Verification of FlexRay using directed and coverage-based testing – A comparison

Markus Baumeister, Philips Research Laboratories Aachen, Germany
Jörn Ungermann, Philips Research Laboratories Aachen, Germany

Abstract

Verification of system behavior by testing is one component to prevent systematic errors and thus improve the system reliability. FlexRay, a fault-tolerant, distributed communication protocol for automotive use in safety-relevant applications, constitutes such a reliable albeit complex system. We describe a coverage-based constrained-random testing approach employing a fully functional reference model, its use to test a FlexRay controller, and compare it with an independently developed directed test approach. Using the errors uniquely found by each approach, we show that the random approach outperforms the directed one resulting in less undetected errors for the cost of a higher error handling effort.

1 Introduction

Although in principle design and implementation errors are to be avoided a priori and not to be found a posteriori, reality shows that errors occur and need to be detected to achieve high quality products. Two competing simulation-based test approaches are employed for such error detection: Directed Testing, where test designers specify concrete test cases, and coverage-based random testing, where test designers rely on randomly generated input and have to determine whether the random input did test everything of importance.

For the FlexRay communication protocol the lucky situation arose that both approaches have been used for testing. A directed test approach is used by the FlexRay protocol conformance test [5] whereas a coverage-based random test approach was employed internally by Philips/NXP. Significant effort was spent on each of the approaches, so this gives the chance to compare "real life" examples of the two approaches.

In the following we will shortly describe both approaches in general and then in the context of the FlexRay protocol verification task. The subsequent comparison is based on the number of errors found by only one of the approaches clustered according to criticality and suspected reason for non-discovery. We will conclude with some observations on whether one of the approaches should be preferred over the other.

1.1 Verification methods

The verification of an implementation under test (IUT) shall assert that it fulfills its requirements or specification. There are at least two systematic approaches to achieve this. One is called directed-testing, the other coverage-based random verification.

For directed testing, a set of dedicated test-cases is defined, generally one test-case per given feature or functional requirement. Each test-case describes the configuration of the IUT, a set of stimuli, and a set of expected responses. Generally it only checks the behavior related to the feature under test. A test-case may vary its stimuli at the discretion of the test designer.

Coverage-based verification also starts with the features. For each feature a set of coverage metrics is defined, i.e. the verification test bench gets the means to determine that a certain aspect of a feature has been used by the IUT. These coverage metrics are combined with checkers that detect wrong behavior of the IUT. Checkers may consist of a reference model that is fed the same stimuli while comparing the outputs, or a set of rules integrated into the verification test bench. The IUT is then stimulated by random inputs until the coverage metrics indicate complete coverage [9]. Random stimuli are generally constrained to generate more 'sensible' input than generated by a purely random process.

The general advantage of directed testing is that one gets results early by writing test-cases for already completed features of the IUT. However, additional runs of the test bench do not deliver further results. Using coverage-based random testing, each run adds new stimuli to the IUT, potentially driving it into conditions not foreseen by both the designer and the verification engineer. In this way, the verification can meaningfully continue also after the coverage has been completed and further errors in the design can be discovered.

1.2 FlexRay and test environments

The FlexRay consortium was founded in 2000 by BMW, DaimlerChrysler, Philips/NXP and Freescale (formerly Motorola) to establish a new standard for "a dependable automotive network" [3]. Basic characteristics of the FlexRay protocol are synchronous and asynchronous frame transfer, guaranteed frame latency and jitter during synchronous transfer, prioritization of frames during asynchronous transfer, multi-master clock synchronization, error detection and signaling, and scalable fault tolerance [4].

The FlexRay Protocol Specification (PS) completely defines the expected behavior of the protocol using SDL diagrams with accompanying explaining descriptions. Nine different process types are defined, of which several are instantiated twice for dual-channel systems, totaling 15 concurrently running, inter-communicating processes [4]. Furthermore several mechanisms of the FlexRay protocol – like the startup and the distributed clock synchronization – require interactions between nodes and their processes. E.g., the FlexRay startup defines three different kinds of general behavior for a node. These roles express themselves in different state traversal in the SDL processes. The simplest of these requires the transition through at least five different SDL states within the protocol operation control process. In addition to these SDL states, internal variables also contribute to the complexity as do the state transitions of the other SDL processes.

The FlexRay Protocol Conformance Test (CT) is a directed test suite specified and implemented for the FlexRay Consortium to ensure a basic level of functionality and interoperability of FlexRay devices [5]. It is further described below.

The NXP e Verification Environment (NVE) for the FlexRay protocol is built around the specman/e toolkit by Cadence which allows coverage-driven constrained random testing [7]. It consists of a fully functional model of the FlexRay protocol engine (PE), a digital channel model, as well as abstract models of host and controller host interface (CHI) to send commands and data to the PE. Its concepts are described below.

The implementation under test (IUT) is an RTL (Verilog) representation of a full FlexRay protocol engine whose conformance to the PS is to be ensured. The RTL model was verified with the NXP VE and the resulting hardware was tested with the FlexRay CT.

2 The test approaches

2.1 The NXP e Verification Environment for FlexRay

The NXP e Verification Environment (NVE) for FlexRay is a functional, transaction-based, coverage-driven verification test bench using constrained random configuration and stimuli. It was created to verify the FlexRay protocol specification and then used to create and verify the FlexRay golden reference SystemC [6] model (also called Executable Model). Finally, it has verified the implementation of the FlexRay protocol engine for the NXP micro controller SJA2510. In the following we will describe its structure, its stimuli-generation, checkers, and verification plan.

2.1.1 Structure

Setting up constrained random tests can be easy if the IUT has a simple enough behavior (or is tested on a high enough level) so that simple, abstract models of the IUT can be used to determine its correct reaction to the random stimuli (see, e.g., [9, chap. 2.4]). FlexRay, as already described, consists of several interacting state machines and maintains an internal state reaching back up to four communication cycles in normal operation. This state influences if – and if not with what error – transmitted bits will be received. Due to several influencing factors, especially in multi-node topologies, calculating this state with anything else but a complete protocol model would most probably be more complex than such a model. Thus, the main component of the NVE is a complete, sample-accurate model of the FlexRay Protocol Engine (an eVC in Cadence's terminology). Driving this model with the same inputs as the IUT provides reference outputs to compare the IUT's output against. Additionally the model can generate protocol-conform stimuli on the bus and allows the "observation" of internal states of a black box IUT by inferring them from the internal states of the model.

Thus, to verify an IUT the protocol engine implementation of the NVE is put alongside the IUT as shown in **Figure 1**. Both models have the same upper and lower interfaces. One connects the model to a FlexRay bus component the other connects it to a controller host interface component. The communication bus and the host are represented by simplified models.

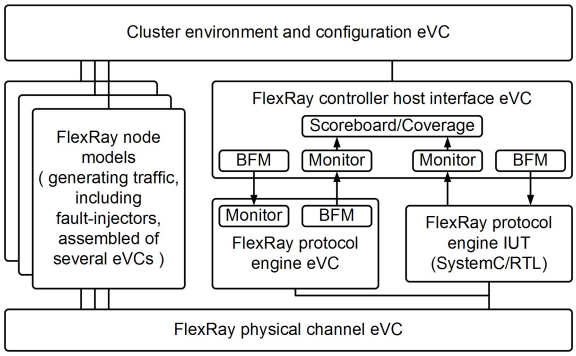

Figure 1: An overview of the NXP e VE, showing the PE-CHI interface in more detail

Each stimulus arriving from the bus or the controller host interface is forwarded to both the IUT and the pro-

tocol engine model. The outputs of the IUT and the model are collected in a dedicated checker unit and compared to one another at appropriate points in time. Since the behavior of FlexRay is characterized by the behavior visible to the outside, this checker is sufficient to determine the correct reaction of the IUT under a given stimuli. In the other direction, bus and the controller host interface are always driven by the IUT's outputs not by the model.

2.1.2 Stimulus generation

Three kinds of stimuli are needed to test the FlexRay IUT: Both interfaces (host- and bus-interface) can provide stimuli but also the configuration space of FlexRay can be seen as a stimulus.

The FlexRay protocol defines more than 70 configuration parameters. The relationship of these parameters is defined by 43 constraints, mainly inequality conditions. A unit within the NVE randomly generates a valid FlexRay configuration obeying these constraints.

Since the FlexRay protocol behavior is defined by several distributed algorithms interacting on multiple nodes of a FlexRay cluster, it is rather difficult to replicate typical or faulty FlexRay network traffic with just a single "stimulus generation" component. Therefore, the verification environment uses multiple instantiations of the protocol engine model attached to the bus model as stimuli generator for bus interface of the IUT. These protocol engines can be configured to behave in a faulty way and function then as fault injectors. A variety of faulty behaviors is implemented in the verification environment, ranging from a stuck bit in one of the internal counters, such as the macrotick or the slot counter, to very specific errors causing the protocol engine to transmit multiple frames within a slot. In addition, a simple noise injector is placed within the channel that can inject periodic or stochastic noise independent of any fault injector in the system. In this way the verification environment simulates a FlexRay cluster, to which one or multiple IUTs can be attached.

Stimuli generation on the host side is simpler in comparison since the host transmits few commands and data to its protocol engine. Several simple specman sequences replicate the behavior, not the functionality, of a real controller host interface. One sequence is responsible for generating the command flow to the protocol engine causing it to enter the various operation modes like READY, STARTUP or WAKEUP. This sequence can operate in various modes from typical behavior to completely random commands at hopefully inconvenient points in time. Further a random traffic generator supplies the protocol engine with data to transmit and includes the ability to cause slot collisions on the bus.

Since stimuli generation is done by using multiple protocol engine models (see above), the component modeling the bus can be kept relatively simple. It introduces various effects that are expected from a electrical physical layer while not trying to model electrical behavior itself. First, it adds propagation delay to transmitted signals depending on the transmitter and the receiver, thereby defining the topology. Second, it shortens the transmission start signal also depending on a configuration matrix simulating certain effects active stars have on signals. Finally, it can shift signal edges in both stochastic and consistent ways and/or add glitches to stress the FlexRay decoding units.

These stimuli are fed into the IUT until one of several predefined conditions of the simulation is met. Mostly, this is a certain number of cycles without 'interesting' events to have passed after the WAKEUP/STARTUP phase has been concluded. The topologies and configurations are randomized in a way that most simulations are kept short, but that still a sufficient number of simulations of several seconds of real time or of complex cluster configurations occur.

2.1.3 Checkers

Checking for correct behavior in the used approach in principle consists of comparing all output signals of both models for equivalence. However, analyzing a low-level "error in bit edge at 131623456ps" for possible causes is difficult for a human. Thus, all signals transmitted to the bus are decoded again by a copy of the decoder, which is already contained in the PE model, and compared in their decoded state. Thereby the checking is performed on transaction level, where it is appropriate.

Also it is not reasonable to expect the RTL implementation to perform all calculations and result delivery within a single clock tick as the NVE is capable of doing[1]. Therefore, certain checkers allow a specified delay for the output from the IUT compared to the NVE model which always responses within the minimal allowed delay. In addition to this delay of calculation results, also certain state transitions of the protocol engine can only take place as result of calculations. Since it is important for the behavior of the protocol engine whether certain other signals arrive before or after this state transition (e.g. the reception of a collision avoidance symbol in the coldstart collision resolution state in contrast to receiving it in the succeeding coldstart consistency check state), the verification environment has to delay its own state transitions according to the IUT using debug lines.

2.1.4 Verification Plan and coverage metrics

The basic specification for a verification is the verification plan. This plan lists the functional features and the interfaces of the FlexRay protocol engine, which have

[1] For most cases the FlexRay Protocol Specification actually allows a certain interval for it.

to be tested by the verification environment. The protocol engine model of the NVE is annotated with many coverage items, which are linked to one or several of these functional features of the verification plan. These coverage items are triggered upon certain events, e.g. events corresponding to a given SDL state transition, but also record internal variables of the clock synchronization like the number of received synchronization frames or the current rate correction term. The checkers also generate coverage values concerning which output the IUT has generated as well as its input stimuli. The last major coverage group is concerned with the FlexRay configuration setup. The verification plan accumulates all these single items and allows high-level statements like that e.g. the clock synchronization works but that the startup state machines have not yet been completely verified.

2.1.5 Verification work flow

The initial problem is to create a correct reference model to compare the behavior of an IUT against. To this extent two different implementations of the FlexRay protocol engine were created [2] as shown in **Figure 2**. One implementation, the SystemC model mentioned in section 2.1, followed closely the internal organization of the SDL diagrams, mapping each SDL state to a clearly defined portion of the source code. As far as possible variable names and source code contained in the SDL were reused for this. This made the adoption of changes of the protocol specification into the model very simple. It also allowed easy identification of errors in the SDL, when such errors were found in the SystemC model.

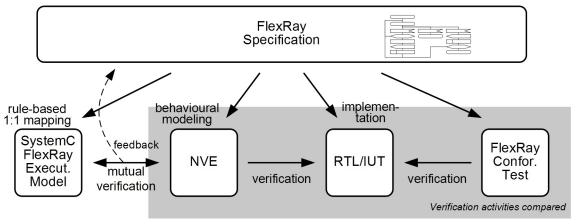

Figure 2: The complete process from specification to product

Since even in doing such a simple transfer, errors can be introduced that are not originally contained in the source material, a second implementation was created as part of the NXP e Verification Environment. This second implementation did not closely follow the SDL diagrams but instead implemented the desired functionality, making the best use of the special capabilities of the 'e' language such as sequences and aspect-oriented programming.

Development and cross verification of the NVE took longer than expected (see section 3.3). To counter this and as a quick feedback mechanism the IUT design team employed some directed tests as a primary verification mean (not shown in Figure 2). Most IUT versions were verified with these directed tests first before being subjected to the NVE. Finally, the FlexRay controller was tested with the FlexRay conformance test for specification conformity.

2.2 The FlexRay Conformance Test

The FlexRay Protocol Conformance Test (CT) is tasked with ensuring compliance of any FlexRay controller with the Protocol Specification. The intention is that compliance will minimize the risk of communication failures of a controller as well as interoperability problems between devices of different vendors. The CT contains in its current 2.1 version 276 directed test-cases. Many are to be executed with 3 sets of parameters and some with additional internal variations. It was specified and implemented by FlexRay Protocol CT work group in collaboration with subcontractors.

The test-cases of the CT are designed to completely test a device with respect to the FlexRay Protocol Specification based on "tag coverage". Tag coverage means that each of the tags attached to transitions in the SDL diagrams needs to be reached at least by one test-case. Thus, tag coverage is basically branch coverage of the specification (not of the device under test). In addition, completeness was checked using the SOVS approach[8], which verifies if every important combination of input and output parameters is tested.

The CT can test either hardware devices or a software (SystemC) representation of the FlexRay protocol, the so called Executable Model. Testing of the Executable Model was used as a quality gate for the CT to reduce the likelihood of test-cases failing for compliant devices.

2.3 Interdependencies of the CT and the NVE

NVE and CT development was largely independent although some interrelationships exists. One of the authors reviewed an early version of the CT specification (V0.1) and pointed out several coverage holes, some of which were taken into account. Also the final editing of the CT specification was taken over by the FlexRay CT work group of which one author is a member. However, the goal during that stage was to ensure a correct CT (i.e. no falsely rejected devices) not the increase of coverage.

In the other direction the coverage of the NVE was analyzed against the test-cases of a pre-final version version of the Protocol Conformance Test (V0.9.5). It turned out that the contexts of several test-cases were insufficiently covered by the NVE. The reason for this is that the CT test-cases are designed to satisfy a structural coverage metric (see section 2.2) whereas the

NVE coverage up to that point was functional. We identified nine clear coverage holes and further analyzed the five most critical of them. For this we designed an error for each of the holes which would be triggered only in the context of the (assumed missing) coverage. These errors were implemented one by one into the SystemC model and the thus intendedly erroneous model was then verified using the NVE. Two of the five errors were detected by the NVE several times within 2000 runs, the other three not. Although this proves that random testing will find errors in areas overlooked by the test designers, structural coverage measures were added afterward to the NVE as well as constraints and error injections to fulfill them.

3 Comparison of approaches

To evaluate the two test methodologies against each other we compare those errors uniquely found by only one of the environments, i.e. not detectable by the other test environment in its current state. Other analyses often compare achieved coverage values [10] or total number of errors found [1]. The former is in our opinion not the real goal of a verification environment and the latter is problematic in our case as the different approaches were partially applied sequentially to the same IUT (see section 2.1.5). For our analysis we only consider the errors found in the RTL implementation, as e.g. problems found in the FlexRay Protocol Specification (PS) were a side-effect and are difficult to compare since directed tests detect such problems mostly during specification and not during execution. The comparison is based on data from two bug databases: One containing the errors found by the NXP VE and the other containing the errors detected by the primary directed tests mixed with the errors detected by the CT[2]. We will start with the results of the NXP VE.

3.1 Errors only detected by the NVE

Using the coverage-based approach in excess of 126 errors[3] were detected. 73 of these 126 are errors with respect to the PS and could therefore in principle be found by the CT also. The rest divides itself mostly between documentation errors, synchronization problems between RTL and VE, and deviations of the RTL from an additional, stricter protocol specification agreed as design goal internally.

Of these 73 potentially detectable errors the error symptoms as well as the error causes were analyzed to determine which test-cases of the CT would detect them. For 23 errors we found such test-cases although one test-case would only detect the relevant error if all time deviation allowed by the test-case was used up by other, legal means. That is sufficiently improbable to not count that error as CT-detectable. Thus, the remaining 51 errors can not be detected by the directed test approach (see **Table 1**).

All errors found by NVE	126
Potentially detectable by CT	73
Actually detectable by CT	22 (+1)
Errors not detectable by CT	51 (-1)

Table 1: Errors found by coverage-based random testing

3.1.1 Criticality of undetectable errors

Having 51 undetectable errors poses two interesting questions: Why does the directed test approach not detect these errors? And: Are these errors relevant or do they just represent nearly improbable situations? To determine the relevance of the CT-undetectable errors we rated them similar to a Failure Mode Effect Analysis (FMEA): according to probability of occurrence and expected consequences. Probability of occurrence is rated 1 to 4 from very rare (“triggered only by some well-timed commands or bus events”) to basically always (“occurs at least once in basically every run of a system”). The consequences are rated from 1 to 6; we distinguish between informational effect, i.e. only secondary data like sync frame identifier lists, is disturbed, which result in a rating between 1 and 3 depending on how probable someone will want to use that information while it is damaged, and functional effects, which are rated from 4 to 6 depending on length and severity of the loss of function and on whether other nodes may be influenced. Criticality is then the product of both ratings.

We classify errors as important starting from a criticality of 9 (common occurrence with effect on normally used informational data) and as somewhat important with a criticality of 5 to 8 (5=very rare occurrence with long-term, local, functional effect). Errors below a criticality of 5 are deemed unimportant. Overall there are six important undetectable errors, the most interesting of them shown in **Table 2**, 22 errors of medium importance and 23 rather unimportant errors.

3.1.2 Reasons for non-detection

To get an indication why the directed test approach can not find these errors, we analyzed what was lacking from it to find them. We especially searched for test cases which would ’nearly’ find an error but failed to do the last bit necessary. In advance we expected results like "lack of parameter coverage", "lack of topology coverage", "lack of negative coverage" and "lack of state coverage" based on the perceived shortcomings of

[2]Alas the CT-detected errors were insufficiently marked and as such can not be determined separately anymore.

[3]About 5 errors reports got lost in the beginning while switching from research to production-level reporting tools.

Name	Context	Symptom	Reason for non-detection
Frame distortion	Bit edges in a frame are shifted within allowed ranges.	Frame is not decoded.	CT design decision
Rate correction in INIT-SCHEDULE	Cluster has nodes during the startup sequence with differently running clocks.	Node calculates wrong rate correction value; worst case: node drops out of startup.	Tested only in NORMAL_ACTIVE (i.e. lacking coverage).
Boundary crossing	PE tries to transmit over a slot boundary and is stopped by an internal check mechanism.	Transmission is stopped only 4 bits after boundary possibly disturbing others.	Tested only with byte granularity not with bit or sample granularity (i.e. incomplete checkers).
Offset correction overflow	PE needs to shift its cycle by more than 2^{13} clock ticks.	Offset is applied negatively, node drops out of synchronization.	No configuration tested which allows large offset corrections (i.e. lacking Parameter space coverage).

Table 2: Excerpt of important errors detected by the NVE but not detectable by CT

the CT specification . Surprisingly, the first two categories (aggregated into "Configuration complexity" in **Table 3**) had basically no effect.

The reasons for errors to be overlooked by the CT can be clustered into five classes (see Table 3). The "Technical restrictions" class comprises cases which are difficult to check in a general test due deviations allowed by the PS (see [5, chap. 1.8]), but which the NVE finds by synchronizing its white-box model with the specific RTL model using debug lines. Since an error of medium importance is contained in this class, a solution for a generic test would be useful.

Reason for non-detection	Number of cases	Percent
Technical limitations	4	8%
Configuration complexity	1	2%
Design decision	17	33%
Lack of Coverage	27	53%
Lack of Checkers	2	4%

Table 3: Causes for errors to be not detectable

"Configuration complexity" comprises the fact that the configuration and setup space of FlexRay is substantial and therefore lack of coverage for some combinations is to be expected. It is surprising that this class is empty except for one – albeit important – example. Either the CT approach of testing each parameter high, medium and low mainly within a simple topology is working or it is improbable that errors are caused only by certain topologies or configurations. The later is supported by our observation that improvements in error-injection result in more new errors found than enabling additional complicated node topologies.

"Design decision" contains the cases in which an error is not found due to a decision made during the specification of the CT. During specification phase it was stated by the specifying companies that they would not design test-cases checking for the absence of behavior, especially not if triggered by not "normal" actions (e.g. a command sent to the PE while it is in an incompatible mode). Also seemingly there was at least an unconscious decision to not test situations where states are entered for a second time. Both decisions are understandable since otherwise the number of test-cases would have grown significantly. However, this category is also the reason for 17 of the 51 non-detectable errors.

"Lack of coverage" and "lack of checkers" enclose the errors remaining undetectable due to an oversight during the test specification and not due to one of the above reasons. The major contributor here is the fact that most things are tested only once or in very few contexts. E.g., correct initialization is often not tested and many test-cases ignore any state before NORMAL_ACTIVE, the protocol's main state. The "lack of checkers" class contains the errors where the error triggering context is reached by the CT but there is no check which would notice the error.

Overall even when counting the first three classes as excused, 29 non-detectable errors remain, five of them important.

3.2 Errors only detected by directed tests

For comparison we also analyzed the errors found by directed tests (either the CT or the primary directed tests) whether they could be found by the NVE. Since we received mostly versions of the IUT successfully passing the primary directed tests, the NVE did not get the chance to discover a large set of errors. For this reason, we analyzed the last 152 entries in the main defect database. Filtering out doubles, change requests, errors originally detected by the NVE, and errors not related to the protocol specification, 73 errors remain which in principle can be found by the NVE.

This analysis is not as straightforward as the previous one since random tests contain a stochastic component so it is sometimes difficult to decide whether an error context is reached. Therefore, we classify the errors found by directed tests into five categories. Four categories based on decreasingly reliable coverage indications within the NVE for the error context and one category based on the existence of sufficient checkers for the error effect:

Full coverage: Coverage values exist from the NVE which indicate that exactly the context triggering

the error is reached. Also the checker detecting the error effect exist and are working.

Weak coverage: Coverage values exist which indicate that a sufficiently small range of contexts containing the triggering context is hit sufficiently often. This is often caused by two necessary preconditions each of which has its own coverage metric but missing cross-coverage. Checkers are working.

Implied coverage: For at least a part of the context no coverage metric exists but the design of the verification environment implies that the context is reached (e.g. the noise injector always randomly runs, so there will be tests where noise in a specific slot is tested given enough runs). Checkers are working.

No coverage: There is no reliable indication on which amount of coverage exists for the triggering context of an error. This doesn't necessarily mean that an error will remain undetected forever but that it is impossible to predict a number of runs necessary to find it with high probability.

No or dysfunctional checker: The checker, which is supposed to detect the deviating behavior of the IUT, is either non-existent or incapable of detecting the deviation. This category also does not guarantee that an error remains undetected since follow-up errors overlooked in analysis could be detected by other checkers, but it is very probable.

Thus, the last two categories are the "'error probably not found"' categories whereas errors in the other three categories are probably found. The number of errors classified into each category can be seen in **Table 4**. Six of the seven errors in the "Dysfunctional checker" category stem from the same two checkers hampered by an IUT-NVE synchronization problem. This problem requires the checkers to accept a range as comparison result that is bigger than the erroneous deviation. Its existence is known for some time and caused by a slightly different interpretation of an imprecise part of the protocol specification between NVE and IUT team. These errors will not be found by the CT for the same reason. Depending on whether these six errors are counted, there are either 4 or 10 errors detected by directed tests, which are not detectable by the NVE. All of the undetectable errors are of low importance according to the classification of section 3.1.1. This probably stems from the fact that the NVE primaryly uses functional coverage metrics and errors of high importance have to significantly influence the IUT's functionality.

Category	**Number of errors**
Full coverage	30
Weak coverage	11
Implied coverage	22
No coverage	3
No/Dysfunctional checker	7
Overall	73

Table 4: NVE classification of errors found by directed tests

Overall, in equal sized sets of errors discovered by the respective other approach (73 errors) the directed test environment can not detect 51 errors, 6 of which are important, and the coverage-based random test environment can not detect 10 errors all of which are of low importance.

3.3 Effort

As discovering errors is always also a question of allocated resources the effort spent in building the respective test environment needs to be considered. For the CT specification and implementation (when excluding hardware and driver costs) quoted effort is around 60 man months. The actual effort is probably higher since both specification and implementation experienced time overruns. About 2/3 of the work went into implementation, 1/3 into specification.

The effort required for the NVE is more difficult to determine as it was built as part of a project with several other activities. Our best guess is that around 3 man years were used to design and build the NVE. Thus, the higher amount of errors discoverable by the coverage-based random test approach was not bought by spending more effort in its realization.

Next to building the test environment there is the question of extensibility. Possible extensions for the CT include adding more configuration parameter sets, bit rates, or clock speeds, and including new protocol features. Adding configuration parameters to a directed test approach requires changing the specification and implementation of existing test-cases, resulting in significant effort. On the other hand, the NVE already includes this feature since it potentially tests all feasible parameter combinations. The situation is similar for additional bit rates or clock speeds although the NVE in this case might require some changes to its configuration constraints. Finally, for adding new functionality the effort is probably comparable, since the CT needs to specify and implement new test-cases whereas the NVE needs to adapt its internal model and coverage metrics.

One effort not yet looked at is the work required to analyze an error indicated by the test environment. No records were kept on this and thus we can not provide hard data but the impression is that error processing causes about twice as much work for the NVE than for a directed test environment for the following reasons:

1. No indication on the context of an error exists. In a directed test environment the best case for an error is that the error is triggered directly by the context tried to reach with the test-case and its cause is related to the feature the test-case is intended to

test. For a coverage-based random test environment using comparison of models like the NVE an error report states that at time x both models had an unacceptable deviation in some output y. Thus, normally the context triggering the error and the cause of it have to be determined by analyzing log files of model behavior.

2. A workaround has to be added to continue verification. When errors are not immediately fixed one by one but only in batches, a directed test can forgo to run the test-cases which triggered an error. For random testing this is not possible since it is not predictable what contexts a random stimulus will reach. To avoid that test runs are cluttered with known errors, either checkers have to be adapted such that a known error is no longer indicated or constraints need to be changed so that the triggering context is no longer reached.

In summary the effort for building and extending the coverage-based random test environment is less or equal to the directed test environment whereas the effort for handling errors indicated by the environment is probably higher.

4 Conclusions

Our comparison of a coverage-based random test approach (in form of the NVE) with a directed test approach (the FlexRay CT) shows that even with a professionally set up directed test the random test can discover significantly more and more important errors undetectable by the directed test environment than vice versa. As shown in section 3.1.2 the reason for this does not seem to be that the random test environment can effortlessly test various configurations and node topologies nor that the model comparison approach allows to cover internal corner cases. Rather it seems to be caused by the inability of directed tests for a sufficiently complex protocol such as FlexRay to test all possible triggering contexts of an error within a limited set of test-cases.

Using a complete functional model as the core of the constrained-random test environment further entails the advantage that stimuli representing arbitrary complex topologies can easily be generated and extensions tend to be cheaper than with directed tests.

Despite this better ability to detect errors, there are at least two shortcomings of coverage-based random tests, which will hinder their usage as, e.g., conformance test. First, as shown in section 3.2, mainly probabilistic statements can be made on whether a given scenario is covered by a coverage-based random test. Since the behavior of the whole cluster of node models is influenced by the behavior of the IUT this problem can not be circumvented by specifying certain random seeds to be used. This could lead to vendors complaining that a constrained-random CT did not test a competitor's device for a certain, seldom occurring error.

Second, the necessity mentioned in section 3.3 to change checkers or constraints to prevent known errors from flooding the error reports could cause random test CTs to require more time and more test runs since any run with at least one error could hide another error behind it and therefore contains the possibility of more errors detected once the first error is solved.

Thus, until these problems can be solved coverage-based random testing will probably not replace directed tests for conformance tests. However, its usage for vendor-internal verification of designs in addition to a conformance test prevents delivering products with otherwise undetected important errors.

References

[1] BARTLEY, M. ; GALPIN, D. ; BLACKMORE, T.: A comparison of three verification techniques: directed testing, pseudo-random testing and property checking. In: *Proc. Intl. Design Automation Conference (DAC)*, 2002, 819–823

[2] BAUMEISTER, M. ; FUHRMANN, P. ; ARMBRUSTER, F.: Taking concept models from standardization to silicon. In: *Automotive Electronics and Systems* Special Issue FlexRay (2005), S. 25–27

[3] BRACKLO, C.: Flexray standardization overview. In: *Automotive Electronics and Systems* Special Issue FlexRay (2004), S. 4–5

[4] FLEXRAY CONSORTIUM: *Flexray Communications System, Protocol Specification V2.1 Revision A*. 2005

[5] FLEXRAY CONSORTIUM: *Flexray Communications System, Data Link Layer Conformance Test Specification V2.1*. 2006

[6] IEEE: *IEEE Standard SystemC Language Reference Manual*. IEEE, 2006 (IEEE Std 1666)

[7] IMAN, S. ; JOSHI, S.: *The e Hardware Verification Language*. Kluwer Academic Publishers, 2004

[8] LAWRENZ, W. ; FISCHER, F. ; HOFFMEISTER, K. ; SCHEURER, M.: Leveled Conformance Tests - A Must for Interoperability in Networked Systems. In: *Proc. Intl. CAN Conf. (ICC2000)*, 2000, 10–19

[9] PIZIALI, Andrew: *Functional verification coverage measurement and analysis*. Kluwer Academic Publishers, 2004

[10] ROMERO, E. ; STRUM, M. ; CHAU, W.: Comparing two testbench methods for hierarchical functional verification of a bluetooth baseband adaptor. In: *Proc. Intl. Conf. on Hardware/Software Codesign and System Synthesis*. Jersey City, 2005, 327–332