Qualification of Formal Properties for Productive Automotive Microcontroller Verification

Holger Busch
Automotive Microcontroller Division
Infineon Technologies AG, Neubiberg, Germany
Tel.: +49 89 234-44307
holger.busch@infineon.com

Abstract—**Industrial automotive microcontroller design requires most advanced verification methodologies for meeting highest quality and safety standards despite very tight project schedules. For efficient verification project management, appropriate metrics for measuring verification progress and completion are essential. As modern verification environments are heterogeneous and include directed and constraint-driven simulation and formal property-checking being even jointly applied to individual modules, common metrics are needed which allow coverage which is contributed by different verification approaches to be merged. While structural coverage figures delivered by simulation are generally well understood, project managers desire compatible sign-off criteria for assessing verification results yielded by formal property checking. This paper describes experience and results from the qualification of proven formal properties of modules of Infineon's new automotive microcontroller family AURIX™. In this next-generation chip design project, metrics for formal property coverage which enable code-based progress tracking and merging with structural coverage yielded by simulation test-cases have been applied for the first time productively. Two alternative methods for measuring coverage of formal properties have been used. The vast number of formal quantification proofs needed for larger modules turned out to be the real challenge for both methods.**

This work contributes to the project RELY, 01M3091A, which is funded by the German ministry of education and research.

I. INTRODUCTION

Formal property checking has been in established in industrial chip design projects for many years. It is widely acknowledged that formal property-checking ensures highest quality of RTL modules and yields very convincing bug-finding statistics. The advances of formal property checking technology have rendered possible exhaustive verification of modules of several 10,000 lines of RTL code.

For the planning of schedules and control of project milestones project managers request easily understandable and applicable metrics which reflect the verification progress in percentages and allow for basic quality control of globally distributed verification work packages, regardless of which verification approach, formal or simulation-based, is applied to different parts of the system-on-chip architecture. Less than 100% coverage of a module's functionality by one verification approach is acceptable, if a complementary verification approach contributes the missing coverage. For this goal, it is necessary to apply common metrics for measuring the quality of verification results.

Code-oriented metrics like line, branch or toggle coverage are widely applied as minimum criteria for a design to be sufficiently simulated. In productive projects, the verification must not be finished before these basic coverage criteria are fulfilled in addition to functional completeness criteria specified in verification plans.

Two new approaches for generating coverage metrics more directly related to the established simulative code-coverage metrics have been tested on system-related modules of new automotive microcontroller developments. The first method uses a new feature provided by Onespin-Solutions called Quantify, which computes formal line and branch coverage of formal property sets. The second approach is based on the test-bench qualification tool Certitude from Synopsys.

Like in other formal coverage approaches, formal property checking is performed in order to determine the coverage contributions of the properties. Unfortunately, the qualification checks in total consume considerably more run time than the regular regression run with all properties, as thousands of different punctual design mutations and re-checking of property sets are iteratively performed. Typically many of these proofs do not yield any coverage contribution, whenever properties are checked which are not affected by the current design mutation.

Without dedicated methodologies and routines for optimizing the iterations with respect to the elapsed run-time and the results yielded in each round, such qualification of formal property sets would only be feasible on small designs. The tool providers have already installed significant improvements since the first experience gathered with these formal qualification approaches. Nevertheless, in several

applications with real chip components, further potential for optimizations has been identified.

Both methods have been applied to system control modules of the new Infineon Microcontroller family AURIX™ (**AU**tomotive **R**ealtime **I**ntegrated Ne**X**t Generation Architecture) which comprises up to 3 independent 32-Bit TriCore™ CPUs, numerous peripherals and memories of several MB and which is designed to meet highest safety standards and performance requirements. In principle both approaches showed their capability of detecting RTL code left uncovered by formal properties and of delivering coverage data which resemble those obtained by simulations. As the formally verified modules have specifications of several 100 pages, several 10 k lines of RTL code, and hundreds of properties, straightforward usage of the basic qualification methods would have resulted in unrealistic run times caused by a vast number of required coverage proofs. Various enhancements of the qualification flows have resulted in considerable efficiency gains.

The following sections of this paper are organized as follows. Section II provides some information on the used tools and reviews related work. Section III summarizes the basics of the two applied quantification approaches. Section IV describes our qualification methodology based on OneSpin's Quantify feature. Section V details our integrated formal qualification flow based on Certitude. Section VI adds experience and results achieved by applying both approaches to system control modules. Section VII discusses conclusions and implications.

II. BACKGROUND

In this section, qualification methodologies are summarized, including the EDA tools used.

A. Ingredients of formal coverage methodologies

Code coverage computation of properties proven by a formal property checker requires a facility for mutating the design in such a way that each individual mutation is related to another piece of code to be covered. Such mutation is either instrumented explicitly in the RTL design, or in an internal model of the design. Obviously, an instrumented RTL design can be read by any verification tool, whereas the internal model is not usable outside of the formal verification tool. The instrumentation can be done once for the complete qualification, if the mutation of each code location is individually activated by an assumption added to the current property being checked.

Some selection mechanism has to determine in each round which property is checked with which activated mutation. This decision is very essential for the overall efficiency. The goal of each individual coverage proof is to check whether a property covers a specific mutation related to a code location, e.g. a signal assignment or branching condition. For simplicity we here assume just 1 mutation per code location. In general the actual instrumentation can generate several mutations for one single code location. A code location is considered to be fully covered if all related mutations are detected by at least one property each. Thus the key to improved efficiency is to optimize the sequencing of these pairings, based on the fact that once a mutation or fault is covered by one property, it need not be checked with others.

This is illustrated by a simple calculation. For a module with 10,000 code locations to be covered, 100 properties and an average proof time of 5 minutes per property, an overall qualification time of 5 million minutes or almost 10 years would be needed for checking all combinations, under the very pessimistic assumption that for each location the covering property is selected as last one. The theoretical minimum would then be 10,000 * 5 minutes ≈ 34 days in the best case, if each code location were directly addressed by the right property and a complete property set were available.

In reality, a code location is often coverable by several properties. On average only 50% of all properties may have to be checked until a code location becomes qualified, resulting in 5 years worst-case run-time. However, incomplete property sets will run exhaustively on code locations not covered by any property.

A book-keeping facility for the results of all proofs is required in order to collect information about covered code and coverage holes, and to prepare the selection of mutations and properties and the iterations. The same formal property checker that has already been used for digital design verification of a specified property set now serves for running the checks to be performed during formal property qualification.

The decisive feasibility question is whether the formal property qualification methodologies can be tuned such that they scale up to the size of realistic module designs being qualifiable in a few days.

B. Onespin's Module Verifier MV 360°

Onespin's module verifier MV 360° has been applied to numerous industrial chips designs[7]. MV 360° is a state-of-the-art property checker which accepts properties in 3 different property languages:

- System Verilog Assertions
- PSL Assertions
- ITL (InTerval Logic, Onespin's proprietary property language) properties with VHDL or Verilog flavor

In one session, property sets written in ITL may coexist with sets of SVA and PSL assertions, and environment constraints may be mixed as well. For instance, it is possible to write ITL properties which not only depend on ITL constraints, but also on SVA or PSL assume-properties. Thus sets of module interface constraints can be used for formal module verification and in SOC simulations for checking whether environment assumptions are kept.

The Onespin tool suite comprises HDL front-ends for VHDL (87, 93, 2008), Verilog and SystemVerilog.

Several different proof engines are available which are optimized for different purposes. For instance, one engine is specialized on generating countertraces reachable from reset. For this purpose, user-defined reset sequences can be entered, if a default reset sequence is not applicable. Another engine proves properties from any start state, however may return unreachable countertraces. When a property check is invoked, the user can specify a set of proof engines to be tried in parallel or sequentially.

Like other commercial property checkers, Onespin offers a formal consistency checker which identifies dead code, in addition redundant code, sticky signals, checks in-code assertions extracted from the design and several other automatically generated assertions. Most consistency checking results are relevant for property set qualification. For instance, mutations within proven dead-code will not be detected, and stuck-at-1 fault of a signal already proven to be constantly 1 need not be considered. The parallelization feature is especially important for running regressions. It distributes proof jobs of property sets to different servers in lsf (load sharing facility) queues. Onespin's extensive debugging features are very helpful for regular property checking, but are not needed for qualifying properties.

Onespin itself supports model mutation by allowing arbitrary port and internal signals at any level of hierarchy to be cut from their fan-in cone. As a result, such signals are split into an external output driven by the original signal's fan-in function and an external input which drives the signal's fan-out. In added property assumptions, any behavior of the resulting artificial inputs can be assumed, including the normal signal behavior if input and output parts are just connected. This special feature e.g. serves for safety [10] and security verification, and can also be used for property qualification, e.g. by measuring the detection rate of injected stuck-at faults of all signals or registers. In a similar way, Certitude is usable for fault injection in safety verification.

Onespin MV 360° comprises a formal completeness checker which applies the strongest possible criteria for *gap-free verification*[1]. These criteria are not compatible with coverage metrics used in simulation environments. A strictly structured property suite of formal operation properties has to be written in order to run formal completeness checking, which is not always feasible within project schedules.

Therefore another feature called Quantify was added which performs coverage analyses closer to the classical notions used in simulators (cf. Section III b).

As the Onespin GUI comprises a TCL-shell and a rich library of useful TCL utilities allowing evaluation and control of internal data such as filtered signal lists, property and constraint lists, or current proof status of properties, users can well add own TCL functions for their purposes.

C. Certitude

Certitude is a tool for qualifying simulation test-benches offered by Synopsys.

Certitude instruments an RTL design under verification with artificial faults which, when activated, block or modify pieces of the original RTL code, thus distorting design behavior. A reduced example is shown in **Example1**. An added fault-vector (f) selectively activates different mutations of the code. If all bits of this vector are cleared, i.e. no fault is active, normal behavior results.

```
entity ex1 is
port(a_i, x_i: in bit;
       …);

architecture rtl of ex1 is
  …
  begin
…
  if x_i =1 then
    ff <= a_i;
  end if;
  …

After instrumentation:
entity ex1 is
port(
 f: in bit_vector(1 to n);
 a_i, x_i: in bit;
…);
architecture rtl of ex1 is
…
begin
 if f(1) = 1 then
   if false then
     ff <= a_i;
   end if;
 elsif f(2) = 1 then
    if true then
     ff <= a_i;
   end if;

elsif f(3) = 1 then
  if x_i =1 then
    ff <= not a_i;
  end if;
  …
else
  if x_i =1 then
    ff <= a_i;
  end if;
end if;
```

Example 1: RTL code instrumentation by Certitude

Test-cases are then rerun on the mutant code with faults selectively activated by Certitude. For each injected fault and each test-case, Certitude returns four possible results:

1. A fault is not activated by any test case, i.e. the code in which the fault is injected is not covered by any test-case.
2. A fault is not propagated by any test-case: the behavioral fault does not have any externally visible effect, e.g. at the interface of the module or a scoreboard of the test-bench.
3. An activated and propagated fault is not detected by any test-case.
4. A fault is activated, propagated and detected.

Certitude does pre-qualification in cooperation with simulators: Faults which are not activated and propagated will not be checked for detection during qualification. For each fault, Certitude keeps the information by which test-case it is activated and propagated. Later in the most expensive detection phase, this information is used for avoiding unwinnable simulation runs with test-cases without chance to detect the fault.

The notion of fault detection is defined as follows: If a normally passing test-case fails with the activated fault, this test-case has detected the fault, otherwise it has not. If a specific fault is not detected by any test-case, the test-bench

has to be augmented accordingly by adding or enhancing test-cases. During the detection phase, Certitude iteratively generates test-case-fault combinations to be transmitted to the verification tool. Certitude contains heuristics which analyze the results of previous verification runs and optimize the selection of test-case – fault pairs to be checked next. As subsets of all faults are automatically identified to be likely to be covered by the same test-case, e.g. faults in nested case constructs, the chances of successful qualification are significantly increased.

Further fault reduction is achieved by dropping faults in sub-branches, if a fault in a super-branch has not been detected by any test-case. **Example 1** shows hierarchically nested faults. If activation of fault 1 and fault 2 do not cause any test-case to fail, fault 3 cannot be detected either, thus fault 3 will be dropped. Faults are automatically categorized and can be filtered by the user. The depth of fault insertion in the hierarchy of conditional expressions can be controlled as well, which allows a shallower qualification in less run-time.

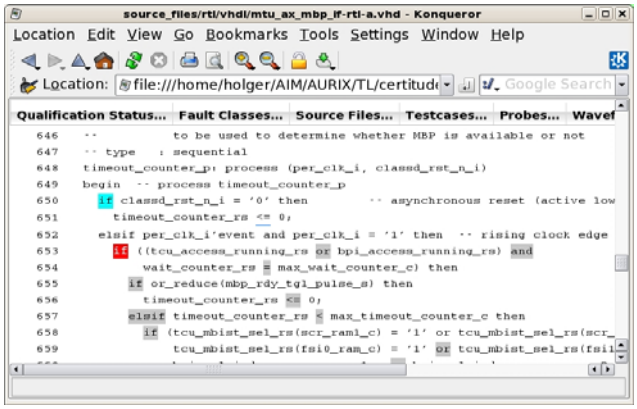

Figure 1: Viewing detection status in RTL code

At each stage of the qualification, a qualification report can be generated on user request in HTML or a proprietary viewing format, with the source code items being highlighted in different colors according to the individual qualification status. Detected and non-detected faults can be inspected in more detail as shown in Fig. 2.

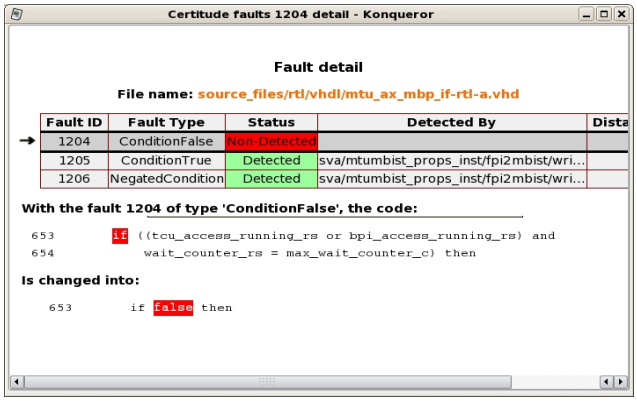

Figure 2: Detailed information about fault status

The qualification result yielded by Certitude is stronger than code or branch coverage. 100% code coverage just means that each code location has been touched, but not necessarily that the behavioral effect of each code line has actually been checked by at least one test-case. In contrast, 100 % Certitude coverage means that each behavioral fault is detected by the test-suite.

D. Alternative Approaches

A simple approach is to generate a witness trace of a property and put this into ordinary coverage checks performed for simulation traces. As the generated witness contains values chosen randomly, such coverage analysis is very weak. For instance, any trace will be a valid witness of the trivial property |- true. By way of realistic constraints, the quality of the witnesses can be improved.

Another basic solution is to compute the required proof radius for bounded-model-checking properties in order to determine the lines of code which can be covered within a limited number of clock cycles according to the proof radius. If some code location is not reachable after n clock cycles, no bounded property within a proof radius n will cover that code location. In a wider sense, this coverage notion is related to fault activation analysis in Certitude. However, a proof radius check includes no analysis whether reachable code is actually checked by one or more properties.

Various approaches have been devised which address the question whether a set of properties specifying a digital design is complete [1,3,4,5,6,9]. The completeness notions applied there, even though formally very strong, are not compatible with simulation coverage metrics. Approaches for formal property coverage must provide valuable information about the quality of the properties by observing the behavior of code locations and not just their activation. In an industrial environment, tools must manage the complexities of real chip designs, must be relatively easy to use, and must fit into verification landscapes where simulation-based methodologies are applied [8]. Moreover, a certain level of maturity allowing smooth integration in productive design flows must have been attained.

III. TWO BASIC QUANTIFICATION APPROACHES

In this section, 2 alternative methods for qualifying formal property sets are discussed. First, Onespin's Quantify feature is summarized, then our basic flow for the usage of Certitude with 360° MV shown. Thus we use Certitude not only for the qualification of simulation test-benches, but also for assessing the fault coverage of formal properties. Both alternatives have been used on different system control modules of the AURIX architecture.

A. Onespin's metric-driven verification approach

Onespin's new Quantify feature is closely integrated into the regular property checking environment of 360° MV. Quantify is a fully automatic push-button tcl-function to be directly called from the Onespin shell. Quantify just reads a

set of properties for qualification and then analyzes without any need for further user interaction which code locations are covered. In a preliminary phase, dead code and redundant code lines are identified and excluded from qualification. Constraints e.g. given as SVA-assume–properties are additionally taken into account, so that further code not reachable due to constraints is identified before the qualification starts. For each property a witness is generated and simulated. This simulation phase allows code locations to be identified which are reached or not reached by the provided properties. This information is used in subsequent coverage evaluations.

In the actual qualification phase, the tool iteratively checks for each code location whether it is covered by at least one property or completely uncovered. The qualification result of a code location does not depend on the syntactical representation of a property, thus the user does not need to restructure properties.

By specifying a maximum effort level when starting qualification, the user can control how deep the coverage checks shall be. With a lower effort level the qualification finishes faster, however, with more code regions remaining open (= unqualified). The tool performs the first iteration cycle at lowest effort level, trying to cover as much code as possible with minimum CPU time. With each new iteration cycle, the effort level is automatically increased, until finally the user-specified maximum is reached. Accordingly the checking times grow.

While the qualification is running, the current status is permanently visualized in HTML format for the code of each module along with statistics about covered lines and branches, as shown in Figures 3 and 4.

Figure 3 Quantify: Coverage Statistics

At each effort level, all code locations still open are checked one after the other with each property of an internally selected subset, until the first property check results in coverage of the current code location. Thus in particular the "uncovered" result is yielded only after a maximum accumulated proof time for all properties which corresponds to the run time for a complete regression for just that code location, which is only reduced by potentially premature giving-up at lower effort levels with the effect that properties with low run times will contribute coverage right from the beginning, while properties with long runtimes will later get a chance to detect covered code locations at high effort level.

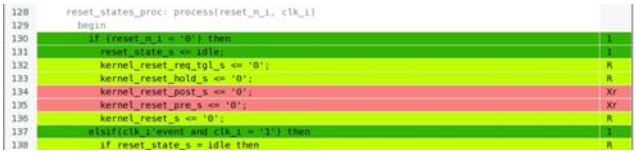

Figure 4: Highlighting of RTL code depending on status

The Quantify feature can be used incrementally. Results of one run are saved regularly while Quantify is active, and a new Quantify run is able to use previous results. As Quantify typically runs several days for designs with several thousand code locations, some interruption e.g. by network instability might well occur. Thus saving intermediate results is crucial for avoiding that a qualification has to re-start from scratch. Simulation coverage can be loaded in addition.

B. Qualification of Formal Properties Using Certitude

As Certitude generates just ordinary HDL code, it is possible to feed the instrumented VHDL design into the HDL-front-end of a formal property checking environment. The formal properties used for qualification then take over the role of the simulation test-cases. Where normally Certitude starts a simulator with a test-case x fault pair, the modified Certitude scripts now invoke the property checker with a property x fault pair instead.

If the property fails with an activated fault, the property checker reports the result back to Certitude which interprets the result as detected fault. The communication between Certitude and the property checker requires some specific scripting on part of the property checker. When the property checker receives a job from Certitude, a script generates a corresponding constraint that the current fault is active and all other faults are inactive and then starts the proof of the selected property. If the proof of the property fails, this result is interpreted as successful detection of the current fault. The overhead for generating the constraint, invoking the property checker, reloading model and property, analyzing the proof result and reporting it back to Certitude is not negligible.

IV. QUANTIFY METHODOLOGY

Even if the efficiency of the fairly new Quantify feature has been significantly improved since the first release, sets of several hundred properties of modules with more than 30,000 lines of RTL code are hardly manageable without specific measures to be taken by the verification engineer.

Onespin's quantification feature is closed and can only be controlled by setting the parameters when it is invoked. The internal strategies are not disclosed to the user. It can be assumed without being documented that Onespin internally takes measures to optimize the sequence of checks.

The Quantify function reads optional user parameters for selecting not only complete modules but also individual line ranges for inclusion or exclusion of code regions from qualification runs. These parameters allow the user to precisely address code partitions by selected property sets. The verification engineer who devised the properties has a good understanding of the subsets which verify different sub-components of the top-level architecture. Thus it is not too difficult to focus each qualification on related code regions in a separate run.

Partitioning of property sets and designs can have a substantial effect, as sketched by the following simple calculation. If the idealized module discussed in Section II A is partitioned into 10 sub-components, with corresponding property sets of 10 properties each, the worst-case figures are significantly improved: Each code partition now needs 35 days, which amounts to about 1 year for 10 partitions qualified sequentially. This reduction by a factor of 10 is already achieved without further measures.

The proof complexity of properties also influences the efficiency in qualification. Very complex properties with high sequential depth cause long run-times, which may even be longer with code mutations. Generally, long-runners of one or more hours tend to jeopardize the efficiency of the qualification. Thus they are better excluded from the first qualification runs or split into sub-properties. The quantification proofs use the same proof infrastructure like ordinary property checking, which can be controlled by user-definable check options. In particular, parallelization and run time limits can be configured in order to optimize the suite of qualification proofs and interrupt long-running proofs.

Aggregate properties, which combine properties of similar instances, are advantageous if their check-times are close to the proofs of the instances. In [2], we showed that aggregate register properties behave like this in Onespin. For instance, a write property of a single register is proven in roughly the same time as a compound property verifying writes to all registers in one single proof. Such an aggregate register property covers many code locations. For one module with many configuration registers, just 5 aggregate register properties like read, sw-write, reset, no-write, and hw-update have covered 70 % of all code locations.

In summary, our methodology using Quantify is based on

- Partitioning of formal qualification according to property subsets for different code regions,
- Prioritizing properties according to their run-time with limit setting in order to avoid long-running proofs in earlier qualification phases
- Aggregate properties which improve the hit-rate and reduce the number of properties.

Additionally, parallelization is configured according to the number of available servers and licenses.

V. ENHANCED CERTITUDE METHODOLOGY

The basic efficiency problem is the same as for Quantify. The main goal is again to optimize the order of the checks, where the faults instrumented by Certitude roughly take the role of the code locations in Quantify. As there is no direct connection between Certitude and Onespin's internal infrastructure, the overhead for the communication between both tools is a topic deserving special care.

The measures described in this section address the optimization of the sequence of checks and of set-up times when proof tasks are forwarded to the property checker.

A. *Improved Tool Interaction*

In a basic implementation of the link between both tools, several steps are performed when a qualification proof task is generated by Certitude and sent to the property checker:

1. Start the proof environment.
2. Load the (instrumented) design.
3. Translate the selected faults into assumptions in the property language.
4. Add these assumptions to the selected property.
5. Load property code.
6. Run the selected property checks.
7. Evaluate and report the proof results.

Except for Step 3 and 4, these steps are the same as in regular regression runs. In contrast to regressions which just have to be started once, for qualification Steps 1-7 are repeated every time Certitude orders new checks. Thus the overhead for Steps 1, 2, and 5 is significant.

Step 2 can be optimized by using Onespin's feature for saving the elaborated model of the design and properties in a database. When Onespin is invoked again, the data-base is just re-loaded. Thus Step 5 is reduced as well, if the modifications are encapsulated such that only modified property code is re-read. Another step is appended for saving the database when a checking round is finished, which is not blocking as the results have been fed-back to Certitude.

2. Load data-base.
5. Load property modifications only.
8. Save design and property data in database.

By way of a specific control (TCL) routine, Steps 1, 2 and 8 are saved completely:

```
proc certQualCheck {
   while {[waitforCertReady]} {}
   while{[qualify]} {
      set cfault [readCertFault]
      set cprop [readCertProp]
      updCertConstr [list $cfault $cprop]
      checkProperty $cprop
      reportCertRes $cfault $cprop
      while {[waitforCertReady]} {}  } }
```

This routine certQualCheck is called in the property checking shell, once the design is read and elaborated and the properties are loaded. The subroutine waitforCertReady then checks whether Certitude has notified that it has evaluated previous results and has taken the decision to

finish or continue the qualification with a new task. Subroutine qualify checks whether a new task with fault x property pair has been received from Certitude. Subroutine updCertConstr updates the constraint for activating the fault currently selected by Certitude and adds it to the assumption list of the selected properties. Subroutine reportCertRes evaluates the results of the property check and writes them out in a format which is processed by a Certitude script.

In this way, the overhead for each check is minimized, as Onespin is not re-started, and design and properties are not re-loaded in each round. The saving of this measure can be in the order of magnitude of a property check.

B. Parallelization

Certitude can be configured to select several fault x property pairs in each round. The routine certQualCheck need not be modified for this purpose, as all subroutines are able to handle lists of faults and properties. If the sizes of these lists are bigger than the number of available property-checking servers or licenses, it makes sense to augment the routine certQualCheckPar with own selection mechanisms which optimize the throughput of the property checking phase. Subroutine selFPs determines an optimal combination of properties to be checked together. For this purpose it accesses internal information about previous run-times, previously successful proof engines for each property, and currently available servers and licenses. Without using latter information, the chosen degree of parallelization could be sub-optimal in an environment with limited resources.

```
proc certQualCheckPar {
   while {[waitforCertReady]} {}
   while{[qualify]} {
      set cfaults [readCertFault]
      set cprops [readCertProp]
      set props $cprops
      while  {$props != {}} {
         set fsprops [selFPs $props $cprops $cfaults]
         updCertConstr $fsprops
         set sprops [getProps $fsprops]
         checkProperty $sprops
         set props [evalProps $fsprops]    }
      reportCertRes $cfaults $cprops
      while {[waitforCertReady]} {}  }  }
```

When proofs of properties are started, a limit can be specified for aborting all proofs hitting this limit. Generally, the proof times during qualification cannot be forecasted, they vary significantly for different mutations and differ from regression run times. Thus premature abortion of all properties might occur, so that the complete run-time spent until then would have been wasted without any coverage having been obtained. We therefore added an automatic abortion facility which runs in parallel to the property checks, controlling their run-times, and cancelling proofs from outside only if they run much longer than the others.

C. Additional Selection of Faults

In addition to the selections delivered by Certitude, own heuristics allow the set of fault-property pairs checked in each round to be augmented. For instance, if an advantageous partitioning of the design and corresponding property subsets, and additionally a mapping between Certitude faults and code lines has been determined, another subroutine generates additional fault x property combinations. Certitude provides information on faults which is usable for this purpose.

These additional pairs are merged with the pairs selected by Certitude and submitted to property checking. All results are reported even though Certitude will not be interested in the extra qualification results in this round. Therefore, all qualification results are stored in extra book-keeping files. In forthcoming qualification rounds, these extra files are first searched for previous proof results, and only pairs not yet checked are submitted to the proof engines.

D. Handling of Indeterminate Results

If a proof is aborted, the result may be indeterminate. The question is whether such a fault can be considered to be detected or not, if it was not detected within the time-limit corresponding to the original proof-time without fault. The run-time of an aborted check would not be a valuable input for Certitude's selection heuristics. With an external book-keeping of all check results Certitude's detection phase is re-started without repeating finalized property checks.

VI. EXPERIENCE AND RESULTS

Both approaches yield useful qualification results on productive designs. Table 1 lists a few results from applying both methods to real system control modules of the new product family AURIX™. The numbers indicate that 100% qualification of large formally verified modules is feasible.

Module verification			Quantify		Certitude	
No.	Locs (VHDL)	Props	Code locat.	Days	Faults	Days
1	25563	85	2316	4	1784	7
2	27374	157	1993	5	3732	12
3	57168	253	5309	7+	4122	17

Table 1: Results for different designs

The results of Quantify and Certitude cannot be 1:1 compared, as the number of faults in Certitude depends on the configured instrumentation and is different from the fault injection performed by Quantify, which uses an approach similar to X-assignment. The instrumented RTL code generated by Certitude can be inspected, although it is not recommendable to read the augmented code. In our verification setting, the same fault injection configurations are used for all modules of the SOC, regardless whether they are verified by simulation or formally. Certitude could be configured to instrument many more faults than shown in this table, while in Quantify the internal fault-model cannot be influenced by the user. Only for the biggest module, we

interrupted Quantify after 7 days with about 80% coverage, 3% code proven uncovered, and the remaining code regions open, because there was no further progress, due to the complexity of some properties. In fact, further progress is expected with future releases of Quantify like in first versions, when big modules could not yet be processed. Both approaches yielded similar results with respect to code regions not covered at all, which gave rise to property enhancements. As can be imagined, long-running properties which need more than 1 hour regular proof time are hardly suitable for such kind of iterative qualification. With the maximum density of instrumented faults, Certitude would yield more detailed qualification results than Quantify, however, in potentially unaffordable qualification time.

Table 2 compares several aspects of the usage of Quantify and Certitude.

Quantify	Certitude
Internal fault model	Explicit fault injection in RTL design, large variety of configurable fault models
OK for small and medium designs (-7 days qualification time with parallelization) Ongoing efficiency improvements	All designs manageable which have been formally verified
Re-startable	Re-startable
Powerful detection of dead, redundant, constrained code	Basic exclusion of dead-code from instrumentation
User control via options specifiable at start	Full observability of qualification checks, intermediate reconfigurability
No problems by fault injection	Sometimes problems with elaboration of instrumented RTL design
No exact relation to simulation coverage (stronger!), but similar to code coverage	100% matching of coverage between simulation and formal property checking
Closed functionality, indirect enhancements from outside	Scripts for communication and check control extensible
Fast growth of qualification results, finalization depending on effort level	Incomplete property sets efficiently handled by fault dropping
Reporting intermediate coverage state as HTML	Reporting intermediate coverage state as HTML or proprietary format

Table 2: Comparison of Quantify and Certitude flow

As Quantify can be directly started by a push-button in an active MV 360° session without further set-up, it makes sense to run it in the background and evaluate the results for completion of property sets. Starting the combined Certitude – Onespin flow is easy as well, once the scripts have been set up, but it may be required to exclude few lines from instrumentation if the model generation fails.

VII. CONCLUSIONS

This paper has discussed methodological enhancements and experience from the qualification of formal properties for AURIX™ modules. The experience with real modules is very positive, and has led to new sign-off criteria for formal properties. The property sets can be created without specific methodology or expertise regarding formal completeness.

By running on average 5 coverage checks in parallel, focusing coverage analyses to property subsets, using heuristics for advantageous property-code-location pairs, and exploiting information about dead, constrained and redundant code computed by MV 360°, the automatic property qualification took up to a week per module.

The two approaches illustrate possible paradigms:

1. fully integrated formal qualification flow where formal engines internally provide functions for optimizing the formal qualification directly
2. open formal qualification flow which provides common metrics for different verification tools.

Paradigm 1 has the potential to provide more efficient internal dependency analyses and data management under the surface by exploiting all highly optimized technology of MV 360°. As the results are also delivered in UCDB/XML format, integration with simulation flows is possible.

Paradigm 2 is reliant on the accessibility of advanced data and control functions provided and used by MV 360°, such as reachability information, options like proof engines, parallelization, run-time limits. It is widely applicable, and is open for combining any simulator and property checker results while applying a common metric.

REFERENCES

[1] M. Siegel,"Verification Coverage and Productivity Through Formal Operation- and Transaction-Level Verification Using SVA", Tutorial at DVCON 2010.

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

[3] Y. Hoskote, T. Kam, P.-H. Ho, X. Zhao, "Coverage Estimation for Symbolic Model Checking," in Proceedings of 36th Annual Conference on Design Automation (DAC'99) , 1999, pp. 300-305.

[4] S .Katz, O. Grumberg, D. Geist, " Have I written enough properties? – A method for comparison between specification and implementation," Haifa Israel. Charme 1999.

[5] H. Chockler, O.Kupferman, M.Y.Vardi, "Coverage Metrics for Temporal Logic Model Checking", in Formal Methods in System Design, pp. 189-212, Vol. 28, Issue 3, 2006.

[6] M. Oberkönig, M. Schickel, and H. Eveking, "A Quantitative Completeness Analysis for Property-Sets," in Proceedings. of FMCAD'07, 2007.

[7] J. Bormann, S. Beyer, et al., "Complete Formal Verification of TriCore2 and Other Processors," Proceedings of DVCON07, 2007.

[8] V. Singhal, P. Aggarwal," Using Coverage to Deploy Formal in A Simulation World", CAV 2011.

[9] F.Haedicke, D.Große, R.Drechsler,"A Guiding Coverage Metric for Formal Verification", DATE-12, 2012.

[10] H. Busch, "Formal Safety Verification of Automotive Microcontroller Parts", ZuE'12, 6.GMM/GI/ITG-Workshop, 2012.