The Role of Mutation Analysis for Property Qualification[*]

Luigi Di Guglielmo Franco Fummi Graziano Pravadelli
Dipartimento di Informatica – Università di Verona
Strada le Grazie 15, 37134 Verona, Italy
{luigi.diguglielmo, franco.fummi, graziano.pravadelli}@univr.it

Abstract

The paper proposes a comprehensive methodology for property qualification based on a combination of dynamic and static techniques. In particular, given a set of properties defined to check the correctness of a design implementation, the methodology first evaluates property coverage, property overspecification, and it identifies vacuous properties. This is commonly performed by exploiting mutation analysis and automatic testbenches generation, i.e., dynamic strategies. This phase allows us to quickly evaluate the quality of properties with respect to the use of formal approaches. Then, a second phase, based on model checking, is applied to the restricted number of situations, where the dynamic approach is not exhaustive. Experimental results show the effectiveness and efficiency of the proposed methodology.

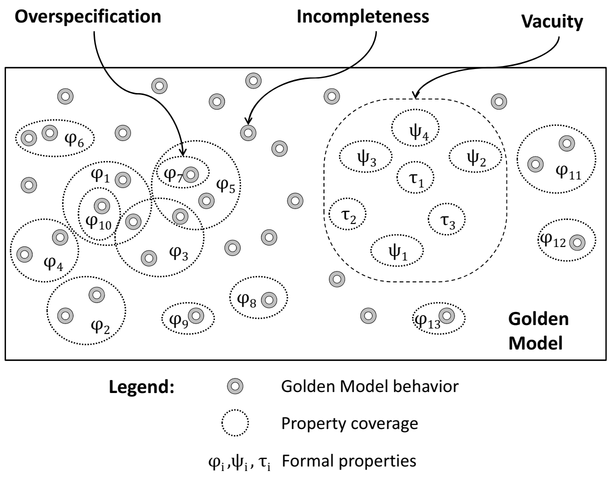


Figure 1. Golden Model analysis.

1 Introduction

Mutation analysis [1] relies on the perturbation of the design under verification (DUV) by introducing new statements or modifying existing ones in small ways. As a consequence, many versions of the model are created, each containing one mutation and representing a *mutant* of the original DUV. Test cases are used to simulate mutants with the goal of distinguishing their outputs from the original DUV ones. In fact, the presence of not-distinguished mutants points out inadequacies in the test cases or in the DUV model. Thus, mutation analysis is generally used in dynamic verification, and its main purpose consists of helping the verification engineer to develop effective testbenches able to activate all DUV sections.

On the contrary, in this paper, we propose to exploit mutation analysis in the context of static verification, with the purpose of qualifying formal properties defined to check the correctness of the DUV. Moreover, we used functional faults as mutants, thus verifying not only the activation of DUV parts, but also the propagation of fault effects. Such kind of mutants allows us to qualify properties. In particular, we propose to use mutation analysis to evaluate how much defined properties adhere to the golden model of the DUV derived from its specification. Mutations are applied to either *checkers* derived from properties (i.e., components described by means of hardware description language integrated into the simulation environment that monitor simulation for detecting violations of the corresponding property), or the DUV implementation, in order to assess, respectively, the quality and the comprehensiveness of the defined properties. In particular, property qualification concerns three aspects (Figure 1):

- *Vacuity*. Properties are vacuously satisfied if they hold in a model, and can be strengthened without causing them to fail. The vacuous satisfaction points out problems either in property, or in environment definition, or in the model implementation.
- *Completeness*. A set of properties could be incomplete since some requirements could be only partially for-

[*]This work has been partially supported by the European project COCONUT FP7-2007-IST-1-217069.

malized into properties. As a consequence behaviors uncovered by properties can exist, so implementation could be wrong even if it fulfills all the defined properties.

- *Overspecification*. The set of properties could be overspecified if it contains properties that can be derived as logical consequence of other properties. For example, it is possible to define a property (e.g., φ_7 in Figure 1) whose coverage is a subset of the coverage of another defined property (e.g., φ_5 in Figure 1). Thus all behaviors modeled by the first property are also modeled by the latter. The presence of such overspecification yields the verification time to be longer than it is required to be.

Several papers in literature address these problems, but the proposed solutions present non-negligible drawbacks:

- they are suited for either vacuity analysis or property coverage or overspecification analysis. To the best of our knowledge, there is not a unique methodology to analyze in a comprehensive way all of them.
- they exploit only formal techniques affecting sensibly the verification time.

In particular, current approaches to *vacuity analysis*, generally, exploit formal methods to search for an *interesting witness* proving that a property does not pass vacuously [2, 3, 4, 5]. Such approaches are, generally, as complex as model checking, and they require to define and model check further properties obtained from the original ones by substituting their subformulas in some way, thus sensibly increasing the verification time. Current approaches for *property coverage*, can be divided into two categories: mutation-based [6, 7, 8, 9, 10, 11, 12, 13] and implementation-based [14, 15, 16]. The firsts rely on a form of mutation coverage that requires perturbing the design implementation before evaluating property coverage. In particular, [13] gives a good theoretic background with respect to mutation of both specification and design. The latter ones, estimate the property coverage by analyzing the original implementation without the need to insert perturbations. The main problem of these approaches is due to the adoption of symbolic algorithms that suffer of the state explosion problem. Finally, as regards the problem of overspecification, automatic techniques for dealing with *overspecification removal* have not been investigated in literature. Such a problem has been partially addressed in [17]. The authors underline that given a set of properties there can be more than one overspecified formula, and they can be mutually dependent, thus, they cannot be removed together. The authors show that finding the minimal set of properties that does not contain overspecifications is a computationally hard problem.

The goal of this paper consists of proposing a comprehensive and homogeneous methodology for property qualification that exploits mutation analysis, based on functional faults, instead of formal techniques, to address the vacuum cleaning, the analysis of the degree of completeness and the overspecification of a set of properties. The proposed methodology allows us to reduce the evaluation time with respect to pure formal approaches providing as effective property qualification.

The paper is organized as follows. Section 3 describes the proposed methodology. Section 4 reports experimental results that highlight the effectiveness and the efficiency of the proposed methodology. Finally, Section 5 is related to concluding remarks.

2 Related works

Let us summarize the most important results published on this topic.

2.1 Vacuity analysis

Vacuity analysis is a mandatory process looking for properties that, passing vacuously, lead verification engineers to a false sense of safety.

According to Beer et al. [2], a formula φ passes vacuously in a model M if it passes in M, and there is a subformula ψ of φ that can be changed arbitrarily without affecting the outcome of model checking.

Automatic techniques to detect trivial passes are proposed in [2, 3, 4, 5, 18, 19, 20, 21, 22, 23, 24] and, generally, define and model check new properties obtained from the original ones by substituting its subformulas in some way.

In particular, in [3, 5], the authors propose a methodology for vacuity detection applicable to CTL* formulas. The authors argue that vacuity detection consists of checking whether all the sub-formulae of φ affect its truth value in the system. This is obtained by replacing each sub-formula ψ by either $\mathbf{true}$ or $\mathbf{false}$ depending on whether ψ occurs in φ with negative polarity (i.e., under an odd number of negations) or positive polarity (i.e., under an even number of negation). Thus, vacuity checking amounts to model checking the system with respect to the formulae $\varphi[\psi \leftarrow \mathbf{true}]$ and $\varphi[\psi \leftarrow \mathbf{false}]$ for all sub-formulae ψ of φ. Note that this method is for vacuity with respect to sub-formula occurrences. While a sub-formula occurrence has a *pure* polarity (exclusively negative or positive), a sub-formula with several occurrences may have *mixed* polarity (both positive and negative occurrences). It is shown in [3] that the method is no longer valid by considering sub-formulae instead of sub-formulae occurrences.

In [4], Beer et al. prove a result similar to the one in [3], that holds for all logics with polarity. However, the authors show a practical solution only for a subset of ACTL, and they check vacuity only with respect to the smallest important sub-formula.

In [19], the aim of the authors is to remove the restriction of [4, 5] concerning sub-formula occurrences of pure polarity. To keep things simple, the authors stick to LTL. Strategies presented in [4, 5] consider only syntactic perturbation on a formula φ, instead, in [19] the authors consider the notion of semantic perturbation which is modeled by universal quantifier (i.e., UQLTL). Such a semantic can be interpreted with respect either to the model (structure semantics) or to its set of computation (trace semantics).

In [22], the authors introduce the notion of weak vacuity. They consider a formula to be vacuously satisfied if there is a subformula that can be replaced by a stronger formula without affecting the truth value of the whole formula. Finding such stronger formulas can be done by solving temporal logic queries but minimal solutions do not need to exist in general. Moreover, the computation of stronger formulas may not justify the truth value of the original formula by a trivial reason. Therefore, the authors restrict the set of possible solutions to a set of user-selected formulas which are considered to be interesting for detecting vacuity (i.e., vacuity causes).

The work in [23] focuses on vacuity detection for SAT-based Bounded Model Checking (BMC). Information (e.g., resolution proofs) generated by BMC executions is exploited by the authors to identify vacuous variables and reduce the number of the required model checking runs. The method, however, is completed by running a naive algorithm on the remaining atomic sub-formulas that replaces such atomic sub-formulas with unconstrained boolean variables and runs BMC for each substitution.

Finally, in [24], Ben-David et al. underline that vacuity detection for certain logics can be complex and time consuming and indicate that not all vacuities detected in practical applications are considered a problem by the system verifier. As a consequence, the authors limit their analysis to the problem of detecting antecedent failure. They define Temporal Antecedent Failure, a refinement of vacuity that occurs when some pre-condition in the formula is never fulfilled in the model.

2.2 Completeness analysis

The degree of completeness of the set of properties is based on the *property coverage* that is the capability of properties to identify mutants of the original design. Property coverage addresses the question of whether enough properties have been defined. With the aim of reducing design error escapes and avoiding property incompleteness, the verification engineers need to continuously strengthen existing properties and specify new ones. Coverage metric can guide designers toward a complete verification of all possible design behaviors. The few papers existing in the literature related to the property incompleteness problem estimate the property coverage either by analyzing the original implementation [14, 15, 16] or analyzing some form of mutation coverage that requires perturbing the design implementation [6, 7, 8, 9, 10, 11, 12, 13] before evaluating the property coverage. However, all these approaches rely on formal techniques.

Symbolic approaches proposed in [6, 7, 8, 9, 10, 13] measure the property coverage by analyzing whether a mutation on an observed signal, at a certain state, changes the truth values of the defined properties. These solutions are affected by the state explosion problem since the worst-case complexity of such algorithms is exponential in the size of the property for safety properties and allow one to compute only state coverage metrics. Moreover, a state metric leaves quite a large portion of the design behaviors unchecked. In fact, it does not deal with path coverage and, thus, possible incompleteness of properties related to wrong paths of an implementation cannot be detected.

To overcome this limitation, in [11] the authors propose several mutation models and coverage metrics to cover different design aspects in a state graph. The proposed solution considers only structural mutations (i.e., inserting or deleting a state or a transition) and, moreover, require to recheck each property for each mutation which is definitely too time-consuming to be applied to complex real cases where a large number of mutations must be considered.

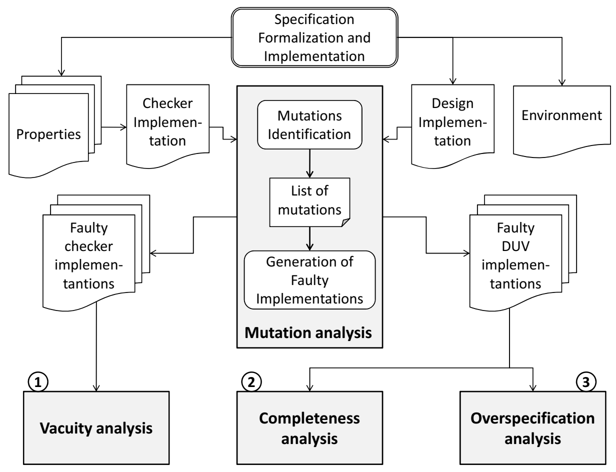
A symbolic approach which does not require perturbations is presented in [14]. The approach requires the instantiation of the design model, the tableau of the property and the simulation relation, thus considerably expanding the state space. Moreover, this approach is also only suitable for safety ACTL properties.

A more efficient approach is presented in [15] where the completeness of properties is evaluated as the percentage of the transitions of the finite state machine (FSM), which models the design, that is covered by the properties. However, since the transition traversal does not consider the values of signals, transition traversal coverage cannot precisely reflect the completeness of properties.

Finally, in [16], the authors highlight the necessity of a coverage metrics for estimating the number of properties necessary to guarantee the correctness of the design under verification. However, they compute the coverage estimation by means of a very complex manual analysis of execution paths not exercised by any test case.

2.3 Overspecification analysis

Overspecification analysis looks for properties that can be derived as logical consequence of other properties. The



Specification Formalization and Implementation
Properties
Checker Implementation
Mutations Identification
List of mutations
Generation of Faulty Implementations
Mutation analysis
Design Implementation
Environment
Faulty checker implementantions
Faulty DUV implementantions
1
2
3
Vacuity analysis
Completeness analysis
Overspecification analysis

Figure 2. Methodology overview.

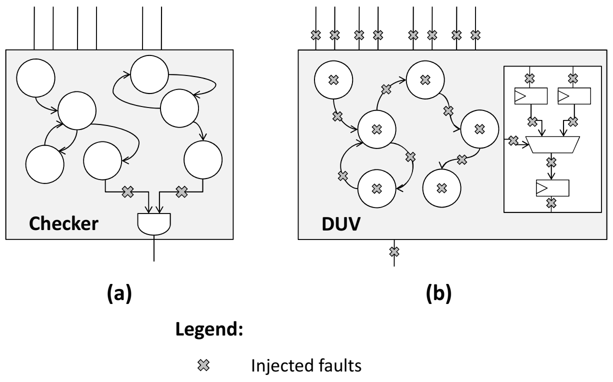


Figure 3. Mutation application.

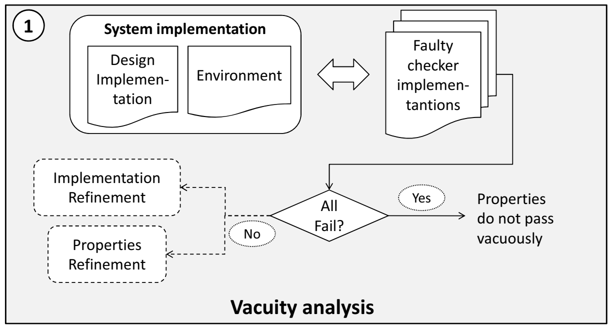


Figure 4. Vacuity analysis.

analysis can be performed by exploiting theorem proving. The complexity of theorem proving is exponential in the worst case and, moreover, it is not completely automatized, since human interaction is very often required to guide theorem prover during the proof. In literature, for the best of our knowledge, automatic techniques for overspecification removal have not been investigated.

However, the problem of property overspecification is partially addressed in [17]. The authors underline that given a set of properties there can be more than one overspecified formula, and they can be mutually dependent, e.g., while for $i \in \{1, 2\}$ φ_i is overspecified with respect to $\Phi \setminus \{\varphi_i\}$, neither is overspecified with respect to $\Phi \setminus \{\varphi_i, \varphi_2\}$, thus, they cannot be removed together. Moreover, the authors show that finding the minimal subset of Φ's properties that is equivalent to Φ, and does not contain overspecifications, is $FP^{\Sigma_2[\log n]}$-hard. Thus, they propose an algorithm that tries to remove properties one at a time, settling for a non-optimal solution.

3 Methodology overview

As shown in Figure 2, mutation analysis allows us to investigate vacuity of properties as well as properties completeness and overspecification. Once the checker [25] and DUV implementation are defined, it is possible to proceed with the mutation analysis, generating the faulty implementations. Mutations are implemented as *faults*, i.e., group of statements that causes an incorrect behavior of the design. Figure 2 underlines that faulty checker implementations are generated with the aim of vacuity analysis, while the goal of faulty DUV implementations is both property completeness and overspecification analysis. In particular, vacuity analysis is estimated in relation to the number of mutants of checker implementations that are distinguished. As regards the incompleteness of properties, it is estimated in relation to the number of mutants of DUV properties cannot detect, similarly to what was proposed in the mutation-based approaches cited in Section 1. Finally, property overspecification is estimated in relation to the number of mutants of the DUV each property can detect.

It is worth noting that our approach is independent of the way properties have been defined and it involves the analysis of both states and paths. Moreover, thanks to the adoption of dynamic verification, our methodology allows us to evaluate the property vacuity, incompleteness and overspecification in a short time with respect to the large time needed by symbolic methods.

The methodology described in Figure 2 is analyzed thoroughly in the following sections.

3.1 Vacuity analysis

When the specification is formalized by temporal properties, it is reasonable to analyze the set of satisfied properties to determine the presence of vacuous formulas. From the practical point of view, such properties are useless for the design verification and also they lead verification engineers to a false sense of safety. Such vacuity analysis can be performed adopting the following simulation-base methodology [26] characterized by two main phase:

- *Generation of faulty checkers* (Figure 3.a). Given the set of formulas that hold in the model of the DUV, a checker is automatically generated modeling each formula φ. Once the set of faults is identified, faults are injected in the so obtained checkers. In particular, for each formula φ, we inject, in the corresponding checker, one fault for each minimal sub-formula of φ. Intuitively, an interesting fault, perturbs the checker behavior similarly to what happen when a sub-formula ψ is substituted by $\mathbf{true}$ or $\mathbf{false}$ in φ [4].

- *Vacuity analysis* (Figure 4). The faulty checkers are connected to the model composed of the DUV and the related environment. Then, testbenches are used to simulate the DUV. Note that, testbenches for the corresponding checkers can be generated by using an ATPG (Automatic Test Pattern Generator). The vacuity analysis relies on the observation of the simulation result. A checker failure due to the effect of a fault f corresponds to prove that the sub-formula ψ associated to (perturbed by) f affects the truth value of φ. On the contrary, faults that do not cause checker failures (i.e., not detected faults) must be thoroughly analyzed, since the incapability of detecting a fault is symptom of vacuity. In this case, our methodology provides the verification engineer with feedback to analyze such a symptom by relating each undetected fault to a single sub-formula of φ.

Completed the vacuity analysis, it is possible to state that the set of remaining properties does not pass vacuously, and it is possible to proceed to measure its degree of completeness.

3.2 Completeness analysis

The degree of completeness of the set of properties can be measured exploiting the following *property coverage*-based methodology [27] that relies on the capability of properties to identify faults that perturb the design implementation. The set of properties is complete, w.r.t. the desired golden model, if at least one property fails in presence of each fault affecting the outputs of the implementation. Otherwise, the set of properties is incomplete. In fact, if a fault does not cause at least a failure among the defined properties, it means that the golden model is satisfied by different implementations: the fault-free and the faulty one. The computation of property coverage consists of two phases:

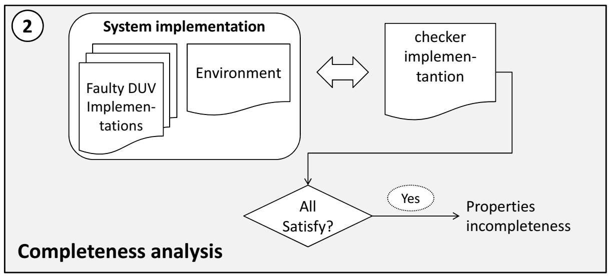


Figure 5. Completeness analysis.

- *Generation of faulty DUV implementations* (Figure 3.b). Perturbations of design implementation are generated by automatically injecting faults inside the design implementation described by means of hardware description language. The obtained fault list must include only *detectable faults*, which are faults that, for at least one input sequence, cause at least one output of the faulty implementation to differ from the corresponding output of the fault-free implementation. We consider only detectable faults to achieve an accurate estimation of the golden model completeness because undetectable faults cannot cause failures on the properties since they do not perturb outputs of the implementation. The set of detectable faults can be identified by using an ATPG [27].

- *Property coverage analysis* (Figure 5). The presence of a detectable fault implies that the behavior of the faulty implementation differs from the behavior of the fault-free implementation. Thus, while the defined properties are satisfied by the fault-free implementation, instead at least one of these should be falsified if checked on a faulty implementation. The computation of such property coverage requires too long time/space resources using only formal techniques. For this reason, we adopt a different metric, called *witness coverage*, that equals property coverage, but can be computed by exploiting dynamic verification, i.e., mutants simulation and ATPG, instead of formal verification, i.e. model checking. In particular, the witness coverage is computed as:

$$C_w = \frac{\text{number of faults detected by properties}}{\text{number of detectable faults}}$$

3.3 Overspecification analysis

To measure the degree of overspecification of the set of properties without using theorem proving, it is possible to exploit the following methodology [28] characterized by two phases:

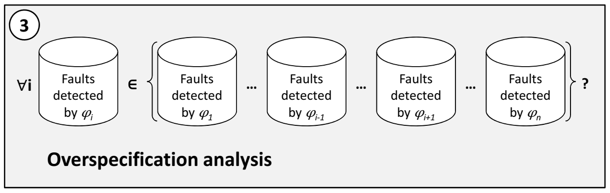

3
∀i
Faults detected by φ_i
∈
Faults detected by φ_1
…
Faults detected by φ_{i-1}
…
Faults detected by φ_{i+1}
…
Faults detected by φ_n
?
Overspecification analysis

Figure 6. Overspecification analysis.

- *Generation of faulty DUV implementations* (Figure 2). This step coincides with the first phase of the completeness analysis process. Thus, once the faulty design implementations have been generated, they can be exploited also for overspecification analysis.
- *Identification of properties role* (Figure 6). The property role is intended as the capability of a property to check the correctness of a subset of the DUV functionalities, i.e., its *property coverage*. The property coverage computation provides, for each property φ_i, the set of faults $\mathcal{F}_i$ which are detected by φ_i. Comparing the set of faults $\mathcal{F}_i$ detected by each property φ_i is possible to determine if a property φ_i is not overspecified. In fact, if a property φ_i can detect a fault that none of the other properties can detect, it is surely not an overspecification.

3.4 Limitations

The adoption of mutation analysis and dynamic strategies for property qualification can lead to an underestimation of the degree of non-vacuity, completeness and non-overspecification of a set of properties. This is due to the lack of exhaustiveness of dynamic approaches. To exceed this limitation, the proposed methodology presents a second phase based on model checking that guaranties effectiveness of the analysis results. For vacuity analysis, model checking is applied only on properties that at the end of the simulation, have not been marked as non-vacuous. As regards completeness analysis, formal verification is limited to those mutants not distinguished by properties during the simulation. Finally, to assure the correctness of the degree of overspecification of the set of properties, properties not marked as non-overspecified by the property coverage analysis, should be checked exploiting theorem proving. Hence, it is worth noting that formal verification is applied only if necessary and on a limited number of cases. Thus, it does not increase sensibly the overall verification time as showed by experimental results reported in Section 4.

Another limitation of the proposed methodology derives from the fact that it is not possible to conservatively generate checkers for all kind of temporal formulas. For example, while formulas that involve branching behavior can be checked by means of formal verification, it can be difficult to evaluate them by simulation. In particular, it is possible to generate checkers for formulas defined according to the *Simple Subset* of PSL [29], that conforms to the notion of monotonic advancement of time. This restriction guarantees that formulas within such a subset can be simulated.

4 Experimental results

The methodology has been evaluated by using some of the well-known ITC-99 benchmarks [30] and a real industrial case provided by STMicroelectronics that is an integer square-root calculator. Their characteristics are described in Table 1. Some properties for ITC-99 benchmarks have been taken from the *vis-verilog-models-1.2* archive [31] while others have been defined by analyzing designs specifications. On the contrary, properties for the square-root model have been defined only by analyzing its specification. Checkers of the properties have been generated by using FoCs, while mutations are modeled by means of a high-level fault model (i.e., *bit coverage*) strictly related to design errors [32]. Faults have been automatically injected into the checker and design implementations by using the features of the Laerte++ automatic test pattern generator, which has been used also for generating testbenches.

Design	PIs	POs	Gates	FF	Lines	Ψ
b02	3	1	72	4	70	14
b06	4	6	140	9	128	17
b10	13	6	657	17	167	20
sqrt	66	64	6383	163	352	10

Table 1. Characteristics of benchmarks.

Columns of Table 1 report, respectively, design names (*Design*), the number of primary inputs (*PIs*), primary outputs (*POs*), gates (*Gates*), flip flops (*FF*), lines of HDL code (*Lines*) and analyzed properties (Ψ).

4.1 Vacuity evaluation

	Fault simulation					Formal technique			
Design	IF	DF	NV	TV	time (s.)	Φ	*f*-Φ	time (s.)	Sav.
b02	35	35	14	30	0.186	35	35	0.817	77%
b06	76	76	17	50	0.526	76	76	3.545	85%
b10	72	72	20	50	0.586	72	72	2.725	78%
sqrt	32	32	10	120	1.632	32	32	31.564	94%

Table 2. Vacuity analysis results.

Vacuity analysis results are shown in Table 2. The columns report the design name (*Design*) and information related to the vacuity analysis methodology proposed in

Section 3.1 (*Fault simulation*), i.e., the number of injected faults (*IF*), detected faults (*DF*), not vacuous properties (*NV*), test vectors (*TV*), and the time required for automatic testbench generation and fault simulation of faults by using Laerte++ (*time*). The proposed approach have shown that the properties included in the *vis-verilog-models-1.2* archive and the ones defined starting from designs specifications do not pass vacuously, since all injected faults have been detected. As a consequence, it is not necessary to model check none of the defined properties.

On the contrary, the adoption of a formal technique for vacuity analysis such as the one proposed in [4] requires to define and model check a further set of new formulas (i.e., witness formulas) whose falsification assures the non-vacuity of the initial properties. Note that the cardinality of the set coincides with the number of injected faults. In particular, the right-most columns of Table 2 report information related to the vacuity analysis performed according to the methodology proposed in [4] *(Formal Technique)*, i.e., number of witness formulas (Φ), number of falsified witness formulas (*f*-Φ) and the time required by such formal technique to model check witness formulas (*time*). Finally, column *Sav.* reports the percentage of computational time saved by using the fault simulation-based methodology instead of model checking witness formulas.

4.2 Completeness evaluation

	Fault simulation				
Design	Det	det	Uncov.	time (s.)	$C_p\%$
b02	58	58	0	0.220	100%
b06	138	138	0	1.176	100%
b10	251	251	0	8.721	100%
sqrt	1657	1657	0	4.020	100%

Table 3. Completeness analysis results.

Table 3 shows the property coverage measured by adopting the proposed methodology explained in Section 3.2. The columns report respectively the design name (*Design*), the number of detectable faults (*Det*), detected faults (*det*), not detected faults (*Uncov.*), the time required for automatic testbench generation and fault simulation using Laerte++ (*time*), and the final property coverage obtained ($C_p\%$). It is worth noting that model checking has not been used to detect mutants. In fact, the adoption of the ATPG (i.e., Laerte++) allows us to distinguish all mutants created with the mutation analysis by observing properties failures.

4.3 Overspecification evaluation

Overspecification analysis results are shown in Table 4. We used SMV to compare the effectiveness of

Design	Φ	OS	NOS	time (s.)	TP (s)	Savings (%)
b02	14	2	12	3.2	5.600	42
b06	17	3	14	27.114	60.350	55
b10	20	1	19	182.099	328	44
sqrt	10	1	9	101.8	560	81

Table 4. Overspecification analysis results.

the simulation-based methodology proposed in Section 3.3 with respect to the use of formal deduction. Actually, SMV is a model checker, but it can be used to search if $\Phi \setminus \{\varphi_i\} \models \varphi_i$ in the following way:

- Instantiate a model by declaring only the entity (i.e., the type of primary inputs and primary outputs). The architecture is not required, since we know that φ_i is satisfied by the model, and we are interested in verifying if φ_i can be deduced by $\Phi \setminus \{\varphi_i\}$ independently from the model.
- Ask SMV to prove if φ_i is true by assuming $\Phi \setminus \{\varphi_i\}$ on the empty model. This can be performed by using the following SMV directive:

$$\texttt{using } \varphi_1, ..., \varphi_{i-1}, \varphi_{i+1}, ..., \varphi_n \texttt{ prove } \varphi_i.$$

Due to the fact that finding a proof of deducibility is very hard process, SMV has been used in the bounded model checking (BMC) mode. Columns of Table 4 show the number of defined properties (Φ), the effective number of overspecified (*OS*) and non overspecified (*NOS*) properties, the execution time (*time*) required to apply the proposed methodology. This result considers the time for computing the property coverage and for comparing set of faults detected by properties, added to the time required to apply BMC for investigating if the properties marked as overspecification by property coverage are really overspecifications. Then, column *TP* reports the execution time required for a pure BMC without exploiting property coverage. Finally, column *Savings*, reports the percentage of computational time saved by using the property coverage-based methodology with respect to the use of BMC only.

5 Concluding remarks

In this paper we have presented a comprehensive and homogeneous methodology for property qualification that concerns vacuity, completeness and overspecification analysis. First, the methodology allows us to analyze the vacuity of the defined properties by simulating their corresponding checkers perturbed by faults associated to specific subformulas. The detection of such faults assures the non-vacuity of formulas. Second, the methodology allows us to measure the degree of completeness of the defined properties by computing the property coverage that is the capability of properties of detecting mutants of the original DUV. Finally, the same property coverage is exploited to analyze the logical consequence of properties. Formal verification, if still necessary, is applied only on a limited number of cases, thus sensibly saving verification time.

References

[1] A. Offutt and R. Untch. *Mutation 2000: Uniting the Orthogonal*. Mutation testing for the new century, pp. 34–44.

[2] I. Beer, S. Ben-David, U. Eisner, and Y. Rodeh. *Efficient Detection of Vacuity in ACTL Formulas*. In *International Conference on Computer Aided Verification*, vol. 1254, pp. 279–290. 1997.

[3] O. Kupferman and M. Y. Vardi. *Vacuity Detection in Temporal Model Checking*. In *Conference on Correct Hardware Design and Verification Methods*, pp. 82–96. 1999.

[4] I. Beer, S. Ben-David, C. Eisner, and Y. Rodeh. *Efficient Detection of Vacuity in Temporal Model Checking*. Formal Methods in System Design, pp. 141–163, 2001.

[5] O. Kupferman and M. Vardi. *Vacuity detection in temporal model checking*. International Journal on Software Tools for Technology Transfer, pp. 224–233, 2003.

[6] Y. Hoskote, T. Kam, P. Ho, and X. Zhao. *Coverage Estimation for Symbolic Model Checking*. In *Proc. ACM/IEEE Design Automation Conference*, pp. 300–305. 1999.

[7] H. Chockler, O. Kupferman, R. Kurshan, and M. Vardi. *A Practical Approach to Coverage in Model Checking*. In *Proc. Computer Aided and Verification*, pp. 66–78. 2001.

[8] H. Chockler, O. Kupferman, and M. Vardi. *Coverage Metrics for Temporal Logic Model Checking*. Formal Methods in System Design, pp. 189–212, 2006.

[9] H. Chockler, O. Kupferman, and M. Vardi. *Coverage Metrics for Formal Verification*. International Journal on Software Tools for Technology Transfer (STTT), pp. 373–386, 2006.

[10] N. Jayakumar, M. Purandare, and F. Somenzi. *Dos and Donts of CTL State Coverage Estimation*. In *Proc. 40th Design Automation Conference (DAC)*. 2003.

[11] T. Lee and P. Hsiung. *Mutation Coverage Estimation for Model Checking*. In *Proc. Int'l Symp. Automated Technology for Verification and Analysis*, pp. 354–368. 2004.

[12] X. Xu, S. Kimura, K. Horikawa, and T. Tsuchiya. *Transition-Based Coverage Estimation for Symbolic Model Checking*. In *Proc. ACM/IEEE Asia and South Pacific Conf. Design Automation*, pp. 1–6. 2006.

[13] O. Kupferman, W. Li, and S. Seshia. *A Theory of Mutations with Applications to Vacuity, Coverage, and Fault Tolerance*. In *Proc. IEEE International Conference on Formal Methods in Computer-Aided Design*. 2008.

[14] S. Katz and O. Grumberg. *Have I Written Enough Properties? - A Method of Comparison between Specification and Implementation*. In *Proc. ACM Advanced Research Working Conf. Correct Hardware Design and Verification Methods*, pp. 280–297. Springer, 1999.

[15] X. Xu, S. Kimura, K. Horikawa, and T. Tsuchiya. *Transition Traversal Coverage Estimation for Symbolic Model Checking*. In *Proc. ACM/IEEE Int'l Conf. Formal Methods and Models for Co-Design*, pp. 259–260. 2005.

[16] P. Mishra and N. Dutt. *Automatic Functional Test Program Generation for Pipelined Processors using Model Checking*. In *Proc. IEEE High-Level Design Validation and Test*, pp. 99–103. 2002.

[17] H. Chockler and O. Strichman. *Easier and More Informative Vacuity Checks*. In *Proc. ACM/IEEE International Conference on Formal Methods and Models for Codesign*, pp. 189–198. 2007.

[18] M. Purandare and F. Somenzi. *Vacuum Cleaning CTL Formulae*. In *International Conference on Computer Aided Verification*, vol. 2404, pp. 485–499. 2002.

[19] R. Armoni, L. Fix, A. Flaisher, O. Grumberg, N. Piterman, A. Tiemeyer, and M. Vardi. *Enhanced Vacuity Detection in Linear Temporal Logic*. In *International Conference on Computer Aided Verification*, vol. 2725, pp. 368–380. 2003.

[20] A. Gurfinkel and M. Chechik. *Extending Extended Vacuity*. In *International Conference on Formal Methods In Computer-aided Design*, vol. 3312, pp. 306–321. 2004.

[21] K. Namjoshi. *An Efficiently Checkable, Proof-Based Formulation of Vacuity in Model Checking*. In *International Conference on Computer Aided Verification*, vol. 3114, pp. 57–69. 2004.

[22] M. Samer and H. Veith. *Parameterized Vacuity*. Lecture Notes in Computer Science, pp. 322–336, 2004.

[23] J. Simmonds, J. Davies, A. Gurfinkel, and M. Chechik. *Exploiting Resolution Proofs to Speed Up LTL Vacuity Detection for BMC*. In *IEEE International Conference on Formal Methods In Computer-Aided Design*, pp. 3–12. 2007.

[24] S. Ben-David, D. Fisman, and S. Ruah. *Temporal Antecedent Failure: Refining Vacuity*. Lecture Notes in Computer Science, pp. 492–506, 2007.

[25] Y. Abarbanel, I. Beer, L. Gluhovsky, S. Keidar, and Y. Wolfsthal. *FoCs: Automatic Generation of Simulation Checkers from Formal Specifications*. In *Computer Aided Verification*, pp. 538–542. 2000.

[26] L. Di Guglielmo, F. Fummi, and G. Pravadelli. *Vacuity Analysis by Fault Simulation*. In *ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pp. 27–36. 2008.

[27] A. Fedeli, F. Fummi, and G. Pravadelli. *Properties Incompleteness Evaluation by Functional Verification*. IEEE Transactions on Computers, pp. 528–544, 2007.

[28] S. Brait, F. Fummi, and G. Pravadelli. *On the Use of a High-Level Fault Model to Analyze Logical Consequence of Properties*. In *ACM/IEEE International Conference on Formal Methods and Models for Co-Design*, pp. 221–230. 2005.

[29] *Standard for Property Specification Language (PSL)*. IEC 62531:2007 (E), pp. 1–156, 2007.

[30] Politecnico di Torino. *ITC-99 Benchmarks*. In *http://www.cad.polito.it/tools/itc99.html*. 1999.

[31] University of Colorado. *VIS*. In *http://vlsi.colorado.edu/ vis*. 1999.

[32] F. Fummi, C. Marconcini, and G. Pravadelli. *Logic-Level Mapping of High-Level Faults*. The VLSI Journal Integration, pp. 467–490, 2005.