation coverage yielded by formal proof exceeds control coverage obtained by witness simulation. Let's assume that a formal register property has proven the correctness of write-transactions simultaneously for all registers on the regular design[2]. Nevertheless, a single witness will only exercise one write-transaction to one single register, i.e. only the lines related to that transaction are activated, even though the property has many other potential witnesses for other registers which would activate much more design code. In contrast, all inserted faults of all design code relevant for the write transactions of all registers will be detected by formal proofs of the same aggregate register property with different mutations enabled.

D. *Complexity Issues for Mutation Coverage Collection*

Instrumenting a design for mutation coverage analysis significantly increases the complexity of the design model due to a huge number of additional blocks with systematically distorted logic and multiplexors for selectively enabling the mutations, resulting in many more signals and much bigger state space. Consequently, the proof times of properties can become significantly longer. For very big designs with already long run-times for regular property-checks, mutation coverage collection tends to become unfeasible, if no specific measures are installed. For simulation of test-cases on mutated designs, this problem seems to be less severe for an individual qualification run. There the high number of qualification runs is more an issue, which also add up to very long overall qualification times. In general mutation-coverage collection requires many more simulation or proof runs than for executing the complete regression once, since each fault is enabled separately. Even in the ideal case that each fault is directly addressed by the right testcase/property, the number of runs needed is theoretically at least as high as the number of instrumented faults, and in practice much higher, since only some runs contribute detections.

From this follows that performance is a very critical factor especially for mutation coverage computation, and for formal verification also substantially increased model complexity. As these endanger the overall feasibility for big designs, we have tuned our flow based on wide experience from many applications, and added several measures summarized later in this paper.

E. *Overview*

In our verification projects we use the formal property checker from Onespin Solutions, and two independent structural coverage flows. The first one is an on-board feature of Onespin called Quantify[3]. The second, proprietary flow is an integration of Onespin with Certitude[4], which is a mutation-coverage tool from Synopsys. Certitude is routinely applied in our environment for sign-off of simulative and formal verification. For formal qualification we use our own flow as a second flow because Certitude serves as a common basis for simulation-based and formal verification and allows deeper fault instrumentation than Quantify, which is restricted to line-level. Quantify is directly applicable to formal properties proven in Onespin and is able to compute control and observation coverage at line level and identifies excludable redundant and dead code. We normally run Quantify before the deeper Certitude analyses of our proprietary Certitude-based flow. By this redundancy we can also provide the confidence that our proprietary flow is sound, which is also required for ISO 26262 compliance. In fact, the coverage results match well not only regarding covered code, but also the individual coverage contributions of the formal properties.

This paper concentrates on our proprietary formal qualification flow. In order to simplify the usage of our flow and fully encapsulate the set-up of Certitude, we have installed a lot of automatic procedures underneath, which will be summarized in the following sections of this paper. Section II covers the preparations needed before the actual detection proofs are run, as described in Section III. In Section IV, various installed performance improvement measures are sketched. Coverage status extraction and sign-off criteria are topic of Section V. The paper is concluded with summary and discussion of experience.

II. Preparation phase

Before the iterative detection runs can start, which will consume by far most of the qualification time, a few non-trivial preparation steps are needed on Onespin and Certitude level. They have been fully automated in our flow, so that formal-verification engineers do not need any Certitude knowledge or in-depth Onespin expertise. In fact, no set-up and no user-interaction with Certitude is required, neither for the preparation phase, nor for the detection phase, since Certitude is automatically configured and invoked out of Onespin when necessary.

A. *Design Instrumentation*

By default, all HDL-code is selected for instrumentation, except libraries with standard components which are just integrated but are verified and covered elsewhere by separate dedicated library verification. Optionally, the user can specify conveniently code regions to be included or excluded. Internally all Certitude configuration files are automatically generated, which has the advantage that a well-defined-setup is uniformly used for all applications, and the user does not have to bother about how to configure Certitude. Before elaborating the design in Certitude, an analysis is performed which HDL sources are needed for the top-level component chosen for elaboration. Additionally, the minimized set of HDL files is ordered according to the component dependencies.

After generation of the Certitude configuration files, Certitude is automatically invoked for elaborating the design and running the model phase for fault instrumentation and optimization which includes identification of equivalent faults or unreachable faults not to be qualified. These are assigned the status “DisabledByCertitude”

If Quantify has already been run before starting the Certitude flow, the user can select to inherit inclusions and exclusions from Quantify. In this case, redundant and dead code yielded by Quantify is additionally excluded from Certitude instrumentation. Since Onespin is a full-fledged highly performant property checker, its formal algorithms for identifying dead code are naturally much more powerful than those of Certitude.

The Certitude model phase generates instrumented copies of the HDL files of the original design and a Certitude database with the initial states of all instrumented faults. Only those with the initial status “NotYetQualified” will be considered in the detection phase. The initial qualification status is dumped to a file accessible from Onespin, which is used for controlling the detection phase on Onespin side.

B. *Qualification Property Set Selection and Property Instrumentation*

All formal properties which have been proven on the regular design are usable for qualification. Optionally, the user can specify any subset of these, e.g. excluding long-runners, or if only components have been selected for instrumentation, a corresponding property subset can be chosen. There are ways to include other properties later.

In order to be usable for qualification, the qualification properties are automatically enhanced by fault-enabling assumptions which constrain an extra input vector of the instrumented design generated by Certitude. In the constraint, the 0-positions of the fault-enabling vector denote those faults which are disabled, which in the qualification proof will cause the corresponding multiplexors to switch to the regular function. The 1-positions cause faults to be switched on. According to the general qualification procedure where each fault is individually qualified, a globally stable assumption is used that at most one position of the fault-enable vector is := 1. Another global assumption is that the fault-enable vector is assumed to be stable over the complete examination window of the property. An additional assumption updatable in each qualification round specifies for each individual qualification run which fault(s) are allowed to be enabled.

Depending on the property language in which the qualification properties are given, an automatic procedure takes care that the fault-enable assumptions are added in the appropriate way, without user involvement. Our flow currently supports any mixture of properties written as SVA or ITL. ITL (Interval language) is an easy-to-use proprietary language of Onespin with simple temporal constructs, a concept of timepoints which allows convenient specification of synchronization, and HDL syntax, which is especially beneficial for the verification of VHDL designs. Mixed usage of both property languages is supported, both languages allow dependencies on constraints written in the own or the other property language. In our procedure, we generate the global and individual fault-enable constraints in ITL and bind them to each qualification property as needed.

C. *Reading and Compiling Mutated Design and Properties*

First, the instrumented HDL sources generated by Certitude are read, then the design is elaborated and compiled, using basically the same options originally used for the regular design. However, as the instrumentation introduces lots of new signals and inputs, special handling of these is required by the procedure in some cases.

After elaboration and compilation of the design using non-trivial augmentations of the original elaboration and compilation options, clocking scheme, an augmented reset sequence, and input attributes taking into account new inputs of the instrumented model are added.

Finally, the property files are automatically read in the right order according to their dependencies together with the fault-enable constraints mentioned above.

Now a status has been reached where proofs of all qualification properties can be run.

D. *Sanity Checks*

In order to ensure that the qualification properties are actually usable and do not contribute false detections, e.g. due to subtle side effects of the instrumentation and measures against these like avoidance of oscillating signals, all selected properties are sanity-checked. For the sanity checks, all fault-positions of the enable-vector are assumed to be 0. As Onespin has provers which do no start state space traversal from reset but super-sets of the reachable state space, we have installed a strategy to split the qualification property set into two subsets, the first for provers searching the complete state space, and the second checked by other provers from reset. These 2 groups are kept separate for the sanity checks and the entire detection phase.

During the sanity checks, statistics are recorded which later guide property selections in the detection phase.

E. *Usage*

One simple user command entered in Onespin's tcl-shell in the module verification mode triggers the automatic execution of all steps of the preparation phase, in the following example with default parameter values which have been chosen in a way that they are normally appropriate:

```
mv> cqm
```

As first parameter, this tcl-procedure accepts a property list which can have wild-carded elements, followed by lists of include and exclude patterns for specifying code regions of HDL-files, which may contain wild-cards:

```
mv> cqm {sfr__w* sfr__r*} {*/ip_*-*} {*sync* *sff*}
```

In above example, register-write and -read properties will be used for qualification, only hdl-files matching the first pattern list are included in instrumentation and from those only those not matching the exclusion patterns.

When this procedure has been fully executed, the detection phase can be started as described in the next section.

III. Detection Phase

The detection phase comprises a potentially large number of qualification rounds. If the user sees a need for intervention, the qualification can be aborted at any time without any loss of intermediate information, so the qualification can be directly continued from the status reached before. For instance, if the LSF-host on which the qualification was started gets an overload problem caused by other LSF-jobs, the qualification can be continued on a different host.

A. *Configuration*

A configuration is automatically generated containing default settings of several parameters which control the qualification rounds like run-time and memory limits for proof jobs, maximum number of parallel proofs, the maximum number of rounds, and others. This configuration file can be updated by the user at any time and the new settings will become effective when the current round is finished and the next is going to start. The user can for instance decide to first use fast-running properties only by setting the time limit so low that long-runners are excluded at the beginning, even though the property selection heuristics automatically take into accounts run-times.

B. *Initialization*

Whenever the command for triggering the qualification rounds is entered, the current detection status is evaluated for determining the actual target fault subset from the intersection of the fault set specified by the user (by default all), and those faults not yet detected in previous qualification rounds. As the procedure also determines which of the user specified-properties are not able to detect any of the initially targeted faults, the qualification property set is internally reduced accordingly, in order to avoid useless qualification proofs.

C. *Property Selection*

In each round a subset of the current qualification property set is heuristically selected, based on previous qualification proof run-times and membership in one of the two prover groups specified above (Sect. II.B). The qualification results can only be evaluated when the last proof of the current round has finished. Therefore, it is often better to not use the maximum configured parallelization, since one long-running proof can block evaluation of detection results from fast-running properties and start of a new qualification round with these, which would yield more coverage than if the long-runner proceeds. The selection algorithm takes this effect into account. Given also that the run-times of proofs can deviate pretty much from previous proof times, it is obvious that such heuristic decisions are better taken on the side of the property checker than by Certitude, which cannot know such specifics.

D. *Qualification Proofs*

Before the qualification proofs are started with the currently selected properties, the fault-enabling assumption is updated according to the current target fault set. Then the qualification proofs of the current round are startedon the configured maximum number of LSF-jobs. In the background, a watchdog LSF-job is started in parallel which observes the status of the proof jobs and may take a heuristic decision to kill still active long-running proof jobs if a certain number of short-runners have already terminated.

E. *Proof Result Evaluation*

When the qualification proofs have finished, the detected errors are collected from each failed property. Properties with hold-result are proven to be unable to detect any of the currently not yet qualified target faults.

Therefore, these properties are removed from the current qualification property set. Nevertheless, when later a new series of qualification rounds is started with a different target fault set, these properties can be re-included.

Given-up properties due to run-time reasons are collected separately and deprioritized in next rounds, until run-times get higher than the time until abortion of these properties.

All information about detections, run-times, abortions, and hold-results is recorded and evaluated for the next round and for the decision whether to continue the qualification rounds with the current configuration. New qualification rounds can be started with a different target fault or qualification property sets, or resource limits.

F. *Usage*

Again, one simple user command with favorable default parameters is sufficient to start qualification:

```
mv> cqd; # run qualification targeting all currently not yet qualified faults with all qualification properties
```

Optionally, the user can specify target fault and qualification property subsets in a flexible way:

```
mv> cqd {1..10,8000..9000} {*all* *sync*}; # target faults in ranges, use properties matching *all*, …

mv> cqd [list [topEntityFile] [topEntityFile]]; # target not yet qualified faults for top-level component

mv> cqd "TopOutputs*;ResetConditionTrue”; # target faults of first category, then of second
```

G. *Filling Coverage Holes*

Remaining undetected faults require properties to be added. Once these are verified, they can be used for qualification. For this purpose, an automatic command triggers sanity checks for these and in case of success adds them to the qualification property set. The command for adding newly proven properties is used as follows:

mv> cqa $new_props;

If *cqa* is called without argument, just all properties with hold-status not yet contained in the qualification property set are added. This is also the case if in the first *cqm* call or subsequent *cqa* calls not all proven properties were included, e.g. because of comparably long run-times.

IV. Measures for Performance improvement

In this section various measures for coping with performance issues in qualification proofs are shown.

A. *Prover selection*

Handling properties of the first group (Sect. II.B) separately yields considerable performance advantages, because the provers not starting from reset normally come to faster decisions. Take for instance a property which proves that an overflow signal is set when a 24-bit counter which starts from 0 at reset exceeds the maximum value. If a fault for flipping the most-significant bit of the counter is activated, the prover originally proving the regular counter behavior will fail after short run-time. If however, one of the provers optimized for generating counterexamples from reset would try to generate a reachable-fail result, it would need a huge number of cycles far beyond the manageable length of examination windows for bounded model-checking. For the same reason it is not possible for such properties in the regular design to generate witness traces from reset.

B. *Model trimming*

As said above, proof run-times also depend on the size of the design model. Thus it is worthwhile to spend some thought on reducing the model size. Onespin provides a feature that input signals can be assigned a signal domain during compilation of the model. We sometimes use this feature e.g. for constraining an input which switches between scan- and normal-mode statically to 0, if our verification addresses regular function only, but not specific behavior in scan-mode. While we could assume scan-mode = 0 individually in assumptions of functional properties, it is more efficient to reduce the design model statically during compilation and trim away the logic for scan-mode.

In our mutation-coverage flow, we have added a feature for applying model-trimming automatically to reduce the complexity of the design model if only a subset of all faults is targeted so that the multiplexor and corresponding mutated logic in the fan-out cone of fault-enable input bits tied to 0 is pruned away during compilation.

Since, however, compilation and model-trimming itself creates varying overhead, model-trimming is not triggered in each round, but only when a configurable percentage of faults have been qualified since the last time. By default, model-trimming is started whenever 20 new percent of the original fault set have been detected. If compilation times for model-trimming are short, the user can decide to configure more frequent model-trimming.

C. *Focusing*

If property subsets refer to different top-level components of the compete module, it may help to qualify faults with the corresponding property sets. The model-trimming feature additionally accelerate the qualification rounds.

D. *Super-Parallelization*

A first level of parallelization is already available in the regular procedure by submitting parallel qualification proof jobs for several properties in each qualification round.

If qualification results cannot be expected to be available as fast as required by the project, a second level of parallelization can be introduced by starting simultaneous qualification sessions and run qualification rounds independently on disjunct fault subsets in order to avoid overlapping qualification results. For this purpose, a simple fully automated user command is available:

mv> cqdp 3 0 1; # open 3 new Onespin gui sessions and run cqd-command in each

mv> cqdp 8 1 0; # keep the running sessions alive, open 8 additional Onespin batch sessions

The command automatically sets up the qualification sub-directories and starts independent qualification rounds in parallel. As the fault subsets are disjunct, which are automatically re-partitioned if sessions are added, a full speed-up by a factor *n* is obtained by *n* parallel qualification sessions. As the target fault set in each individual session is accordingly smaller, model-trimming here again yields additional acceleration.

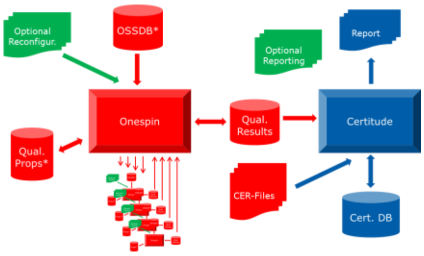

Figure 1: Parallel Qualification Sub-Sessions

E. *Merging Qualification Results*

Results of separate qualifications can be merged by further automatic commands:

1. Results of parallel sub-sessions from defined sub-directories generated by the cqdp-command are merged. As all detected faults refer to the same Certitude, the fault detections are directly merged as well as the hold-results of detection proofs for fault-subsets.
   For this purpose, the *cqdm*-command is available can be used without arguments and accesses the qualification results from all sub-directories in the parent qualification directory.

2. Qualification results from Certitude databases with different fault instrumentations, but for the same design with same generic parameter sets can be merged. The corresponding command takes a list of Certitude databases with directory paths as follows:
   mv> mcq [list $dir_0/certitudeDB … $dir_n/certitudeDB]
   As a result, a combined Certitude database is obtained with an instrumentation of the union of all faults previously instrumented and a union of all qualification properties used in all databases. For this purpose, first a new united instrumentation is generated, then the faults of the separate qualifications are mapped to the differently numbered faults of the partial instrumentations.

The second approach can make sense if the verification is split into parts with higher and less priority. For instance, the design may contain generated HDL-code which is likely to be correct by construction. Verification and qualification can then first be concentrated on hand-written code, and if the project timeline allows further qualification effort to be spent on the generated code which can then be addressed by separate instrumentation. Moreover, this second approach also fits with a methodology jointly applying formal and simulative verification. The resulting separate Certitude databases of the same design can then be merged to one joint database.

F. *Inheritance*

In a typical project, design changes can well happen until tape-out due to new change requests when verification and qualification have been almost finalized. While the verification regression will just require minor adjustments in case of local design changes, previous qualification results are invalidated in the first place. However, just starting the qualification from scratch would be a waste of resources. We have therefore added a specific inheritance approach which works as follows.

The preparation phase is fully executed like before on the new design version, but takes just a small fraction of the previous qualification time. The detection phase is organized differently. As the fault numbering will have changed, a fault-mapping is performed which relates faults from previous and new design version. From the previous detection results, pairs of properties with the new fault numbers mapped from the previous ones are formed. These are then used for directed qualification, which involves full qualification proofs; thus, this approach is safe. As the likelihood that same properties identically detect mapped faults is very high for most faults, the previous qualification status will be attained much faster than if the complete qualification starts from scratch. After directed qualification, a few previously detected faults may be left which can be addressed together with completely new or unmapped faults for regular qualification. In total, this re-use approach can help to save a lot of run-time. It is used by just entering the *cqd* command with a Certitude database path as argument.

V. COVERAGE STATUS

Since all qualification results are automatically collected in the background on Onespin side, possibly distributed over different qualification directories, it is desirable to take from time to time a snapshot of the overall current qualification status. For this purpose, a command at linux-level is available which traces all files containing qualification results obtained on Onespin-side, combines these and automatically invokes Certitude to generate a standard report of the current detection status which can be shown to project management.

The overall current coverage status is inspected with respect to the fulfillment of sign-off criteria defined for the project. In contrast to the regular Certitude flow used for simulation, where fault-activation is used as a preliminary analysis before the detection phase in order to reduce the set of fault-test-case combinations potentially going to be checked in the detection phase, our Onespin-Certitude flow does not need such pre-analyses, and they would not be meaningful for the reasons discussed in Section I.B. Since formal verification includes white box-properties, which involve internal signals rather than exclusively global inputs and outputs, we do not need to identify and sieve out faults which cannot propagate to the outputs of the module under verification.

For sign-off, we require 100% detection of the faults in all instrumented reachable and non-redundant code in the initial *NotYetQualified*-set yielded by the Certitude-optimizer, with commented exceptions being accepted only after individual risk analyses.

VI. CONCLUSIONS

Experience from applications of previous flow versions[1] has led to continuous enhancements which have improved the applicability to a wider range of applications.

Compared to the early days of formal-property-checking being used for the verification of productive hardware designs, the state-of-the-art of verification completeness assessment has substantially developed, resulting in better convergence and predictability.

The number, size and diversity of applications for the Onespin-Certitude flow has significantly increased since the first projects. Meanwhile not only VHDL-designs, but pure SystemVerilog modules or mixed designs with SystemVerilog, Verilog, and VHDL sub-components have been successfully processed, which required some enhancements of our flow. Applications with several 10k instrumented faults and several 100 properties are handled by the flow in the range of days to few weeks. With the option to partition the qualification and later merge the results, we are confident that all designs formally verifiable can be processed by our mutation-coverage flow.

While it is becoming more difficult to hire qualified verification engineers, growing design and EDA tool complexity, increasing cost and time-to-market pressure, and at the same time tight quality requirements for automotive products developed according to the ISO26262 standard are severe challenges, which can only be met by systematic and highly automated verification methodologies and easy-to-use qualification flows.

With the measures presented in this paper, we are in a better position to master these challenges and extend the scope of formal verification applications.

REFERENCES

[1] H.Busch,"Qualification of Formal Properties for Productive Automotive Microcontroller Verification", in Proc. of DVCON 2013.

[2] H.Busch, "Generation of Complete Aggregate Formal Properties", Proc. of DVCON 2008 San Jose, 2008.

[3] Onespin documentation, https://www.onespin.com/quantify, 2022.

[4] Certitude datasheet, https://www.synopsys.com/cgi-bin/verification/dsdla/pdfr1.cgi?file=certitude-ds.pdf, 2022.