Automated Firmware Testing using Firmware-Hardware Interaction Patterns

Sunha Ahn  
Princeton University  
sahn@princeton.edu

Sharad Malik  
Princeton University  
sharad@princeton.edu

ABSTRACT

Firmware is low-level software which can directly access hardware and is often shipped with the hardware platform. This component of the system is increasing in scale and importance, and thus firmware validation is a critical part of system validation. Firmware validation relies on the interacting hardware components which are usually not available until the late design stages. This is generally addressed through co-simulating C/C++ based firmware code and HDL hardware models (including SystemC). However, this tends to be slow, and is further exacerbated by the large number of possible interleavings between the concurrent firmware and hardware threads. Typically, in the co-simulation, the scheduler, such as the SystemC scheduler, will only explore a single, or at best a small number of possible firmware-hardware interleavings and thus may miss critical bugs. In this paper we present an alternative approach to firmware validation that is based on automatically generating a test-set for the firmware with the goal of complete path coverage while considering its interactions with hardware and other firmware threads. It uses a service-function based Transaction Level Model (TLM) which has been used in the past for firmware-hardware codeign. The test generation is based on concolic testing which has been successfully used in software test generation. However, existing concolic testing tools are used for test-generation of sequential code, and cannot directly consider the interaction of other hardware/firmware threads with the target firmware thread during test generation. We address this limitation by exploiting specific interaction patterns between the firmware and hardware threads that can be analyzed from the TLM. We show how these patterns, along with the firmware and hardware threads are used to automatically generate a sequential program that is test-equivalent to the target firmware transaction and that can be used with a standard sequential program concolic test generator. The tests generated can be (i) directly used for the firmware transaction and (ii) account for the multi-threaded interactions. These interaction patterns are practically relevant as they occur often in practice in real firmware benchmarks such as Linux device driver code, and its interacting QEMU emulated hardware code. Finally, we demonstrate the efficacy of our techniques for these benchmarks through a practical implementation that is automated and built on top of Fram-C, a static code analyzer, and KLEE, a concolic testing tool.

1. INTRODUCTION

Firmware is low-level software which can directly access the physical memory space of its interacting hardware devices. This hardware-specific nature distinguishes it from higher-level software such as the operating system (OS) or application code which is device independent. This higher-level software communicates with the hardware via firmware. Correct functionality of firmware is critical and its malfunction while accessing critical physical memory can crash the OS or even the entire system. For example, bugs in device drivers were considered to be the cause of 85% of the failures of the Windows XP Operating System [21].

Firmware is typically fully verified with the hardware in the late design stages, but this significantly increases time-to-market. Further, fixing firmware-hardware bugs may need hardware updates, and doing this late in the design cycle is costly. To bring this earlier into the design cycle, firmware and hardware co-design with uniform modeling languages has been used. For example, SystemC has gained popularity for integrating firmware and hardware because of its ability to model hardware at higher abstraction levels [18, 20]. SystemC manages the concurrency between the firmware and hardware threads through its scheduler. The SystemC scheduler typically considers only a single schedule, or interleaving, of these threads. At best it may consider a small subset of the exponential number of possible interleavings, limiting the coverage obtained through the co-simulation and thus missing possible bugs. An alternative approach to firmware validation is to consider the use of automated software test-generation techniques. In recent years, concolic testing has emerged as a powerful software test-generation methodology [19]. However, this is largely limited to sequential code, and cannot be directly applied to firmware test-generation as it cannot consider the effect of the interacting hardware and other firmware threads. Ignoring this interaction results in an over-approximation of the set of possible firmware behaviors, and thus false positives during firmware testing. An excessive number of false positives can significantly limit its practical use. This has been partially addressed in recent work in firmware test generation [2]. This work introduced a service function-based Transaction Level Model (TLM), for modeling both firmware and its interacting hardware. This modeling framework captures the reactive nature of these interacting components. Typically, each firmware thread repeatedly responds to inputs from its interacting software or hardware layer by providing appropriate services. Similarly, the hardware components get requests from the firmware or physical environment and respond to these requests. Thus, in this framework, both firmware and hardware functions are a set of services. Each service, or a unit of work, is referred to as a transaction. Each
transaction is repeatedly executed in a loop in response to requests. An iteration of the transaction is called a transaction instance. A system is modeled as a set of concurrent firmware and hardware transactions. They show how the firmware test generation problem can be simplified for the case where (i) the hardware thread is a producer of the shared data and the firmware thread is the consumer of this data and (ii) all transactions are stateless, i.e., the execution of a transaction instance is independent of the execution of previous instances. For this case, they show how the effect of the interaction of the hardware threads on a firmware thread, over all possible interleavings, can be captured through additional constraints that can be added to the concolic test generation for the firmware. Besides the limited application to this case, the analysis to identify this case and the constraint generation of the concolic testing instances is manual in that work - which significantly limits its practical utility. In this work we address both these limitations. We consider both stateful and stateless transactions, and provide a fully automated test generation framework. Stateful transactions are challenging as a transaction instance can depend on a possibly unbounded number of previous instances. Thus, test generation may involve unrolling the transaction for multiple instances, or iterations. Thus far, there is no easy way to determine the path coverage for a transaction for a given number of iterations. In this paper we show how this coverage can be determined for a large space of interaction patterns. Conversely we can determine if a given number of iterations is sufficient for complete path coverage for this space of patterns.

Our implementation consists of first detecting specific interaction patterns using static program analysis. Then, our custom code generator uses these interaction patterns along with the firmware/hardware interaction patterns using a static program analysis. Then, our custom code generator uses these interaction patterns along with the firmware transaction generation for the firmware. Besides the limited application to this case, the analysis to identify this case and the constraint generation of the concolic testing instances is manual in that work which significantly limits its practical utility.

We demonstrate the automatic detection of the specific interaction patterns of interest for the transactions using a single-threaded concolic testing tool. We use the interaction patterns to avoid exploring asynchronous interleavings of the concurrent threads. (§ 3-4)

Our algorithms guarantee the minimum number of iterations needed to cover all feasible paths or alternatively determine the path coverage for a fixed number of iterations. (§ 3-4)

We demonstrate the automatic detection of the specific interaction patterns of interest for the transactions using a static analyzer based on Frama-C. We describe a code generator that uses these interaction patterns along with the firmware and hardware transactions to generate a test-equivalent sequential program which can then be used with a standard sequential concolic testing tool. We describe how the concolic test generator KLEE is modified to provide code coverage in this context. (§ 5)

Figure 1: Rockbox SERIAL0 firmware transaction and its producer firmware transaction serial_bitrate

- We demonstrate the efficacy of our testing process on 15 transactions from two published real firmware benchmarks, TMP105 and Ethoc, from the Linux-QEMU platform. We highlight a bug discovered by this testing process in a published benchmark. (§ 6)

2. SERVICE-FUNCTION BASED TRANSACTION LEVEL MODEL

Our testing algorithm is based on the recently proposed service function-based Transaction Level Model (TLM) [2]. We briefly review this here.

2.1 TLM Definitions

2.1.1 Service Function

The functionality of a firmware program can be organized as a set of services. An illustrative example is the service that a USB driver thread provides. The USB thread waits until it detects the cable’s physical connection or disconnection. Then it broadcasts the USB cable connection status to other firmware drivers and completes the service. These other drivers also provide their own services, and they typically run concurrently. After the service is completed in response to a specific request, it ends, and a new instance of this service can start in response to a new request. During the service, it may interact with other concurrent service functions. This process is repeated indefinitely.

2.1.2 Transaction Level Model (TLM)

In TLM, a transaction for a service function has a clear start and end. Formally, a transaction is an untimed High-Level State Machine (HLSM) [23] with a start state (no incoming transition) and an end state (no outgoing transition). This is related to the notion of an extended finite state machine (EFSM) [5]. HLSM extends the Finite State Machine (FSM) by distinguishing the data states from the control states. Fig. 1 represents a simplified version of the Rockbox [1] iPod serial protocol transaction in the form of an HLSM. Rockbox is an open source MP3 player firmware code supporting various devices. In each state of Fig. 1, the transaction computes state transition functions and/or performs arithmetic operations.
computations for data update. Each execution of a transaction is referred to as a transaction instance. A transaction instance initiates its execution at the start state, then takes a particular path through the HLSM, and it eventually reaches the end state. In Fig. 1, an instance initiates, and at $f_0$, it checks if $rx$ is ready. If it is ready, then it takes the true path (right branch) and goes to the $f_0$ state. Here, $f_0$ assigns a hardware register value to $temp$, computes the transition function based on $newpkt$ and $autobaud$ values, and goes to the next state, either $f_c$ or $f_r$. Hence, the path that an instance takes is determined by the current state data and the state transition function. After this instance is finished at end state, the next instance can begin. Each transaction repeatedly executes its instances sequentially in this fashion. Intuitively, an instance is triggered by some input signal (e.g., $rx$ is ready), and while it is performing a service, it possibly accesses the hardware memory and updates data state such as $autobaud$ and $badbaud$.

The TLM for a system consists of a set of concurrent transactions which communicate via shared variables. For example, in Fig. 1, the SERIAL0 and serial_bitrate transactions share the autobaud variable. serial_bitrate writes the variable in $g_b$ and $g_c$, and SERIAL0 reads the variable in $f_b$ and $f_c$.

2.2 Transaction Interaction Patterns

In previous work [2], some specific characteristics of transactions were introduced. These characteristics captured from the TLM were shown to be beneficial for testing concurrent transactions in some cases as they help avoid considering all possible interleavings between concurrent transactions.

2.2.1 Stateless vs. Stateful Transactions

Transactions can be classified as being stateless or stateful. In a stateless transaction, an instance does not depend on the previous instances as the state of variables is not retained across successive instances. In this case, each instance depends only on new values read through the input or shared variables. Statelessness significantly simplifies test generation. To consider all possible behaviors of a stateless transaction, it is enough to iterate the transaction only once since this can exercise all possible paths depending on the new values read. For example, in Fig. 1, $g_b$ gets a fresh value every instance and depending on this value, the instance takes a path through $g_b$ or $g_c$. Hence, by considering different inputs for $g_b$, one can possibly exercise either path in the first iteration. To consider the effect of serial_bitrate on autobaud, one can simply summarize it as "the autobaud value can be either 0 or 2 at any point."

On the other hand, in a stateful transaction, an instance depends on the previous instances. For example, in Fig. 1, if an instance of SERIAL0 reaches $f_0$, it increases badbaud by one. Here, this instance needs to know the value of badbaud written in the previous instance. Then, if badbaud is greater than or equal to 6, the instance goes to $f_3$ and sets the badbaud value to zero. The next instance may read this new value. To sum up, an instance of a stateful transaction reads variables updated in any of its previous instances. Such variables are stateful variables.

2.2.2 Producer and Consumer Transactions

When a transaction $T_p$ updates a variable $x$ and this value is independent of other transactions, we say $T_p$ is a producer of $x$. For example, both SERIAL0 and serial_bitrate are producers for autobaud in Fig. 1. Moreover, when a transaction $T_c$ reads a variable $x$ updated by $T_p$, $T_c$ is a consumer for $x$ and $T_p$ and $T_c$ are in a producer-consumer relationship. SERIAL0 is a consumer for autobaud.

2.3 Transaction Interaction Patterns and Test Generation: Previous Work

2.3.1 Avoiding All Interleavings

In general, testing concurrent threads needs full exploration of all the interleavings to consider all possible paths through code. However, the previously introduced firmware testing method [2] based on the proposed TLM showed how this could be significantly simplified for a specific interaction pattern. The first observation was that in testing a target firmware transaction, the only interacting transactions of interest were those that could help produce values for this firmware transaction, $F$. These values determine the behavior of $F$ and thus the paths that can be exercised in it. Transactions that consume values produced by $F$ are not relevant for testing it since they do not determine its behavior. The second observation is that a stateless transaction needs only one iteration to explore all the behaviors. They combined these observations into an algorithm that could be used for testing $F$ when $F$ and all transactions that produced values for it were stateless. They showed how a concolic test generation tool for sequential programs such as KLEE could be used to generate tests for $F$ with full path coverage.

2.3.2 Limitations

The test generation algorithm is limited to stateless transactions which is the simple case. For stateful transactions, it may be required to run several iterations of a transaction, i.e. many instances, before a certain behavior or program path may be exercised. It is unclear what the path coverage is for a certain number of iterations. Further, in the previous method, the shared variables are limited to being written to by only the producer transactions. The case where $F$ may update the shared variables in addition to the producer transactions is not permitted. Finally, there is a lack of automation. While they present an algorithm to convert the system TLM to a sequential program $P$ that can be used by KLEE to generate tests for $F$, the analysis of the interaction patterns, and the generation of $P$ is manual. This lack of a broad consideration of interaction patterns and automation limits the practical application of their work.

In this paper, we address the gaps in previous work as follows.

- We cover a much larger set of interaction patterns. In particular, we explore testing stateful transactions with stateless producers. Further, our method covers testing stateless transactions with both stateful and stateless producers. Also, for all cases, in addition to the producers, the target transaction $F$ may write to the shared variables.
- We determine the path coverage obtained using a fixed number of iterations of the stateful transactions.
- All aspects of the test generation are automated including analysis of the interaction patterns and generation of $P$.

3. TLM BASED CONCOLIC TEST GENERATION FOR STATEFUL TRANSACTIONS: OVERVIEW

We now describe the main ideas for handling stateful transactions and avoiding generating all possible interleavings of $F$ with its interacting threads.

3.1 Challenges with Stateful Transactions: A Motivating Example
void SERIAL0(void) {
    temp = rx_readc(); // get a fresh value
    if(newpkt & autobaud>0) {
        switch(temp) {
        case 0x0FF:
            break;
        default:
            badbaud++;
            if(badbaud > 6) {
                autobaud = 2;
            }
        }
    return;
    }
    bool pkt = iap_getc(temp); // get a fresh value
    if(newpkt & !pkt) 
        autobaud=0;
    newpkt = pkt;
}

Figure 2: HLSM for the simplified version

In this section, we illustrate the challenges in testing stateful transactions through a motivating example.

3.1.1 The Example Transaction
Consider the example in Code 1. This code is a C version of the SERIAL0 transaction in Fig. 1 that has been simplified for presentation of the key ideas (Fig. 2 is a reduced version of Fig. 1). In this example, newpkt, autobaud, and badbaud are the stateful variables. At line 2, temp gets a fresh value in each instance by reading a register. Then, it checks the stateful newpkt and autobaud values that are updated in the previous instance. If the condition is true, the instance checks the value of temp. Assuming the value was not 0x0FF, line 8 increases badbaud by reading the old badbaud value updated by the previous instance. Therefore, badbaud is a stateful variable. Next, it compares badbaud with 6 (line 9), and if the condition is satisfied, sets autobaud to a constant value (line 10). This value will be used in future instances. Line 15 gets a fresh value depending on the status of its environment. Line 17 sets the autobaud value, and line 18 sets the newpkt value.

3.1.2 Possibility of Infeasible Paths
The goal of this work is to cover all feasible paths of stateful target transactions or target transactions with stateful producers. We start by considering how this can be done for the stateful transaction in the motivating example. As we can easily see, there are 6 paths in Fig. 2, an HLSM version of Code 1. Traditional unit testing for SERIAL0 will find tests for all 6 paths. However, some tests may require values for the stateful variables that cannot be obtained from any sequence of executions from the starting state. Thus, these tests cannot be used for this. Further, note that, due to the reactive nature of transactions, each transaction is continuously executed in a forever loop in its request to tests as follows: // initialize stateful variables;
while(1){ SERIAL0(); }
Depending on the initialization, there could be some infeasible paths in SERIAL0. Let us assume that the initial value of the stateful variable badbaud is 10, and during the first iteration, the instance passes through the bold path from st to f0. In f0, the badbaud value is read, increased by 1, and updated to 11. The following if condition becomes true. Next, the instance traverses f1 and end. In the next iteration, suppose the bold path is followed again. In f0, now badbaud becomes 12 and the if condition is true again. The dashed path from f0 to end is not taken again. In fact, from here on, any of the following instances cannot take the dashed path as the only place to update badbaud is f0, and this can only increase badbaud. Hence, the path from st to end via f0, f1, and f0 is an infeasible path. On the other hand, if the initial value of badbaud is, for example, 0, then this path (bold path + dashed path) can be feasible. Depending on the initial values of the stateful variables, feasible paths can be different for the same transaction code. Traditional unit testing methods that ignore this stateful behavior do not find feasible paths correctly. Overall, the challenge of testing stateful transactions is the need for possibly unbounded transaction unrolling from the initial condition to find all the feasible paths. Considering an unbounded number of iterations is practically impossible. What is desirable is that for a fixed number of iterations we can determine the path coverage obtained. This will also let us stop at the fewest iterations needed to obtain full path coverage. We show how this can be accomplished with our algorithm.

3.2 Overview of the Methodology
Given a TLM with a target firmware transaction $F$ and its interacting hardware and firmware transactions, our goal is to automate the test generation for $F$ with high path coverage. Our methodology for this is depicted in Fig. 3.

First, the Frama-C based automatic transaction analyzer takes as input a TLM with $F$. It outputs (1) whether $F$ is stateful, and if so, the list of stateful and stateless shared variables and (2) any producer-consumer relationships with other transactions, including the set of shared variables propagated from each producer to each consumer. These outputs are then used by a code generation algorithm that also takes as input the TLM with $F$. It generates a sequential program $P$ that is test-equivalent to $F$. A test generation algorithm using KLEE is used with $P$. These tests can be directly used for $F$. Further, the test generation program also provides the
coverage obtained for $F$. The code generator that generates $P$ handles stateless and stateful transactions differently. We now look at the specific algorithms for these cases.

4. STATEFUL AND STATELESS TEST GENERATION ALGORITHMS

We start by briefly introducing the concolic testing tool KLEE [4] as it is the execution engine for the algorithms. Concolic testing [19] performs symbolic execution [12] and concrete execution together to explore all feasible paths related to the symbolic variables. Say a variable $x$ is instrumented as being symbolic. While running this instrumented code, when KLEE sees a branch condition containing $x$, say $x > 0$, KLEE forks a new process to execute each path. One process follows the $x > 0$ branch with the path condition $x > 0$ attached to this process. The other process takes the $x \leq 0$ branch with the path condition $x \leq 0$. Every time KLEE sees a branch with symbolic variables, it takes both paths by forking a new process. Each path conjuncts the corresponding branch condition to the existing path condition. At the end, if there are $n$ paths in the code, there will be $n$ processes. Each process asks a constraint solver for a set of concrete values for the symbolic variables that satisfy its path condition. This set of concrete values, if one exists, is a test case for the particular path. This is how KLEE covers all the paths of a program. Other concolic testing tools such as DART [9] work in similar manner. We chose to use KLEE as it is well-maintained and open-source.

4.1 Testing Stateful Transactions

4.1.1 Main Idea

The test generation algorithm faces multiple sources of complexity when $F$ is stateful.

1. C1: With stateful variables, it needs to deal with a possibly unbounded number of iterations to exercise all feasible paths.
2. C2: With other interacting transactions, it needs to deal with the possibly exponential number of interleavings that can determine the state of the shared variables.
3. C3: For each path, it needs to deal with the constraint satisfaction of the path conditions.
4. C4: The number of paths may be exponential in the size of the program.

C4 is a characteristic of software testing in general. It can be handled by making only key branch conditions symbolic. C3 is handled through efficient constraint solvers such as SMT solvers that are part of KLEE. C2 can be made manageable for special cases. Specifically if the producer transactions for $F$ are stateless, then we can use the prior work which uses KLEE for constraining the values for the shared variables in $F$ to the values that can be produced by its producers [2]. For now, we will consider the restriction to this case. This will be relaxed in the next section where we consider stateful transactions. However, unlike previous work, we permit $F$ to also write such shared variables. The main contribution of this section is to show how C1 is handled using a concolic testing based algorithm, and how this is considered with the solution for C2 to restrict the values from the producer threads. The main idea is to use concolic testing by iteratively unrolling $F$, and at each iteration, testing if the path condition for paths not yet covered can be satisfied with the restriction on the values of the shared variables imposed by the producers. If a satisfying assignment for a path is found, it is not tested in future unrollings. Thus, each path is tested using the shortest test that will exercise it. Further, the algorithm can stop when all paths are tested, or we reach some pre-assigned bound $k$ on the number of iterations in the unrolling. At this point, we know the path coverage that has been obtained.

We now illustrate the algorithm on the reduced $SERIAL0$ stateful transaction in Code 1 interacting with the stateless $serial_bitrate$ producer in Fig. 1. The stateful shared variable $autobaud$ is also written by this concurrent producer. For inputs from the physical environment, we assign constrained values based on the environmental constraints such as for the $temp$ value (line 2) and the $pkt$ value (line 15) in Code 1.

**STEP1: Early Pruning of Infeasible Paths**

First, the algorithm performs unit testing for the stateful transaction $SERIAL0$ using KLEE. Instead of the stateful variables $autobaud$, $newpkt$, and $badbaud$, the algorithm substitutes symbolic variables $autobaud_{sym}$, $newpkt_{sym}$, and $badbaud_{sym}$ and calls $SERIAL0$.

![Figure 4: path1](image)

**Figure 4: path1**

![Figure 5: Testing stateful transactions with stateless producers: Rockbox SERIAL0 example from Fig. 1](image)

**Table 1: Path conditions for the 6 paths produced in STEP1 of Fig. 5**

<table>
<thead>
<tr>
<th>Path</th>
<th>via</th>
<th>Path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>path1</td>
<td>$f_0, f_k$, $f_{l_1}f_m$</td>
<td>pc1 = (newpkt_sym &amp;&amp; autobaud_sym&gt;0) &amp;&amp; (newpkt_sym &amp;&amp; pkt)</td>
</tr>
<tr>
<td>path2</td>
<td>$f_0, f_k, f_m$</td>
<td>pc2 = (newpkt_sym &amp;&amp; autobaud_sym&gt;0) &amp;&amp; (newpkt_sym &amp;&amp; pkt)</td>
</tr>
<tr>
<td>path3</td>
<td>$f_0, f_k, f_{l_1}f_m$</td>
<td>pc3 = (newpkt_sym &amp;&amp; autobaud_sym&gt;0) &amp;&amp; (temp == 0xFF) &amp;&amp; (newpkt_sym &amp;&amp; pkt)</td>
</tr>
<tr>
<td>path4</td>
<td>$f_0, f_k, f_{l_1}f_m$</td>
<td>pc4 = (newpkt_sym &amp;&amp; autobaud_sym&gt;0) &amp;&amp; (temp == 0xFF) &amp;&amp; (newpkt_sym &amp;&amp; pkt)</td>
</tr>
<tr>
<td>path5</td>
<td>$f_0, f_k, f_{l_1}f_m$</td>
<td>pc5 = (newpkt_sym &amp;&amp; autobaud_sym&gt;0) &amp;&amp; (temp == 0xFF) &amp;&amp; (badbaud_sym&lt;6)</td>
</tr>
<tr>
<td>path6</td>
<td>$f_0, f_k, f_{l_1}f_m$</td>
<td>pc6 = (newpkt_sym &amp;&amp; autobaud_sym&gt;0) &amp;&amp; (temp == 0xFF) &amp;&amp; (badbaud_sym&lt;6)</td>
</tr>
</tbody>
</table>
is run with symbolic values starting from a single thread. However, through its execution, KLEE explores the 6 possible paths in 6 different processes. For example, the unique process proc1 in Fig. 5 is dedicated for the path1 in Table 1. Similarly, path2 in Table 1 is executed in proc2, etc. For path1, in Fig. 4, only the concrete values of autobaud and newpkt satisfying pc1 can exercise it. This condition pc1 is built up while KLEE traverses path1. Every time KLEE reaches a branch, it adds (concats) the condition of the branch that it takes to pc1 (Fig. 4). Hence, in proc1, after running the instrumented SERIALAL0, the algorithm can get the corresponding path condition pc1. Similarly, each of the 6 processes has a corresponding path condition after STEP1. Table 1 shows the path conditions for all 6 paths. If a path condition, e.g. pc1, cannot be satisfied by the constraint solver then we know that path1 is infeasible as it cannot be reached from any state. This allows for early pruning of some infeasible paths.

STEP2 : Determining Set of Possible Producer Values In the rest of the algorithm the same code is executed in each process independently with different path conditions. Here, we will explain what is happening in proc1. After getting pc1, a function of *_sym variables, the algorithm runs the stateless producers of the stateful variables if there are any. In this example, the algorithm code calls serial_bitrate which writes the shared variable autobaud. In composing the transaction code for the producer serial_bitrate we use specialized library functions to read/write the shared variables. Additionally, we are using statically assigned shared variables. Whenever the producer writes on the stateless variables using the library function, the produced values along all possible paths in the producer are recorded during the instance. Hence, we can automatically gather the written values from serial_bitrate by calling the producer multiple times. Our algorithm calls the producer as many times as the number of writes on the shared variable in the producer transaction code. Each call, or instance, only goes through one path of the producer. Hence, this way, we are guaranteed to gather all possible produced values as KLEE will implicitly cover all possible cases. Thus, the additional bookkeeping of recording each written value in the producer, combined with KLEE’s ability to cover all possible paths through implicit enumeration of its symbolic variables enables any possible run of the producer to be accurately captured for use by the target transaction. Note that the set of output values from the producers can be either a small finite set or a larger range. In the former case, the shared registers representing device status values (e.g., pressed buttons) updated by the hardware producers would only have a number of possibilities (e.g., pressed or not pressed). In the latter case, values such as temperature or battery level will be defined as symbolic, and we will end up choosing a single value within the particular range.

The other processes, proc2-proc6, are similar in their composition with serial_bitrate.

STEP3: Iterative Unrolling for Path Testing This step is specific to the stateful transaction. Here, the algorithm checks if each of the 6 paths is feasible or not in each process. For example, in proc1 it checks if path1 is feasible on starting from the initial state. To do that, we now substitute the stateful variables autobaud, newpkt, and badbaud with concrete variables autobaud_conc, newpkt_conc, and badbaud_conc in SERIALAL0. More specifically, the algorithm initializes the autobaud_conc, newpkt_conc, and badbaud_conc variables with the initial values of autobaud, newpkt, and badbaud (in Fig. 5). Note that, in STEP1, the algorithm runs SERIALAL0 with symbolic values to represent all possible initial states. In contrast, STEP3 checks each path condition with concrete values for the initial states. Let us say that the initial value of newpkt, autobaud, and badbaud are 1, 1, and 4, respectively. At the first iteration iter1 in proc1 (Fig. 5), the first instance starts from st to f1, then goes to f2 since (newpkt&kautobaud > 0) is true (Fig. 2). Further, in the first iteration, assuming that the constrained value assigned for temp is 0x01, the instance takes path5. After the first iteration, the algorithm takes the concrete values (1, 1, 4, 0x01) which just exercised the first instance, and checks if pc1 ∧ (newpkt_sym == 1) ∧ (autobaud_sym == 1) ∧ (badbaud_sym == 4) is satisfiable or not (the algorithm selected a constrained value for pkt so that the pkt could be selected to satisfy the formula). proc1 checks the feasibility of path1 by finding concrete values of the stateful variables which can exercise path1. If the formula is satisfiable, it means that the concrete values of the stateful variables (1, 1, 4) can exercise path1. In this case, of course the formula is unsatisfiable as these concrete values are for exercising path5. Then, the algorithm unrolls another iteration. At the second iteration iter2, we repeat this. Now, the value of the stateful variables are (1, 1, 5) as badbaud was increased by 1 at the first iteration. Let us say that this time, temp was 0xFF. The second instance reaches f2 via f1, f3, and f4. Assuming the pkt is 0, the instance goes to f5, f6, and end (path3). After the second iteration, the algorithm checks pc1 ∧ (newpkt_sym == 1) ∧ (autobaud_sym == 1) ∧ (badbaud_sym == 5) again. The formula is still unsatisfiable, so we keep unrolling. In the third iteration, since autobaud is now 0, iter3 goes through f3 and f4. Since newpkt is now 0, this iteration takes path2, and keeps unrolling. Assuming that 1 is assigned to pkt during the third iteration and 0 is assigned to pkt during the fourth iteration, iter4 finally takes path1. This means that path1 is feasible. After iter4, the algorithm checks pc1 ∧ (newpkt_sym == 1) ∧ (autobaud_sym == 0) ∧ (badbaud_sym == 5), and since it is satisfiable, the set of concrete values that exercised iter4 is the test case for path1. The algorithm now quits unrolling in proc1. If the path is not found to be feasible after a given bound k on the number of iterations, we terminate our search.

Note that as we unroll the stateful transaction, we consider the possible updates from the producers as specified in the transaction composition in STEP2. While the stateless producers are running concurrently with the target stateful transaction, the target transaction may read the shared variable values updated from the producers. For example, SERIALAL0 may update autobaud to 0 at f3. Then, the next instance is supposed to take the false branch from f3. However, serial_bitrate can write on autobaud in the interim and this drives the next SERIALAL0 instance to f4 instead. As the producers for the stateful variables are stateless for the current case, the order of updates within the stateless producers does not matter. In STEP3, every time the target transaction reads its concrete stateful variable, it either reads from its old value or any of the producer values. For example, in f5, SERIALAL0 reads either the previous value of autobaud, or reads 0 or 2 (producer values). Similar to path1, the feasibility of path2-path6 is determined in the same fashion in proc2-proc6, respectively. Once a path is determined to be feasible, no further unrolling is necessary to test that path. Thus, the algorithm requires only the least possible unrolling for each path and provides the shortest possible test for each path.

### 4.1.2 Pseudocode

Code 2 provides the framework for the testing algorithm. In Fig. 3, once the automatic transaction analysis is done to determine the interaction pattern and other related information, our code generator composes a customized program based on this framework for stateful transactions. In Code 2, x[i] are the stateful variables, and NUM_SFV is the number of stateful variables. Lines 1-6 are for
for(i=0; i<NUM_SFV; i++) {
    klee_make_symbolic(x_temp);
    x_sym[i] = x_temp;
    x[i] = x_sym[i];
}

Call the target transaction with mode 0
for(i=0; i<NUM_SFV; i++) { 
    x_sym[i] = x[i];
    for each shared stateful variable with producers, v{
        for(j=0; j<NUM_OF_WRITES_on_v_in_p; j++){
            Call producer p
            klee_make_symbolic(choosen); klee_assume(0 <=
            choose < number of write during the instance);
            wv[j]=s_hw_v.writes[choosen];//stores a produced
            )           //value during the instance
        }
        Initialize the stateful vars x_conc[i] 
        for(i=0; i<NUM_SFV; i++){
            x[i] = x_conc[i];
        }
        while(1){
            klee_set_forking(1,1);
            Call the target transaction with mode 1
            for(i=0; i<NUM_SFV; i++){
                x_conc[i] = x[i];
            }
            klee_set_forking(1,0);
            if((x_sym[0]==x_conc[0]) && (x_sym[1]==x_conc[1]) && .
               && (x_sym[NUM_SFV-1]==x_conc[NUM_SFV-1])) break;
            k++;
            if(k > K_BOUND) klee_silent_exit(0);
    }
}

Code 2: Testing stateful transactions with stateless producers

STEP1. In lines 1-5, we make x_sym[i] symbolic and substitute x_sym[i] for x[i]. Then, we call the stateful transaction (line 6) so that all paths of the target transaction are explored in multiple processes. Each process now has a corresponding path condition. From line 7, the rest of the code is executed in the multiple processes independently. In each process, lines 7-9 restore a possibly changed x[i] during the particular path into x_sym[i]. Lines 10-18 are for STEP2. For each stateful variable v with stateless producers, we gather all possible written values for v from the producers and store them in the array wv. The rest of the code is for STEP3. Lines 19-22 initialize the x_conc[i] values and substitute them for x[i]. Then, in the while loop, the transaction is called with mode 1. Mode 1 means that every time the transaction reads the concrete stateful variables, it reads from the variables’ old values or the corresponding producer values. Line 27 restores x_conc[i] values after an iteration and then, line 30 checks the condition pc \(\land\) NUM_SFV (x_sym[i] == x_conc[i]), pc is implicitly conjuncted along the process. If the condition is true, it breaks out of the loop, else, it updates the iteration count. If a process is not able to find a test in an upper bound of k iterations, it terminates the search. Line 29 forces KLEE to choose the condition in line 30 to be true. This is a modification we made in our version of KLEE. In the regular version of KLEE, KLEE will try to explore both cases where the condition (line 30) is satisfied and not satisfied. However, as long as it is satisfiable, we do not care about the other case. In this case, we do not want KLEE to explore the false branch of line 30 needlessly. Hence, we modified KLEE’s source code, and if klee_set_forking(1,0) is executed. KLEE disables forking but prefers the true branch. If KLEE’s constraint solver decides that the true path is feasible, then KLEE only forks the true path, not the false path. If the true path is not feasible, then our modified KLEE forks only the false path. This enables KLEE to perform guided search by excluding cases we do not need. On the other hand, klee_set_forking(1,1) brings KLEE back to the normal mode. Finally, note that this algorithm naturally covers stateless shared variables with stateless producers and multiple shared variables and multiple stateful producers for each shared variable.

4.2 Testing Stateless Transactions

4.2.1 Basic Idea
Testing stateless transactions shares some ideas with testing stateful transactions. The goal here is to find all feasible paths of the stateless transaction. Hence, this time, we explore all paths including possible infeasible paths of the stateless target transaction, then check if each of the paths is feasible or not by considering possible updates on shared variables from the stateful/stateless producers. As an illustration, in Fig. 6, our algorithm first explores all paths by making all the shared variables with the stateful producers symbolic. Depending on the values of the shared variables, some of these paths could be infeasible. Each explored path in each process has an associated path condition such as pc1, pc2, or pc3. For example, there is a shared variable between the stateless transaction and the stateful producer in Fig. 6, and path3 reads the variable twice. The producer writes twice during the first instance, and three times during the second instance. pc3 is a function of r1 and r2. If path3 is feasible, pc3 must be satisfiable when r1 is wi and r2 reads from wj (i ≤ j). Hence, after the path exploration, each process along with a path condition starts to unroll the stateful producer from its constant initial state in a while loop. For example, after one iteration, the algorithm gathers the constant written values w1 and w2. i and j are constrained to 1 ≤ i ≤ j ≤ 2. Then, if pc3 ∧ (wi == r1) ∧ (w2 == r2), is satisfiable, we stop iterating and break out of the loop as path3 is feasible. If not, we keep unrolling, and after the second iteration, i and j are constrained to 1 ≤ i ≤ j ≤ 5. The path condition is checked again for satisfiability. The algorithm keeps unrolling the stateful producer until it finds the path condition satisfiable or hits the upper bound k on the number of iterations. If the shared variable is also produced by the stateless target transaction itself, r1 or r2 either reads from the last value updated by itself or the stateful producers.

4.2.2 Pseudocode
This pseudocode in Code 3 is written for a single shared variable and a single producer for ease of understanding. We support multiple shared variables (either stateless or stateful) and multiple stateless and stateful producers for each shared variable. Line 1 calls the stateless target transaction. It uses the customized library read function whenever it reads from the shared variable so that during the path it reads from the variable’s old value or a sym-
Call the target transaction
Gather the values from the stateless producers
while(1){
klee_set_forking(1,1);
Call the stateful function
old = 0;
for each shared variables with the stateful producers{
for i=0 to (NUM_READ-1) {
  klee_make_symbolic(st, sinfo(t, "t"));
  klee_assume(t>=old); klee_assume(t<WRITE_LEN);
  c[1] = t;
  old = t;
}
}
klee_set_forking(1,0);
if((y_sym[0]==(prod.writes[c[0]]) V any of stateless
producer’s update)) && ... && (y_sym[NUM_READ
-1]==(prod.writes[c[NUM_READ-1]] V any of
stateless producer’s update)) break;
k++;
if(k > K_BOUND) klee_silent_exit(0);
}

Code 3: Testing stateless transactions with stateless/stateful producers

bolic value dedicated for each read. NUM_READ is the number
of reads from the shared variables written by the stateful transac-
tion along the particular path (it is 2 in case of proc03 in Fig. 6).
y_sym[0], ..., y_sym[NUM_READ −1] are the symbolic val-
ues for the reads. After line 1, KLEE explores all paths of the
stateless transaction and multiple processes are created for each
path. Line 2 gathers the values written from the stateless produc-
ers. Then the algorithm enters the while loop and calls the state-
ful producer in line 5. Lines 7-12 simply pick indices c[0], c[1],
..., c[NUM_READ −1] constrained to satisfy 0 ≤ c[0] ≤
c[1] ≤ ··· ≤ c[NUM_READ −1] ≤ WRITE_LEN is the number of total writes on the shared vari-
able from the series of the producer instances so far. Then, in line
16, the algorithm checks if the following condition is satisfiable:
if the path condition is satisfied:

pc ∧ (y_sym[0] == (prod.writes[c[0]]) V any of stateless
producer’s update)) ∧ ... ∧ (y_sym[NUM_READ −1] == (prod.writes[c[NUM_READ-1]] V any of stateless
producer’s update)) where the prod.writes array stores the produced values
from the stateful producer in order. Intuitively, the condition checks
if there are any set of values from the stateful/stateless producers
that satisfy the path condition. The iteration of stateful produc-
ers goes on till the condition is satisfied or it reaches the k bound.
Again, the simple prior work case of stateless variables shared with
stateless producers is naturally included in the above.

5. IMPLEMENTATION

Frama-C [8] is an open-source plugin-based platform written in
ML that provides various static analyses for C code. Each plug-
in performs a specific analysis on the code. One of the plugins
is PDG (Program Dependence Graph). PDG includes DDG (Data
Dependence Graph) and CDG (Control Dependence Graph).
In Fig. 7, the bold edges represent DDG. In this figure, the i th
instance executes path1. In fi, pkt is written. Then, fi reads this
pkt value and writes this value into newpkt. Thus, there is a data
dependence from pkt = inp_gete(temp) to newpkt = pkt in-
struction. The formal definitions of stateful transactions and the
producer-consumer relationship [2] are based on the DDG of the
transactions. We use these definitions with the PDG plugin to iden-
tify the shared variables and interaction patterns between the target
transaction and its interacting transactions.

We adapted the Frama-C source for our work. If we analyze a
transaction with the original Frama-C, the PDG plugin would an-
alyze only the executable branches depending on the initial state,
while we would like to analyze all possible paths independent of
the initial state as these paths may be exercised in some iteration.
In our modified version, the plugin takes all branches at a branch
instruction to cover all paths of the transaction code. We now brie-
fly discuss the specific analyses done using PDG.

Identify stateful transactions: If there is an edge across instances
of a transaction in the DDG then the transaction is stateful [2]. For
example, in Fig. 7, there is an edge between instances i and i + 1
as fi in the (i + 1)th instance reads autobaud value updated from
fi in the ith instance. Therefore, SERIAL0 is stateful.

Producer-Consumer relationship: The dependence table obtained
from the Frama-C PDG provides information about the variables
that a specific variable depends on. This enables tracking where
variables are defined and used. This can then be directly used to
determine which transactions are producers of a variable and which
ones are consumers.

Number of writes on the stateless variables: In STEP2 of the
Stateful Transaction Algorithm we need to know how many times
a variable is written to in its producer transactions. This can be
tracked by Frama-C on updates to these variables.

Code generator: Based on the above information collected in the
Frama-C based analysis, our code generator writes a configura-
tion file for the target transaction and a customized algorithm for
it based on the stateful/stateless algorithm framework as appro-
priate. The algorithm is executed by our modified version of KLEE.

6. EXPERIMENTS

For the experiments, we used 2 published benchmarks from Horn
et al. [11] (Table 2). These benchmarks consist of Linux device
drivers [7] and the corresponding QEMU device emulator code [14]
written in C. QEMU is an open source machine emulator support-
ing a wide range of hardware components including devices such
as storage devices, network cards, etc. Horn et al. extracted several
device models (Table 2) to run standalone, i.e. without dependen-
cies from other parts of QEMU. The corresponding Linux device
drivers are co-simulated with these hardware QEMU models. Like
the Rockbox SERIAL0 example used throughout this paper, the
QEMU and Linux benchmarks can be organized as a set of trans-
actions. Table 3 shows TMP10S transactions and their interaction
patterns. Table 4 describes Ethoc transactions. For the stateful
transactions with stateful producers case we do not handle com-
<table>
<thead>
<tr>
<th>Device</th>
<th>Main Functionality</th>
<th>LOC of the Benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP015 [22]</td>
<td>A temperature sensor</td>
<td>2460</td>
</tr>
<tr>
<td>OpenCores ethernet mac (Ethoc) [15]</td>
<td>An Interrupt (or polling) driven Ethernet MAC with a DMA ring</td>
<td>4130</td>
</tr>
</tbody>
</table>

Table 2: Linux-QEMU device driver benchmarks

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Interaction Patterns</th>
<th>Interacting Transactions</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmp05 Tx</td>
<td>stateful + no prod</td>
<td></td>
</tr>
<tr>
<td>readword</td>
<td>stateful + no prod</td>
<td></td>
</tr>
<tr>
<td>tmp05 set</td>
<td>stateless + stateful prod</td>
<td>tmp05 tx</td>
</tr>
<tr>
<td>show_temp</td>
<td>stateful + stateful prod</td>
<td>tmp05_tx, tmp05_set</td>
</tr>
<tr>
<td>set_temp</td>
<td>stateless + no prod</td>
<td></td>
</tr>
<tr>
<td>lm75 suspend</td>
<td>stateless + stateful/stateless prod</td>
<td>tmp05_tx, lm75_resume</td>
</tr>
<tr>
<td>lm75 resume</td>
<td>stateless + stateless/stateless prod</td>
<td>tmp05_tx, lm75_suspend</td>
</tr>
</tbody>
</table>

Table 3: Interaction patterns of TMP05 transactions

completely, we treated the producers as stateless by allowing the stateful variables to have any initial value, giving us over-approximated results for this case.
Our experiments show the efficacy of our interaction-pattern-specific algorithms for testing concurrent transactions. There is no other tool that provides this direct capability, thus a direct experimental comparison with other approaches is not possible.

6.1 Test Generation Results
The result of test generation for the stateful transactions in Table 3 and Table 4 are provided in Table 5. We covered 9 out of 10 stateful transactions completely within the bound k = 6. In the open_eth_moder transaction, our algorithm was not able to find a test for one path and kept unrolling, eventually terminating at a timeout of 10000 seconds. In ethoc_rx, ethoc_tx, show_temp, and ethoc_interrupt transactions, all the paths could be covered after only one iteration although it is a stateful transaction. This is because all the branch conditions involved with the stateful variables are a disjunction of an expression including the stateful variables and an expression of only stateless variables. Our algorithm could cover these branches by finding satisfying stateless variable values only. Hence does not have to unroll the stateful transactions multiple times. Similarly, we could cover the open_eth_receive transaction after an iteration, but this transaction has a while loop inside whose stop condition is a function of symbolic variables. In this case, KLEE tries to find all the test cases enabling any number of the loop iterations, which is infinite. Therefore, it reached the timeout. We also tested the SERIAL0 transaction.

The result of testing the stateless transactions in Table 3 and Table 4 are in Table 5. Some transactions such as open_eth_reg_read have a relatively small number of test cases as their variables shared with the stateful producers have little effect on the control flow.

6.2 Bug Exposed
The experiments exposed a bug in the published benchmarks. The open_eth_moder_host_write transaction is in charge of updating the MODER register. Fig. 8 is a version of the transaction simplified for ease of exposition. val is the value the user wants to write in MODER. When the user wants to reset the register (meaning RST bit is 1 in val), the transaction checks if the RST bit of MODER is clear at $m_a$. If true, the transaction is allowed to reset the MODER register. In this case, the transaction sets the MODER register to 0000.4000h ($m_b$). The RST bit is cleared again, which means the user can reset it in the next instance. The code is correct up to this point. However, the transaction instance does not end there and executes additional code which sets the RST bit again by writing val on MODER ($m_c$). Hence, each instance will not set MODER to 0000.4000h at the end as the data sheet [15] specifies. Suppose that this instance goes through the thick edges. In the next instance, resetting the MODER register is not allowed, and paths such as the one via $m_a$, $m_b$, $m_c$, $m_a$ are not feasible. This erroneous trace can only be found with more than one iteration. This highlights the practical need for unrolling to expose behaviors of stateful transactions.
6.3 Limitations
This work is driven by the primary motivation of avoiding considering an exponential number of interleavings to consider the impact of interacting transactions on \( \mathcal{F} \). We have shown how their impact can be considered through the algorithms presented in this paper which generate a custom sequential program \( \mathcal{P} \) that captures the interactions of other transactions for specific cases. In particular we showed that if either the producer(s) or the consumer transaction is stateless, this can be done through an appropriate composition of the transactions in generating \( \mathcal{P} \). This composition provides appropriate constraints to our modified version of KLEE. Unfortunately this approach is not applicable to the case where both the producer and consumer are stateful or if the transactions do not have a producer consumer relationship at all. In this case, we will need to revert to considering interleavings with possible optimizations, or overapproximations which may result in false positives. Fortunately, an analysis of benchmarks indicates that these cases are infrequent [2].

7. RELATED WORK

Testing concurrent programs: Traditional approaches for testing concurrent programs use Partial Order Reduction (POR) [10], a widely used technique to explore all interleavings of a concurrent program. SCOOT [3] is a model checking tool for SystemC based on POR. POR techniques cover complete behaviors but are subject to the state-space-explosion problem. To avoid this computational complexity, many works reduce concurrent programs to sequential ones with a bounded number of context switches. KISS [17] transforms concurrent programs into sequential ones by simulating context switches but is limited to only two context switches. Lal and Reps [13] could reduce concurrent programs to sequential ones under any given context switch bound with different program abstractions. CHESS [16] repeatedly executes idempotent tests, and systematically searches interleavings each time. CHESS bounds the number of context switches as well.

In contrast, by focusing on specific interaction patterns that occur often in practice, we avoid the complexity of considering transaction interleavings. This can potentially provide full coverage without explicitly considering the interleavings.

Loop unrolling: Loop unrolling in software validation has been done in various contexts. It is used in test generation tools like DART [9] as well as Bounded Model Checking tools such as CBMC [6]. These tools are able to discover deep bugs after many loop iterations. The main contribution of our work is to show how this can be done in combination with the composition with the interacting transactions. This composition with unrolling is the heart of the code generator which constructs a sequential program \( \mathcal{P} \) that is test equivalent to the target firmware \( \mathcal{F} \).

8. CONCLUSIONS

Validating firmware is challenging as it needs to consider the concurrency between firmware and its interacting hardware/firmware components. A previously published uniform transaction-based modeling framework for firmware/hardware has the potential for non-exhaustive pattern-based testing for concurrent programs. The main idea there is to use specific interaction patterns between transactions to avoid exploring the possibly exponential interleaving of transactions during test generation. We extend this work to consider a much larger class of patterns to include transactions that may be stateful. This extended class covers the most common interactions patterns seen in practice. Further, we provide a fully automated solution that builds on our modified versions of publicly available static analysis and concolic testing tools. Given a TLM and a target firmware transaction \( \mathcal{F} \), we generate a sequential program \( \mathcal{P} \) that composes \( \mathcal{F} \) with its interacting transactions. This enables a slightly modified version of the KLEE concolic testing tool to be used for generating tests for \( \mathcal{P} \) which can be directly used for \( \mathcal{F} \). For each feasible path, the algorithm generates the shortest possible test. The applicability of this testing methodology is demonstrated using published benchmarks from Linux drivers with the QEMU emulator code for the interacting hardware components.

9. REFERENCES