HYBRID TESTING AND VERIFICATION TECHNIQUES FOR A COGNITIVE RADIO SYSTEM

Xueqi Cheng  Nannan He  Michael S. Hsiao
Department of Electrical and Computer Engineering, Virginia Tech, Blacksburg, Virginia, 24061, USA
(xcheng, nhe, mhsiao)@vt.edu

ABSTRACT
In order to enhance the reliability of software defined radio and cognitive radio (CR), we designed, tested, and formally verified a software module called mask verifier to guarantee the legality of the output from the embedded software radio system with respect to the regulations. In order to validate the correctness of the mask verifier, we propose a hybrid two-stage testing and verification framework for validating the reliability of a CR system. Software Bounded Model Checking (BMC) as a formal verification technique is used in both stages. In the code validation stage, we combine unit testing with BMC for validating the mask verifier. In the other stage, we incorporate mutation testing with BMC for test suite refinement to improve the next iteration of code validation. Experiments show that such a hybrid framework is effective in uncovering hidden bugs and proving the correctness of the code with reduced computational costs.

KEY WORDS
FCC, cognitive radio, mutation testing, model checking

1. Introduction
In recent years, due to the advantages of the flexibility, configurability and adaptability brought by Software Defined Radio (SDR) and Cognitive Radios (CR), they have been increasingly used by the military communication services, and are quickly expanding to other radio communication applications in the commercial and civil government sectors.

The Federal Communications Commission (FCC) establishes regulations for radio transmission. For example, the waveform emitted from any Ultra WideBand (UWB) system must be subject to the frequency and power limitations set by UWB masks [1]. One main purpose of these regulations is to help avoid band interferences that could conceivably affect critical government and non-government spectrum users. All radios have to be approved by FCC that they comply with the regulations before they can be allowed to use. These include waveform power, frequency, etc., produced by the radio should comply with the regulations. Since such applications are security critical, they require the SDR and CR to be highly reliable with respect to these specified regulations and protocols.
Moreover, the essential feature of SDR and CR is that most of the functionalities of radios are performed via the core software, such as automatically sensing radio environment, flexibly defining the optimal waveform based on its own capabilities and user requirements. It even adapts itself to handle unexpected situations by incorporating some computational intelligence. Therefore, this core software can become very complex.

Motivated by the above discussion, the reliability of this complex embedded software system is critical to the safety operations of the underlying radios. An error in the code may potentially produce illegal operations or cause disasters. Therefore, the design of the safety critical modules and effective verification approaches are essential to ensure the system correctness. Software testing via test pattern simulation (testing in short) and formal verification are two main kinds of validation techniques. In most of current SDR and CR systems, conventional testing is still the main stream technique used, for example unit testing and integration testing, etc. In [2], the authors build a framework that can help the Model Driven Development (MDD) of the SDR system, so that the test patterns can be automatically generated from the models at the same time that the code is generated.

However, such testing methods cannot cover 100% of the possible input space, so it is difficult for it to prove absence of bugs. On the other hand, formal verification attempts to prove the implementation correctness with the complete state coverage, but it can be very costly. For example, model checking may take a long time to exhaustively explore all states. Therefore, a hybrid of both techniques may work better than applying each of them independently. The hybrid of testing and formal verification techniques has invoked much interest in recent years, especially the test pattern generation directed by model checking. In [3], test patterns can be generated by symbolic model checking the mutations of specifications. In [4, 5], the authors exploited the counterexamples generated from software model checking tools. But there are very few researches on applying the hybrid approach to verify the software in radios to be compliant with regulations.

In this paper, we first present a software module that is designed to guard the outputs of a CR system: Public Safety Cognitive Radio (PSCR) [6] with respect to regulations. We also propose a new two-stage testing and verification framework for guaranteeing the reliability of this guarding module. One stage aims for code validation; the other is for test suite refinement. In each stage, we apply a two-layered testing and formal verification
techniques. For formal verification, an automatic formal technique called Bounded Model Checking (BMC) is used. In the code validation stage, we combine unit testing with BMC. After the software passes the unit testing where easy bugs have been detected and removed, we apply the more expensive BMC to target the hard bugs in order to prove the software correctness. In this way, the validation could benefit from both the efficiency of testing and the completeness of formal verification. In the test suite refinement stage, we combine mutation testing with BMC to make use of formal verification to enhance the quality of the test cases with the counterexamples returned by the model checker. Finally, these two stages form a general framework that can enhance the system reliability through an iteratively refined test suite.

The remainder of the paper is organized as follows. In Section 2, we introduce our Cognitive Radio Reliability Framework. Two-layered testing and verification strategies are presented in Section 3. We report our experimental results in Section 4, followed by conclusion in Section 5.

2. Cognitive Radio Reliability Framework

In order to assure the conformance of cognitive radio systems to the regulations (in this Section, we’ll use FCC regulations on UWB system as an example), our cognitive radio reliability framework can be divided into two steps, each of which works at different system development phases. First, at the code implementation phase, we constructed a mask verifier to detect and report illegal radio waveforms to the cognitive engine according to the FCC regulations. Next, at the program validation phase, we apply testing and verification methods to functionally verify whether the code does what it is supposed to do.

2.1 Mask Verifier Architecture

Due to the stringent security requirement of radio systems, it is essential to guarantee that the waveforms emitted from a radio device are legal, i.e., under the permission of the given specification, such as the FCC regulations. We first embed a Mask Verifier (MV) module in the cognitive radio system PSCR as a guardian to prevent illegal waveforms from deployment. The task of the mask verifier is to detect and report the legality of each incoming waveform generated by the cognitive controller. As shown in Figure 2.1, when a target waveform, defined by a set of parameters such as frequency, power, bandwidth, etc., is suggested by the cognitive controller according to a series of objectives, it sends it to the MV for legality check. The MV then fetches the corresponding regulations (e.g., FCC) stored in a database, and uses it to check whether the waveform lies within the legal range specified by the regulations. As a result, the MV returns the legality result back to cognitive controller. If the waveform is illegal, the mask verifier also calculates and returns the illegal area of the waveform, which can be viewed as a feedback indicator on the degree of deviation that the illegal waveform has from the specification, and can further be utilized in the next iteration of cognitive optimization algorithm.

Using a separate database for the regulation storage enables the MV handling different specifications other than FCC with single MV code implementation, which enhances the modularity of the program.

2.2 Mask Verifier Validation

While the mask verifier attempts to check that the system does not generate any illegal outputs, the correctness of the mask verifier must be rigorously verified, since possible human errors may be inserted into the designing of security critical systems like the cognitive radio. Therefore, we perform software validation over the MV to further enhance the reliability of the system.

![Figure 2.2 Hybrid Testing and Verification Framework](image)

The proposed Hybrid Testing and Verification framework is illustrated in Figure 2.2. Basically, it contains two stages: Code Validation (CV) and Test Suite Refinement (TSR).

At the Code Validation stage, we apply a two-layered unit testing and formal verification scheme to verify that the functionality of the MV code is consistent with the specification, that is, the MV can generate correct legality result based on the given regulation. Given a set of test cases $T$, unit testing is applied to detect and remove easy bugs, when the unit testing no longer could find any errors in the MV code with respect to $T$, the validation flow is transferred to formal verification (e.g. bounded model checking), where the MV is checked against the corresponding formal specification through implicit exhaustive traversal within the input state space.

Next, at the Test Suite Refinement stage, we enhance the quality of the test suite constructed during the CV stage by a two-layered mutation testing and formal verification scheme. At this stage, we use mutants to emulate possible program faults. For any mutant that cannot be killed by mutation testing with respect to current test suite $T$, model checking is applied to formally verify the conformance of the mutant to the specification. If the mutant can be killed
by model checking, the corresponding counterexample returned by model checking will then be appended to $T$, thereby enhancing the thoroughness of the test suite. In addition, the formal verification can also be utilized as an effective method for equivalent mutant detection. The detailed information of our two-layered algorithms for both stages will be given in Section 3.

The MV validated at the CV stage will be used as a golden MV in TSR stage. On the other hand, unlike testing, which is unable to guarantee the performance of validation process can be improved. On the other hand, unlike testing, which is unable to guarantee the completeness of verification. For example, at the CV stage, we are able to take advantage of their complementary benefits: efficiency in testing and completeness of verification. For example, at the CV stage, we can integrate testing approaches targeting both mutant coverage and code coverage (e.g., branch coverage) with model checking to obtain a test suite with a higher mutant and branch coverage, as model checking can be used to check the reachability of a certain unreach branch by the existing test suite. In other words, by combining testing and formal verification approaches to the validation process, we are able to take advantage of their complementary benefits: efficiency in testing and completeness of verification. For example, at the CV stage, with a high quality test set, it is able to detect and then correct many program errors with the “unit testing + debug” loop, thereby reducing the number of times that the more expensive formal verification method needs to be invoked for each specification rule. As a result, the performance of validation process can be improved. On the other hand, unlike testing, which is unable to guarantee the absence of errors, formal verification is able to provide complete checking by implicitly exploring the complete input state space of the code, which guarantees the conformance of the code to each specification.

3. Two-Layered Testing and Verification

3.1 Unit Testing and Test Suite Construction

![Image](image1.png)

Figure 3.1 FCC Regulations and Test Case Generation

Figure 3.1 shows the FCC UWB operation limits for handheld UWB systems, legal waveforms are defined through a set of 2-dimensional ranges bounded by a certain frequency $f$ and power $p$. If we treat waveforms lying within each range as an equivalence class, we can sample points at the boundaries of each range as test inputs for the mask verifier function with respect to both dimensions respectively, illustrated as the red dots in Figure 3.1, since most of the program bugs happen at the boundary cases. For example, given a 2-dimensional range bounded by $[f_1, p_2]$, we select points at $[f_1, p_{2-1}]$, $[f_1, p_{2+1}]$, $[f_{1-1}, p_{2}]$, $[f_{1+1}, p_{2}]$ for every $f_1$ chosen at the boundaries of $[f_1, f_2]$ along the horizontal (frequency) axis. Bandwidth is another important parameter for a given waveform. To represent the boundary cases of a bandwidth with respect to the ranges defined in the FCC regulation, we sample three bandwidth values for each range, which are “completely within a range”, “right at the range boundary”, and “expand over the range”, as shown by the three waveforms in Figure 3.1.

Therefore, each test case for the MV function is a 5-tuple: <(modulation_type, bandwidth, freq, power), legality>, where the first 4 elements are the input parameters for the MV that define a given waveform, and the legality is the corresponding expected output of MV. We generate the values of freq, power and bandwidth based on the above boundary-value based approach, while enumerating all the possible modulation_types defined in the specification.

After this initial test suite construction, unit testing is performed to check whether the output of the MV code is consistent with the expected legality value for every test case. If any error is found, the validation session ends and the user/designer needs to go back to debug the code; otherwise, we transfer the MV to formal verification to check its correctness.

3.2 Mutation Testing

Mutation testing is an effective way to quantitatively measure the quality of a test set based on the assumption that a program is well tested if all simple faults can be detected and removed, since complex faults can be detected using the tests that detect simple faults according to the coupling effects between the two kinds of faults [13].

![Image](image2.png)

Figure 3.2 Regulation Database Mutations

We apply mutation testing to measure the thoroughness of our boundary-value based test set. Due to the modular architecture of the MV code, mutation is performed over two parts: one is code mutation, where the mask verifier code is seeded with a set of slight variations (or faults) to construct a set of mutants, each of which contains one fault; the other is database mutation, where every time, a data item (e.g. frequency, power limits) stored in the regulation database is changed slightly to emulate a mutated
regulation. As illustrated in Figure 3.2, for a given 2-dimensional range in the regulation, the power limit is lowered by $\alpha$ in (a) and the lower bound of the frequency range is extended by $\beta$ in (b), and thus generated two mutation regulations.

After mutant creation, the test set is used to execute each mutant MV function with either mutated code or mutated database to check whether the output can be distinguished from the output of the original unmutated code. If the outputs can be distinguished, we say that the mutant is killed. Otherwise, the mutant is passed to formal verification for further test set enhancement.

3.3 SAT-based Bounded Model Checking

As a formal verification technique, model checking proves whether a finite state system $I$ correctly implements the specification $S$ by systematically checking the satisfaction of $S$ on every state of the model defined by $I$. It has become increasingly popular in software formal verification mainly due to its automatic feature [7]. And it can produce counterexamples to show the erroneous behaviors when the system fails satisfying the specification. This information is very valuable for debugging. In our work, we also make use of the counterexamples for refining the test suite.

SAT-based bounded model checking [8] is a model checking technique that uses a SAT-solver to check the states within a bounded length $K$ transitions from a given initial state(s). When it is applied to formally verifying the software program written in ANSI-C, a typical checking procedure has the following steps. Given a program $CP$ and its specified property $P$, an asserted program $CP'$ is first built by formulating $P$ as assertions and inserting into $CP$. For example, a property requires the value of some waveform frequency $f$ is always between $v_1$ and $v_2$ after the program executes a set of statements $\{S_1, ..., S_n\}$ no matter what input values are given. It can be written as an assertion “assert($f \geq v_1$ && $f \leq v_2$)” and inserted after statement $S_n$.

Secondly, data variables and the control structure of the asserted program $CP'$ are modelled in bound respectively. More specifically, every program variable is modelled as a bit-vector with a certain bit-width $M$, which is determined by its data type and the actual bit width representation in the underlying processor. For example the integer data type is set to 32 bits wide in most existing embedded processors. And all loop structures and recursions are unwound to a certain bound $K$. When $K$ reaches the maximum iteration, we could completely verify the property. After the bounded modelling, we could obtain a bit-vector equation $E$ that symbolically represents the bounded program execution. Next, a Boolean formula $F$ can be generated through a circuit translation of the program operators in $E$, for example the integer addition operator “$+\text{v}$” in the code is translated to a 32-bit full adder.

Finally, a SAT solver is applied to checking the satisfiability of the formula $F$. Since the negation of the asserted statement is set as the target state, the property $P$ is proved if $F$ is un-satisfiable that means no execution can set the assertion false. Otherwise, the satisfying assignment returned by the SAT-solver is regarded as a counterexample that shows the erroneous behavior violating property $P$.

3.3.1 SAT-based BMC for Code Validation

At the code validation stage for MV, we focus on formally checking the legality property of MV that MV shall correctly return the legality of any waveform with respect to the given regulation. The first step is to formally specify the property based on some waveform regulations so to clearly define the meaning of legality. For the example of FCC regulation shown in Figure 3.1, we generate a formula $F.L$ in Difference Logic (DL) [9] as the formal property specification. It consists of Boolean combinations of inequalities with the form $x - y \leq b$ where $x$ and $y$ are scalar variables of double data type and $b$ is a double constant. This formula has two parts $F.L_l$ and $F.L_h$ for specifying the property of legal and illegal waveform separately. Since the FCC regulation refers to the relationship between frequency and power of a waveform, three scalar variables with double data type $\langle f_{\text{low}}, f_{\text{high}} \rangle$, $p$ are used in this property formula, where $f_{\text{low}}$ and $f_{\text{high}}$ represent the low-bound and high-bound frequency of a waveform and $p$ represents its maximum power.

Since the specification is an important aspect of verification, we also validate the correctness and completeness of this property formula from two perspectives. First, there shall not be an overlap case between $F.L_l$ and $F.L_h$ that a waveform shall not satisfy both of them at the same time. We validate this by checking the satisfiability of another formula $F_\emptyset = \neg(F.L_l \lor F.L_h)$). The specification is correct with no overlapping if $F_\emptyset$ is un-satisfiable. Second, the property formula shall cover all possible cases that for any waveform modelled by $\langle f_{\text{low}}, f_{\text{high}} \rangle$, $p$ shall satisfy either $F.L_l$ or $F.L_h$. We validate this by checking the satisfiability of $F_\emptyset = \neg(F.L_l \lor F.L_h)$. The property is complete if $F_\emptyset$ is un-satisfiable. We could use a Satisfiability Module Theories (SMT) solver [10] for Difference Logic over Reals to check the satisfiability of $F_\emptyset$ and $F_\emptyset$. However, this kind of solvers models the variables as unbounded integers rather than bounded bit-vectors and they often require specific input format. Therefore we build a short program with two assertions of the negation of the two formulas as in Figure 3.3, and perform assertion checking by BMC.

```c
void specValidation () {
    double flow, fhigh, p;
    Bool F_Ll = DL_1(flow, fhigh, p);
    Bool F_Lh = DL_2(flow, fhigh, p);
    assert (F_Ll || F_Lh);
    assert (F_Ll || F_Lh); }
```

**Figure 3.3 Code of Validating Property Specification**

After validating the property formula, we insert into MV the assertion “assert( legality $\lor$ F.L_l ) $\&\&$ ( ~legality $\lor$ F.L_h )” where the value of legality is computed by MV. Then we also perform assertion checking of the asserted MV’ by SAT-based BMC. Since the unrolling bounds of loop structures in MV reach the maximum loop iterations, we could completely verify the property with BMC.
assertion holds, MV is proved being able to correctly determine the legality of the waveform with respect to the regulation (e.g. FCC).

3.3.2 SAT-based BMC for Test Suite Refinement
With the golden MV (GMV) that is obtained after the code validation stage, we continue make use of BMC to determine whether each mutated MV (MMV) is functionally equivalent to the golden MV and uses the generated counterexample to help enhance the test suite in case they are not equivalent. Figure 3.4 illustrates how mutant equivalence checking is done by BMC. The asserted combined MV (CMV) is checked by SAT-based BMC. If the assertion holds, the sliced MMV is proved to be equivalent to the GMV with respect to the legality property; otherwise the counterexample returned can be used to refine test suite.

Figure 3.4 Mutant Equivalence Checking by BMC

Since each mutant contains only one error, the structure of the GMV and MMV are very similar. We make some simplification in the CMV to reduce the burden of BMC by exploiting this structural similarity. The basic idea is as follows. Given a mutant at line \( l \) in MMV code, we perform forward program slicing from \( l \) and get a sliced program \( SP \). Since the statements outside the \( SP \) have the identical computation to that in GMV, we only need to rename the variables defined by the statements in \( SP \) and combine it with GMV to generate the CMV. In this way, the size of CMV model can be greatly reduced.

4. Results

In this section, we discuss the results of applying our two-layered testing and verification algorithms on the mask verifier in the cognitive radio system at both CV and TSR stages. In order to demonstrate the generality of our approach, the mask verifier is checked against two different specification constraints: FCC UWB spectrum regulation and DySPAN regulation[14]. To run the cognitive radio system under different regulations, we only need to keep a single implementation of the mask verifier code that interacts with different regulation databases.

4.1 Mask Verifier Validation
Table 1 shows the results of mask verifier validation at the CV stage. The test set could either be the initial boundary-value based set or an improved test set generated by TSR stage from the previous iteration. Here, we only focus on the results of the former. For different specifications, we created different sets of test cases based on boundary-values of the corresponding frequency-power ranges defined in the specification respectively. The size of the test set is directly related to the number of ranges in the given regulation. For each input vector in the test case, we manually generate the expected legality output.

As shown in Table 1, during the CV stage, we first apply unit testing on the original mask verifier code with respect to the FCC test set, and found 1 error in the code. After diagnosis and correction, the modified mask verifier is sent for unit testing again, this time, it passed the testing over both FCC and DySPAN test sets; therefore, we apply model checking with a Software Bounded Model Checking tool called CBMC [11] on the modified mask to prove its correctness against the formal specifications. The BMC found another error in the MV code: an overflow error caused by arithmetic computations, which has been demonstrated as a corner case based on the returned counterexample, and is not easily caught by any generally created test set.

Finally, the CV stage ends since no more errors were found. The mask verifier code that passed the CV stage validation will be considered as a golden model for the TSR stage during which mutation testing and model checking are conducted.

4.2 Test Suite Refinement
At the TSR stage, we generated mutants by applying a set of mutation operators listed in Table 2, which is a subset of the Mothra mutation operators [12].

For the mask verifier code, a total number of 431 code mutants were generated using the above mutation operators. During mutation testing, we feed both the FCC test set and DySPAN test set into the mutated mask codes to check whether they behave differently from the mask verifier golden model.

As the results shown in Table 3, 272 out of 431 code mutants could be killed by the original FCC test suite. Likewise, 333 out of 431 code mutants were killed by the DySPAN test suite. The reason that the DySPAN test set was able to kill more mutants is likely due to the different shapes of ranges in two specifications: the 2-dimensional legal ranges in FCC regulation are continuously spanning over all the frequency values greater than 0, while the DySPAN regulation defines a set of discrete legal ranges, so that the test cases generated from DySPAN set was able to exercise some statements of the mask verifier code dealing with illegal frequency ranges while the FCC set couldn’t.
Those 78 code mutants that could not be killed by either test set are sent to the second-layer model checking for formal verification. According to the results, model checking was able to either kill or prove that the mutant is equivalent to the unmutated code for all of the 78 code mutants. The results further demonstrated the effectiveness of model checking in identifying equivalent mutants; nearly half of the mutants that could not be distinguished by mutation testing were proved to be functionally equivalent to the golden MV. Without this formal step, one may spend an extraordinarily large computational efforts trying to find a test to distinguishing these equivalent mutants. Among the 41 inequivalent mutants, there were two different cases according to the counterexample returned by the model checker. The counterexamples for 35 mutants reflect a common situation: the data overflow (or underflow) because of the bounded modeling of data value. This means that these 35 mutants can be distinguished only in the data overflow case. Since the data overflow is an error that can only be detected in execution, these counterexamples are not very useful. So we didn’t use them to refine the test suite. The remaining 6 mutants can be distinguished without considering data overflow. And the new test cases are extracted from the counterexamples and added to the test suite because they can distinguish the mutants that the current test cases can’t.

Table 3 Code Mutants Results (TSR Stage)

<table>
<thead>
<tr>
<th>Specification</th>
<th>Total Mutants</th>
<th>Mutation Testing</th>
<th>BMC (Inequiv.)</th>
<th>BMC (Equiv.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FCC</td>
<td>431</td>
<td>272</td>
<td>6+35</td>
<td>37</td>
</tr>
<tr>
<td>DySPAN</td>
<td>333</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

For the database mutants shown in Table 4, the mutation in the frequency bounds or power limits at discrete ranges in DySPAN can be easily caught by test cases sampled at the range boundaries. However, for regulation containing continuous ranges, like in FCC, even if a mutation occurs at a frequency bound, the new power limit defined by this new frequency bound would probably be annulled by the power limit defined by its left-side adjacent range (see Figure 3.2(b)), such mutants would be viewed as redundant. The 5 database mutants which could not be killed by the FCC test suite are also sent to the second-layer model checking for formal verification. All 5 FCC database mutants could be distinguished from the correct database using model checking, and the counterexamples can be used to refine the test suite as well. This result shows that the hybrid framework is highly effective, that the testing is able to quickly uncover many bugs without excessively invoking formal verification for each mutant and/or spec rule. The improved test suite generated at the TSR stage can also be utilized in the next iteration of code development to further expand the error detection capability of the unit testing at the CV stage, thus improve the overall efficiency of code validation.

5. Conclusion

Urged by the high reliability requirements from the security critical radio applications as well as the essentiality of the functionalities of core software in a SDR or CR system, in this paper, we designed, tested, and verified a mask verifier module as a guard mechanism to enhance the reliability level of a CR system. A new two-stage hybrid testing and verification framework is proposed to validate the conformance of the mask verifier to the specifications, such as FCC regulations. The two stages include: a code validation stage combing unit testing with bounded model checking for functionality correctness checking, and a test suite refinement stage to enhance the quality of the test set using two-layered mutation testing and bounded model checking. The two stages can be iteratively applied during the CR system development process to enhance the efficiency of code validation. The results demonstrate that our framework is able to effectively take advantage of the complementary benefits from both testing and formal verification, providing a general scheme for enhancing the reliability of CR systems.

Acknowledgement

This research was supported in part by NSF grants 0519959, 0524052, and by NJI grant 2005-IJ-CX-K017.

References