AUTOMATED FIRMWARE VERIFICATION USING
FIRMWARE-HARDWARE INTERACTION PATTERNS

SUNHA AHN

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
ELECTRICAL ENGINEERING
ADVISER: PROFESSOR SHARAD MALIK

MAY 2016
Abstract

Firmware refers to low-level software that is tied to a specific hardware platform. For instance, low-level drivers that physically interface with the peripherals are an example of firmware. An emerging trend in system design is to implement complex system management functions in firmware rather than hardware. For example, firmware has grown to include software that manages critical hardware platform functions such as power management. As the scale and the importance of firmware is increasing, its validation becomes a critical part of system validation.

Firmware validation relies on having good models of the interacting hardware components because firmware needs to be shipped with the hardware and shares many of the same critical design concerns as the hardware. This is generally addressed through co-simulating C/C++ based firmware code and HDL (including SystemC) hardware models, which are usually not available until the late design stages. However, co-simulation tends to be slow, and is further exacerbated by the large number of possible interleavings between the concurrent firmware and hardware threads. Typically, in co-simulation, the thread scheduler, such as the SystemC scheduler, only explores a small number of possible firmware-hardware interleavings and thus may miss critical bugs.

A firmware function is mostly reactive: it continuously provides a service, with a clear start and end, in response to inputs from its interacting software or hardware layer (i.e., the environment). Thus, a firmware function is often inherently associated with an infinite loop structure. This often makes it impossible to guarantee the completeness of the verification results.

To this end, I address two key problems in this thesis. First, I describe how to co-design firmware with the system components at the service function level, also referred to as the transaction level. Second, I discuss how to validate firmware interactions with their connected hardware modules while pruning the verification search space and ensuring com-
plete verification. To solve these problems, this thesis first introduces a specific Service-
Function Transaction-Level Model (TLM) for modeling firmware and interacting hard-
ware components. I capture the particular structure of the proposed TLMs through cross-
transaction interaction patterns, such as statelessness, i.e., when variable values are not
retained between transaction executions, or producer-consumer relationships. Using the
TLM, this thesis presents a scalable firmware validation approach that is based on au-
tomatically generating a test set with the goal of complete path coverage for firmware.
Instead of explicitly exploring all the interleavings of the concurrent transactions, this
thesis exploits the interaction patterns to automatically generate a sequential program,
which is test-equivalent to the target firmware transaction and can be used with a standard
single-threaded concolic test generator. The tests generated can (i) be directly used for the
firmware transaction and (ii) account for the multi-threaded interactions. However, due to
the infinite loop structure of the TLMs, the sufficient bound that guarantees the complete-
ness of verification needs to be determined. This thesis provides inexpensive static bound
analysis techniques using commonly occurring firmware code patterns. Further, the bound
analysis is combined with the previous interaction pattern-based sequentialization method
to cover all common interaction patterns with the complete coverage. The resulting gen-
eral sequentialization method enables the direct application of standard software bounded
model checkers such as CBMC on this sequentialized program. The practical aspects of
our work are evaluated using real firmware benchmarks, e.g., the Rockbox mp3 player
firmware code and Linux device driver code with its interacting QEMU hardware emulator
code.

In summary, this thesis presents novel verification methodologies with new techniques for
reducing the search space of concurrent and unbounded firmware systems and their inter-
acting firmware/hardware transactions using common interaction patterns derived from the
novel service function-based TLM proposed in this thesis. In doing so, it enables both
scalability and completeness of verification whereas many other existing methodologies
typically trade completeness for scalability, or vice versa. As the complexity and size of firmware continue to grow, the techniques developed in this thesis will provide the firmware designers a modeling framework for firmware systems that is more natural and close to the human thought process, and it will help firmware developers to have a complete but practical explanation of the behaviors of complicated concurrent and unbounded firmware systems.
Acknowledgments

I am about to be PhinisheD! During my time at Princeton, I received so much support from numerous people. I believe that I would not have been able to get my Ph.D. without them, and I would like to give special thanks to them.

First, I am truly grateful to my adviser Sharad Malik. He was always there for me for any kind of advice, and he guided me throughout the years with sharp ideas and technical/professional feedback. He also helped me to improve in all aspects of my technical skills. Now I am out of school and already realize how valuable these skills are for my career. I deeply thank him for this. Moreover, he has been my role model as a professional. He is very passionate at work, and he leads his students with respect and sincere care. His influence on me will help me throughout my career.

I would also like to thank the rest of my thesis and general exam committees: Aarti Gupta, Niraj Jha, David Wentzlaff, Prateek Mittal, and Margaret Martonosi. Their feedback and suggestions made this thesis possible. In particular, Aarti gave me lots of helpful feedback on my thesis, and she put a lot of effort into our in-depth discussions during the weekly meetings. She also taught me how to write papers well. These all made my thesis much better. She also gave me a lot of advice when I was looking for a job. I am very lucky to have her as my second adviser. I also thank Margaret Martonosi, not only for her helpful feedback in the early stages of my work, but also for inviting me to many group dinners and parties. Niraj, David, and Prateek gave lots of helpful comments on my thesis and the FPO slides.

Moreover, I would like to thank the EE administrative staff members. Lori, Stacey, Colleen, and Heather worked hard to make sure things ran smoothly for me. They also took care of me during my hard times and my good times.

My research group has been a source of support and academic motivation: Daniel, Carven, Arnab, DJ, Georg, Yavuz, Charlie, Shuyuan, Pramod, Nestan, Burcin, Pareesa, Sayak,
Zhixing, Yakir, Jade, and Bo Yuan. Their hard work and passion kept me motivated to work. They were all open to technical advice as colleagues or random chats as friends. C305 was a good place to go every morning because of you.

I also thank other members of the EE department: especially Ozlem, Wenhao, and other members of Margaret Martonosi’s research group.

I thank Shelly Lim, who I shared lots of good memories with. I am glad both of us survived here! Thanks to Ena and Dongbeom for everything you have done to me from the very first day I arrived at Princeton. Haywon unni, I have gotten lots of advice from you. I thank Haemin and Hyuncheol (and Julia of course for being cute), too. Haemin unni, our weekly lunch was one of my favorite times from Princeton. You were a very good friend to me, and I could share my very personal stories with you. I also thank both Taewooks, Taehee, Junhwan, and Yeohee who made my Equad life so much fun. Thanks for all the coffee times that made me laugh.

Dan, I cannot thank you more for all you have done for me. This thesis would have been impossible without you. I went through so many difficult times during my Ph.D., and there were so many times I doubted my ability to finish this journey. However, you always gave me courage to believe in myself and kept me focused. Because of your endless support and faith, I feel strong when I am with you. Also, thank you for all the happy memories we share, all the laughs, and all your efforts to learn Korean (and somehow being surprisingly creative). Lastly, my system wins.

Most importantly, my deepest thanks goes to my family.

I would like to start with my mom and dad, whom I love more than anyone. Mom, I have a little secret that I never shared with anyone: my biggest source of motivation for all my hard work has always been you. I want to be successful not only for myself, but more for you, to make you happy and proud. As I was growing up, you were the best mother one could have: you always cared about me first, taught me numerous life lessons, and always
respected my decisions. Thanks for teaching me never to give up and to have the courage to accomplish every step of my life.

Dad, thanks for being a strong and respectable man as I was growing up. There was never a lack of anything, and that is all thanks to you. I now understand how difficult it is to be such a successful man and to support a big family, and I truly respect you for that. Also, thanks for always being a nothing but a sweet dad with endless love and attention.

I would like to thank to my sister, Hyehyun. Although we are best friends, our relationship has been a bit unfair. You always took care of your little sister, and I am already pretty spoiled by that, but I will try to be your supporter from now on. Also, you encouraged me when I struggled with graduate school in my early years, and I still remember all the advice you gave me. That actually drove me through my Ph.D. I still think about those words whenever I feel life is tough. I would like to be like you, someone who manages a family and work successfully, while still having a big heart for people around you.

My little brother Chongwoong, you inspire me as you dive deeply into whatever you do and go through any difficult situation with your mental toughness. I respect your fearless attitude towards life. That makes you a very special person, and I would like to be someone like you too! Thanks for being a source of inspiration.

Kinam hyungboo, I appreciate that you have become such a great son to my parents and my big brother I never had. I am happy that I have another successful and respectable family member on my side. I also thank you for treating me well over many years, and for bringing the cutest person into the world.

My aunt Young also deserves my gratitude. She is the coolest and sweetest aunt one can ever have. She showed me many beautiful aspects of the world. I am very lucky to have such an elegant and tasteful woman as my aunt.
Seungwoo, my precious nephew, you may not know what you did but you did a very important job, providing emotional support, during the last 2 years of my Ph.D. You made your aunt’s world so much happier and brighter. Watching your videos made me smile first in the morning, all throughout the day, and right before I went to bed every single day ever since you were born.

Lastly, my little Mungmung brings me enormous happiness.
To my parents.
Contents

Abstract ........................................................................................................ iii
Acknowledgments ..................................................................................... vi
List of Tables ........................................................................................... xvi
List of Figures ............................................................................................ xviii

1 Introduction ............................................................................................ 1

1.1 Motivation ........................................................................................... 1

1.2 Research challenges and goals ............................................................. 5

1.2.1 Traditional firmware development and verification ................. 5

1.2.2 Emerging trend: Integrated development/verification flow using
executable specification models ................................................................. 10

1.2.3 Challenges in verifying firmware-hardware TLM ..................... 13

1.2.4 Using a well-structured modeling framework to overcome the chal-

lenges ........................................................................................................ 15

1.3 Research overview .............................................................................. 16

1.3.1 Service function-based TLM ....................................................... 16

1.3.2 Automated firmware testing using firmware-hardware interaction

patterns ........................................................................................................ 18

1.3.3 Completeness bounds and sequentialization for model checking of
interacting firmware and hardware .......................................................... 20
3.3.2 Analyzing data dependencies between transactions ........................................ 53
3.3.3 Prevalence of the interaction patterns in practice ............................................. 56
3.4 Using transaction interaction patterns in firmware testing .................................... 59
  3.4.1 Automatic test generation using KLEE ................................................................. 59
  3.4.2 Test case generation for stateless producer-consumer transactions ................. 61
  3.4.3 Limitations ........................................................................................................... 68
  3.4.4 Experimental results ............................................................................................ 69
3.5 Related work ............................................................................................................ 70
  3.5.1 Transaction-level models ...................................................................................... 70
  3.5.2 Concolic testing for concurrent programs ............................................................ 71
3.6 Chapter summary ...................................................................................................... 72

4 Automated Firmware Testing using Firmware-Hardware Interaction Patterns ................. 74
  4.1 Introduction ............................................................................................................. 74
  4.2 A running example of the service-function based TLMs ......................................... 76
    4.2.1 Transaction interaction patterns occur in the running example ........................ 78
    4.2.2 Transaction interaction patterns and test generation: Previous work and its limitations .............................................................. 78
  4.3 TLM based concolic test generation for stateful transactions: Overview ............... 80
    4.3.1 Challenges with stateful transactions: A motivating example .......................... 80
    4.3.2 Overview of the methodology .......................................................................... 83
  4.4 Stateful and stateless test generation algorithms ................................................... 84
    4.4.1 Testing stateful transactions .............................................................................. 84
    4.4.2 Testing stateless transactions .......................................................................... 93
    4.4.3 Summary ........................................................................................................... 95
  4.5 Implementation ....................................................................................................... 96
  4.6 Experiments .......................................................................................................... 98
4.6.1 Test generation results ................................................. 99
4.6.2 Bug exposed .......................................................... 101
4.6.3 Limitations ............................................................. 102
4.7 Chapter summary ......................................................... 102

5 Completeness Bounds and Sequentialization for Model Checking of Interact-
    ing Firmware and Hardware ........................................... 104
  5.1 Introduction ............................................................ 104
  5.2 Background ............................................................ 106
    5.2.1 Service function-based TLMs ................................... 107
    5.2.2 Program dependence .............................................. 108
    5.2.3 Bounded model checking (BMC) ................................ 109
  5.3 Completeness bounds: Key ideas ................................... 110
    5.3.1 Dataflow chains in TLMs ....................................... 112
  5.4 Completeness bound: Algorithm .................................... 114
    5.4.1 Sufficient BMC bound for bounded chains ................... 114
    5.4.2 Sufficient BMC bound for unbounded chains ................. 119
    5.4.3 Implementation .................................................. 122
  5.5 Unified sequential firmware-hardware model ...................... 123
    5.5.1 Producer-consumer relationship ................................ 123
    5.5.2 Producer analysis ................................................. 123
    5.5.3 Sequentialization for model checking ......................... 127
  5.6 Experiments .............................................................. 129
    5.6.1 Sufficient BMC bound calculation ............................. 129
    5.6.2 Automated model checking ...................................... 131
    5.6.3 Discussion ........................................................ 132
    5.6.4 Comparison with the partial order reduction method ........ 133
  5.7 Related work .......................................................... 136
5.7.1 Loop bound analysis ........................................ 136
5.7.2 Verifying concurrent programs ........................................ 137
5.8 Chapter summary ........................................ 137

6 Conclusion ........................................ 139

A List of Publications ........................................ 143

Bibliography ........................................ 144
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Rockbox transactions and their functionality (FW: Firmware, HW: Hardware)</td>
<td>56</td>
</tr>
<tr>
<td>3.2</td>
<td>Relationship of the interacting Rockbox transactions (p-c means producer-consumer)</td>
<td>57</td>
</tr>
<tr>
<td>3.3</td>
<td>Linux-QEMU interacting transaction pairs (p-c: producer-consumer, SL: stateless, SF: stateful)</td>
<td>57</td>
</tr>
<tr>
<td>3.4</td>
<td>Test cases of Rockbox transactions</td>
<td>70</td>
</tr>
<tr>
<td>4.1</td>
<td>Path conditions for the six paths produced in STEPI of Fig. 4.4</td>
<td>86</td>
</tr>
<tr>
<td>4.2</td>
<td>Linux-QEMU device driver benchmarks</td>
<td>98</td>
</tr>
<tr>
<td>4.3</td>
<td>Interaction patterns of TMP105 transactions</td>
<td>99</td>
</tr>
<tr>
<td>4.4</td>
<td>Interaction patterns of Ethoc transactions</td>
<td>99</td>
</tr>
<tr>
<td>4.5</td>
<td>Testing results of the stateful/stateless transactions</td>
<td>100</td>
</tr>
<tr>
<td>5.1</td>
<td>Fitness value calculation table</td>
<td>121</td>
</tr>
<tr>
<td>5.2</td>
<td>Fitness table for non-wrap-around monotonic variables</td>
<td>126</td>
</tr>
<tr>
<td>5.3</td>
<td>The sequentialization for the test target $\mathcal{F}$ and its concurrent producers. * stands for a non-deterministic value ($\geq 0$).</td>
<td>127</td>
</tr>
<tr>
<td>5.4</td>
<td>Linux-QEMU device driver benchmarks</td>
<td>129</td>
</tr>
<tr>
<td>5.5</td>
<td>Bound calculation results for the given properties</td>
<td>130</td>
</tr>
<tr>
<td>5.6</td>
<td>Model checking results of the Linux/QEMU transactions</td>
<td>131</td>
</tr>
</tbody>
</table>
5.7 Model checking results of the Linux/QEMU transactions using Verisoft with unbounded outer loop executions .......................... 134

5.8 Model checking results of the Linux/QEMU transactions using Verisoft with bounded outer loop executions .......................... 136
List of Figures

1.1 Computer system components ........................................... 2
1.2 An example firmware system ........................................... 2
1.3 Typical design cycle of firmware and its interacting device [99, 102] .... 6
1.4 Integrated design flow of firmware and its interacting device ............ 10

3.1 Rockbox code structure (shown in the gray color) ....................... 48
3.2 Example Producer $T_k$ and Consumer $T_n$ (a dashed arrow represents a trans-
action instance and a solid arrow is an edge of the dependence graph) ..... 54
3.3 Stateful transactions $T$ and $T_r$ ...................................... 54
3.4 The execution of Code 3.1 on KLEE and the resulting test cases .......... 60

4.1 Rockbox SERIAL0 firmware transaction and its producer firmware transaction

serial bitrate ............................................................... 77
4.2 HLSM for the simplified version ....................................... 81
4.3 Overview of the automated testing methodology ........................ 83
4.4 Testing stateful transactions with stateless producers: Rockbox SERIAL0

example from Fig. 4.1] .................................................... 87
4.5 path1 of Table 4.1. This is one of the paths in the Rockbox SERIAL0

example from Fig. 4.1] .................................................... 88
4.6 Illustration of the algorithm for testing stateless transactions with state-
less/stateful producers .................................................. 93
Chapter 1

Introduction

1.1 Motivation

Firmware is low-level device-specific software which can directly access hardware and is often shipped with the hardware platform. This hardware-specific nature distinguishes it from higher-level device-independent software such as the operating system (OS) or application code. Firmware and hardware share information with each other through memory-mapped input-output and special-purpose registers. Further, hardware may communicate with firmware through interrupts. Example firmware systems are embedded software in embedded systems for consumer electronics (mobile, cameras, or mp3 players), home electronics, industrial, automotive, medical, commercial and military applications, etc., the basic input/output system (BIOS) in computers, and device drivers.

Firmware is a critical part of computer systems. Firmware is everywhere; it can be in the form of software semi-permanently embedded in chips, drivers for specific physical components, or low-level OS code that directly interacts with hardware. In each case, the firmware is placed between the hardware and higher-level software (OSs or applications) as in Fig. [1.1], thereby enabling them to communicate, via firmware, with each other. In other
words, software uses firmware to drive hardware or get the status of hardware. Without firmware, the device does not function, and software is not able to access the hardware device.

Fig. 1.2 shows how a temperature sensor works in an embedded system. The sensor system contains a microcontroller, which reads the physical temperature value and updates the shared register dedicated to the sensor using the sensor driver. The sensor can in turn trigger an interrupt which is detected by the sensor driver. The sensor driver sends the requested sensor data to the OS, then the OS responds to this data. The driver is designed to communicate with both the high-level OS and the sensor.

Firmware has distinguishing features, which make it special compared to application software. For example, a firmware function typically provides a reactive service. In other words, a firmware function repeatedly performs its specific service in response to events from the software or hardware layer by interacting with its physical environment via sensors or other communication interfaces. This is shown in the sensor example in Fig. 1.2. The microcontroller continuously requests the sensor driver to process the new sensor (physical)
data. The driver, in response to this input data, performs its service by updating the shared memory (shared with the OS) so that the OS can take a consequent action if necessary.

While firmware relates to both software and hardware, firmware development engineering has shown much less progress compared to software or hardware development. First, the variety of firmware systems prevents the development of a standard platform, whereas modern computer systems use standardized platforms for many components. Further, while there are a wide variety of high-level languages for general and special-purpose software engineering, firmware development techniques are relatively primitive. For instance, there is no universally accepted machine-independent higher-level firmware language. Scalability is a key requirement in system development. The development of typical hardware or software components involves a number of abstraction layers, supported common platforms, and virtualization techniques for scaling the target systems to address the scalability issues while maintaining the reliability of the systems. On the other hand, firmware’s relatively ad-hoc development style can make it less reliable. Second, the non-functional requirements such as real-time constraints make it challenging to develop generic and automated firmware development solutions as they are device-specific. Third, firmware developers need to have extensive knowledge of the device. Developing the control systems of devices requires understanding the functions of the target device. Insufficient knowledge across different layers often results in poor quality of the firmware systems.

Despite the inherent challenges in developing automated solutions for firmware engineering, the importance of firmware has been rapidly increasing as seen in its growing size and complexity. Firmware has been growing since the first commercial microcode implementation on the IBM 360 series in the early 1960s \[28\] (microcode is viewed as a form of firmware). Firmware now contributes to a large part of the recent computer systems. For example, a desktop PC may have tens of devices, including input devices, display, storage, USB controllers, and so forth, where each of these devices requires a driver. The increas-
ing number of device drivers composes a big part of the OS kernel code. Today, 70% of
the Linux code base is the device drivers [61]. Furthermore, many everyday electronic
products, such as cellphones, cars, or medical systems, contain embedded software. The
number of embedded systems contained in any new product today lies in the range of one
to hundreds: the market for embedded systems is already 100 times bigger than the desktop
market, and it is expected to grow exponentially in the future [32].

The recent trend of the increased migration of hardware functionalities into firmware has
led to even larger firmware [67]. Firmware has grown to include software that manages
critical hardware platform control functions, such as power and even security manage-
ment. These functions were previously implemented in dedicated hardware controllers, but
their increasing complexity, coupled with the increasing availability of on-chip processing
cores, has led to their migration to firmware [67]. By moving control from hardware to
firmware, programming these functionalities becomes easier and hardware becomes sim-
pler. Firmware is easier to update before shipping than hardware as well. This growing
firmware needs to be shipped with the hardware and thus shares many of the same critical
design concerns as the hardware. The two concerns that we address in this thesis are: co-
design with other system components, and validation of the firmware interactions with the
hardware.

Firmware validation is a critical part of system validation because of its increasing size and
importance. Ensuring the correctness of the software on top of the complex underlying
hardware is a challenging problem. While the speed of modern computer systems has al-
ready reached a “good enough” level for mainstream users, reliability has become a rapidly
growing concern for both computer users and product developers. 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 OS [39, 101]. This suggests that en-
Hanced firmware verification/development techniques can significantly reduce OS crashes. Growing firmware size also contributes to its increasing complexity and bugs. Another example showing the importance of the firmware verification is the boot firmware. Boot firmware controls the series of boot steps in which the hardware is initialized and the OS is loaded. Since this typically runs in the privileged mode, it could potentially corrupt the device and the OS.

Further, firmware validation in the early design stages can reduce a significant amount of development cost.Fixing firmware-hardware bugs is much more expensive than fixing software bugs. Firmware bugs due to the (environmental) hardware input can be found after the full hardware and software development, prior to or post fabrication. Hence, fixing these bugs may require physically replacing the integrated circuits or memory whereas erroneous software can be easily replaced. Doing this late in the design cycle or even after the chip fabrication is costly.

1.2 Research challenges and goals

Although firmware is in the form of software code, typical software verification techniques cannot be directly applied. Firmware is not a stand-alone product, but an element of a system, which includes its interacting hardware devices and/or software applications on top of it. Firmware development must consider various constraints related to the hardware platform. This makes firmware development and verification far different from device-independent software.

1.2.1 Traditional firmware development and verification

A typical firmware development process, including its validation with the interacting hardware platform, is depicted in Fig. 1.3 [99, 102].
However, this typical design cycle of firmware together with the hardware device shown in Fig. 1.3 has some issues.

**Lack of formal specification models**

First, the design flows for firmware and hardware may start together at the initial system design level. A system design typically starts with a specification and a constraints list. However, the system starts to be designed separately in different teams very soon. This causes the system specification to be partitioned. This decision is based on the available libraries, re-usable components or templates, the application, the constraints, etc. Next, on both the hardware and the software side, detailed functional specifications that describe the design requirements, such as intended capabilities, size, interaction with users, etc., are written. From the specification, through stepwise refinement, the final product is de-
Specifications are usually written in human languages, not in formal languages. Hence, throughout the development process, it is hard to synthesize the functionalities that are matched with the informal specifications. This gap between the specification and the implementation is filled in through human intervention, which is expensive.

**Lack of language support for firmware development techniques**

Current firmware development does not have much language-support for various firmware development needs. There are few languages for writing firmware. Esterel [6] is a deterministic concurrent programming language to model the reactive system by maintaining permanent interaction with their environment. Esterel, however, assumes ideal synchronous systems, i.e., the computations are infinitely fast. This does not faithfully model the real-world firmware or hardware systems. ESP [67] is a language for programming event-driven state-machines for programmable devices. ESP provides debugging ability using the model checker SPIN [54], but it requires modeling of safety properties in the specific specification language of SPIN.

Further, insufficient debugging support for firmware is also a problem [67]. For example, observing the memory locations of the host device may be needed to diagnose the bugs, but observability on the device can be very limited.

**Difficulties in debugging**

Debugging the integrated models at the late design stage is exceedingly difficult as the firmware failure can be a combination of device problems, and thus hard to locate and reproduce as it is hard to accurately record the program or device states. For software, by simply inserting some primitives, it is possible to record the trace information and the states of the variables of the program. However, there are usually no firmware primitives to control the output devices. External aids are necessary to observe the behavior of...
the firmware code or the hardware device at an event occurrence on the other side of the firmware-hardware boundary.

**Difficulties of integrating firmware and its interacting device**

Although early integration of the firmware and hardware can dramatically reduce the development cost, their co-design and simulation pose some challenges. Firmware implementations are often written in high-level languages, such as C/C++ together with lower-level assembly code. On the other hand, hardware models are written in hardware description languages, such as VHDL, Verilog, or SystemC. Co-simulating two completely different models is challenging. It usually requires nontrivial overhead in communication (for example, sending data via a network) or in simulation speed. Specifically, register-transfer level (RTL) simulation for the hardware tends to be slow. In conventional RTL simulation, an application program which takes 1 second to run on a real chip with 100 MHz frequency can run more than 10 days to simulate with an RTL simulator [80] (because the RTL simulator is run at 10-100 Hz). A number of research efforts have tried to tackle this problem. Software/firmware can be executed using a virtual model of the hardware, i.e., a higher level abstraction of the implemented hardware model [102]. Some research uses compiled simulation to speed up the co-simulation [107]. However, with these methods, timing and performance analysis are usually restricted. Several works use such higher-level verification methodologies [80, 38]. These works use C/C++/SystemC-level models for the software/firmware, and use hardware emulation models written at the same level [8]. This can speed up simulation speed to 1 MHz, but does not allow cycle accuracy of the verification result. More importantly, the simulation may still be limited by the large number of possible interleavings between the concurrent firmware and hardware threads, of which generally only one interleaving is simulated.
Recently, hardware-software co-design tools have been introduced in the hardware community to overcome the integration difficulties. Cadence’s Virtual Component Co-design tool VCC [17] or Synopsys’ CoCentric tool are examples. The software community has also developed UML [74], a set of notations for developing specifications of object-oriented systems. Instead of direct use of programming languages to code software, UML provides a more intuitive and expressive graphical notation. This is a higher level of abstraction than native programming languages. Using these tools, developers specify the complete functionality of the firmware, and the generators can automatically create the code implementing these functionalities. However, the hardware side and the software side still remain separate with these techniques.

**Lack of communication and understanding**

The firmware developers and the hardware developers may not understand each other’s implementation well due to the separate development. Integrating subsystems developed by different teams or even companies without some standards may lead to building heavy but ineffective testing scenarios.

**Time-to-market and cost**

Firmware implementation usually needs to be designed in conjunction with the hardware design. Verifying the integrated hardware and firmware model before fabrication is a mandatory step. This enables full hardware and firmware development before fabrication. However, the hardware model is usually only available late in the design stage. Therefore, integration of the two models happens late, and the firmware model can be verified late in the design stage. This results in a long time-to-market. Moreover, there is a risk of uncovered bugs in each part of the design cycle until very late. Fixing firmware-hardware bugs
may require hardware updates. This is costly, especially after the hardware is fabricated. Hence, fixing bugs earlier is much preferred.

### 1.2.2 Emerging trend: Integrated development/verification flow using executable specification models

Firmware and hardware development do not take any advantage of the shared design concerns during the design stage in the majority of cases. All the issues above call for better design flow and a proper modeling framework which is common to both hardware and software designers. At a minimum, a modeling framework that correctly captures the firmware-hardware interface interactions is needed. Firmware needs to be validated in the context of this interface model.
To overcome the lack of a common framework in practice, a unified formal specification modeling methodology for both firmware and hardware has been called for, especially within the hardware-software (firmware) co-design community. In Fig. 1.4, a possible design flow scenario using the specification-level unified models illustrates why a high-level unified formal model is beneficial compared to the separated design flow shown in Fig. 1.3. A specification describes what is to be accomplished in the system, e.g., a clear description of functionalities, interfaces with other systems, the size of the systems, etc. Specification-level modeling methodologies can integrate hardware and software designs from the very beginning of the design cycle. Functionalities are modeled formally, and important design decisions can be made early with integrated specification-level models. Then, design requirements can be defined and refined together in each synchronization point.

Compared to Fig. 1.3, the integrated flow has great benefits with the executable specification: (1) implementing lower-level models from high-level functional formal models is much easier as the specification model acts as a bridge between the specification written in human languages and a formal implementation. (2) If the specification model works correctly, it can be used as a golden reference model for the development/testing process. Verifying implementations is easier as we have the golden model to check against. It also helps to reveal errors earlier in the design cycle. (3) It helps communication between the designers of the host devices and the firmware (software-level) programmers. (4) The integrated design flow enables concurrent development and verification of firmware and hardware — their designs and verification tasks can be synchronized and performed together in the middle of the design stage, i.e., hierarchical design and verification becomes possible. Time-to-market is significantly reduced this way. Research shows that this design flow leads to savings of up to six months in many cases out of a typical 18-24 month design cycle [102]. By designing and verifying at the higher abstraction level, re-verifying everything at a low level is not necessary. At a lower level, only the low-level architec-
tural issues are in focus to verify the consistency between the high-level models and the low-level models. This way, there are far fewer bugs found late in the development cycle, which can be too costly to fix.

However, implementing an integrated software-hardware development flow requires significant work on design tools. These tools act as the bridges that integrate the software implementation and the hardware modeling environment for the synchronization or special co-simulation techniques. Eventually, spending more design time for a careful analysis and exploration of design options can expedite technological progress greatly.

There have been approaches for the early design models for both firmware and hardware in a common modeling language. As an intermediate solution for software (firmware) and hardware languages, pure C and C++ have been extended to cover both sides well. SystemC [8] and SpecC [38] are well known and well established. SpecC is an extension of the ANSI C programming language, and SystemC is a class of libraries of C++. Both languages provide useful data types and concurrent programming methodologies for software/hardware. Especially, SystemC has gained popularity for integrating firmware and hardware because of its ability to model hardware at higher abstraction levels [93, 100].

Using SystemC, the timing aspect of the design can be simulated using the event-based simulation library. Also, there are tools from Forte [35] or Cadence [13] which enable hardware synthesis from SystemC models. SystemC provides the modeling ability at the transaction level. Transaction-level models (TLMs) [14, 40, 71] are functional models that capture the interactions between components without necessarily providing all the low-level details. Co-simulation of the firmware and hardware TLMs is then the workhorse for validating the firmware and hardware. SystemC manages the concurrency between the firmware and hardware threads through its scheduler. However, the main challenge associated with this methodology is that the SystemC scheduler typically does not/cannot consider the prohibitively large number of possible interleavings between the concurrent
hardware and firmware. 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. Thus, this simulation is incomplete with an unknown coverage of interleavings.

1.2.3 Challenges in verifying firmware-hardware TLM

As suggested by the emerging trend, if a high-level specification model is used early in the design cycle and can then be coded in a high-level software language, well-developed software verification techniques can be applied. In fact, verification of firmware programs is based on software verification techniques to a large degree [28]. However, there are some benefits in firmware verification compared to the software field. Firmware usually is of a manageable size, and each of its transactions is relatively simple and defined clearly. In this way, this nature of firmware makes its verification more tractable than software verification. Furthermore, many software verification methods have been criticized because they are only practical for small programs. However, simple firmware programs fit into this scale [83], and the software verification techniques together with such characteristics of firmware programs can result in powerful and practical verification methods. On top of using well-developed techniques for software, firmware verification can enjoy the benefits of having smaller and simpler programs and a simpler relationship with its (often concurrent) environmental hardware or software threads.

This thesis considers a design methodology that uses high-level unified TLMs for both hardware and firmware to take advantage of software verification techniques while overcoming obstacles of typical firmware verification methods. However, there still exist challenges inherent to the nature of firmware. This section addresses two challenges: concurrency and unboundedness of firmware.
Concurrency

One of the biggest challenges in firmware validation is concurrency [67]. A firmware function processes the input events from its interacting software or hardware layer at a speed imposed by the environment. Hence, firmware functions are inherently concurrent with the hardware components. Not only are firmware and hardware concurrent, but firmware functions are often concurrent with each other. Concurrent programs are inherently hard to write correctly, and thus they often suffer from various problems, such as race conditions, large search space, and so on.

In practice, to analyze and verify this concurrency, the typical approach is to reduce concurrent executions to sequential ones with a feasible order of interleavings of concurrent threads. This sequentialization technique is widely used for analyzing firmware systems, e.g., in Esterel [6], a reactive language used for real-time applications. Similarly, SystemC [8], a widely used language for integrating firmware and hardware, uses a scheduler to execute the system sequentially. However, the total number of possible interleavings to explore is simply too many. It grows exponentially with the size of the model, and even the state space of small-sized programs can easily blow up. Hence, these tools often explore only a small fraction of possible interleavings. 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 co-simulation and thus missing possible bugs. To reduce the space of interleavings of the firmware and hardware state updates during verification, traditional methods, such as partial order reduction [47] or the context-switch bound [68, 86], can be applied. However, they can be still computationally expensive, over-approximate, or incomplete in avoiding costly computations.
Unboundedness

Firmware often consists of a set of asynchronous tasks, each of which may be repeated an infinite number of times. Thus, in this setting, firmware code is inherently associated with an infinite loop structure. Its state space is explored iteratively, and some bugs can only be found by executing through a large number of instances/unrollings. Many software model checkers are able to discover counterexamples of deep bugs only after exploring a large number of iterations/unrollings of such loop structures. Bounded model checking (BMC) \cite{7} searches for property violations within a given bound $n$ of unrollings. However, without a threshold for $n$ that guarantees completeness of verification, this method cannot prove correctness. A key challenge in using BMC lies in computing such a completeness threshold. Recently, SAT-based methods \cite{76,77}, such as interpolation-based methods, have been applied to verify unbounded models. However, to guarantee completeness, the verification engines need to store the set of reachable states or their over-approximations, which can limit the scalability of these techniques on practical programs.

1.2.4 Using a well-structured modeling framework to overcome the challenges

To this end, this thesis presents a TLM model which is a unified modeling framework for both firmware and hardware. Using this well-structured modeling framework, this thesis introduces systematic and scalable testing approaches that can result in much more extensive testing than the traditional methods for real-sized firmware code. Here, this thesis presents a new compositional form of sequentialization that exploits specific structural information between the firmware and hardware threads. Further, to provide complete verification, this thesis introduces a new technique to determine a sufficient BMC bound to prove the property or find a violation. It is a static analysis approach that exploits characteristics of
common code patterns found in firmware to provide inexpensive termination checks for unbounded models. This is described in detail in the following section.

1.3 Research overview

This section provides a high-level overview of this thesis.

1.3.1 Service function-based TLM

Chapter 3 first presents the novel service function-based transaction-level model for the co-design of firmware and its interacting hardware components [2]. This modeling framework captures the reactive nature of these interacting components. Firmware often consists of a set of asynchronous finite-duration tasks, each of which may be repeated an infinite number of times. More specifically, 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 (Fig. 1.1). A service function provides a service in response to a specific trigger, much like an interrupt-service routine responding to an interrupt. Thus, in this framework, both firmware and hardware functions are modeled as a set of services. This unit of work (service function) is referred to as a transaction in the service function-based TLM. Each transaction has a clear start and end (i.e., it is finite) and is repeatedly executed in a loop. Each execution of a transaction is referred to as a transaction instance. A system can be modeled as a set of concurrent firmware and hardware transactions. Transactions can be specified using high-level imperative languages such as C or C++. Hence, an execution of a TLM model, comprising firmware and hardware transactions, is an asynchronous interleaving of state updates (program statements), which come from the infinite streams of the firmware/hardware transactions.
The TLM approach can be highly beneficial for verification as a transaction can also be a unit of verification. This helps to partition the testing size, i.e., only related transactions need to be tested together.

Moreover, in structuring the transactions in the form of service functions, it is easy to capture some high-level structural information about the interaction patterns of the interacting hardware and firmware transactions. As firmware services are usually relatively simple and well-defined actions, commonly-used and well-defined patterns can be found easily.

To this end, this thesis shows how the characterization of the interaction patterns of the transactions can be useful in addressing the challenges in firmware validation. Chapter 3 introduces a common and important class of interaction patterns:

**Stateless vs. stateful:** A transaction is either stateless or stateful. In stateless transactions, values of variables are not retained between transaction instances. Each instance depends only on fresh values read from its inputs or shared variables updated by other transactions. Hence, any possible behavior of a stateless transaction can be exercised in a single instance of the transaction. On the other hand, in a stateful transaction, there is a data dependency between instances and an instance reads some variable values updated by some previous instance. Such variables are stateful variables.

**Producer and consumer transactions:** If a transaction $T_p$ updates a variable $x$ and this computation is independent of other transactions, $T_p$ is a producer of $x$. If a transaction $T_c$ reads a variable $x$ updated by $T_p$, $T_c$ is a consumer of $x$ and $T_p$ and $T_c$ are in a producer-consumer relationship.

These patterns can reduce the search space of verifying concurrent TLM with firmware and hardware transactions. For example, if two concurrent transactions are in a producer-consumer relationship, the producer may not depend on the consumer, exploring all the interleavings of the transactions is not necessary as shown in this thesis. Also, to verify a stateless transaction, it is sufficient to unroll the transaction only once.
Lastly, Chapter 3 shows the value of these patterns in practice. It shows the prevalence of these patterns in practice using the Rockbox open-source mp3 player firmware code and Linux device driver code with the interacting QEMU hardware code. QEMU is the hardware emulator code that includes a wide range of devices. Horn et al. [57] have generated three large usable benchmarks by extracting several device models that can run in standalone mode by eliminating the complex dependencies in QEMU. Also, Chapter 3 shows how the effect of the interaction of the hardware transactions on a firmware transaction, over all possible interleavings, can be captured through additional constraints that can be added to the test generation for the firmware.

1.3.2 Automated firmware testing using firmware-hardware interaction patterns

Chapter 4 describes an alternative approach to firmware validation to consider the use of automated software test-generation techniques. In recent years, concolic testing has emerged as a powerful software test-generation methodology [72]. Concolic testing tools [72] use symbolic analysis to generate concrete test cases along with specific paths of the tested software program. Concolic testing is typically used for unit testing of single-threaded programs. In this work, the plain unit testing result of an instance of a transaction can include infeasible paths if it does not consider the starting state of variables. Depending on the initial state of the target transaction and the shared state with other concurrent transactions, some paths found as a result of unit testing can be infeasible as they cannot be reached from any of the initial or shared states. As a result, the unit testing result may include paths that will never be executed. Chapter 4 exploits the ability of concolic testing tools to perform automatic symbolic execution and to use a constraint solver to automate the test generation algorithms. However, this concolic testing technique 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, i.e., some reported bugs may not be real bugs. An excessive number of false positives can significantly limit its practical use.

Thus, Chapter 4 uses the interaction patterns of transactions to provide a fully automated test generation framework. The verification goal is to achieve high path coverage. 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. This chapter shows how this coverage can be determined for a large space of interaction patterns. Conversely this work can determine if a given number of iterations is sufficient for complete path coverage for this space of patterns.

The implementation of the concolic testing method introduced in this chapter consists of first detecting specific interaction patterns using static program analysis. Then, the custom code generator uses these interaction patterns along with the hardware/firmware transactions to generate a custom sequential program $P$ that considers the transaction interactions. $P$ is test-equivalent to the target firmware code $F$, i.e., (i) the set of tests of $P$ can be directly applied to $F$, and (ii) the test coverage for $F$ can be determined using the test coverage for $P$.

The test-generation methodology in this chapter is fully automated. This testing framework is built using KLEE [11], a concolic testing tool, and Frama-C [26], a static software analyzer. Frama-C provides various analyzer plugins such as PDG (program dependency graph) for C programs. I tested the practical applicability of this framework on Linux device driver code and the interacting QEMU code. This chapter demonstrates the efficacy of the testing method for 15 transactions in these benchmarks.
1.3.3 Completeness bounds and sequentialization for model checking of interacting firmware and hardware

Chapter 5 introduces a new technique for determining a sufficient BMC bound to prove a property or to find a violation. It is a static analysis approach that exploits characteristics of common code patterns found in firmware to provide inexpensive termination checks for unbounded verification.

Based on the computed bound, this chapter shows a new compositional form of sequentialization that exploits specific interaction patterns between the firmware and hardware transactions. This improves the sequentialization method in Chapter 4 to cover all common interaction patterns and does not miss any errors when completeness bounds can be successfully computed.

The novel sequentialization method introduced in this chapter is general and can be independently combined with a broad range of verification techniques. This chapter applies BMC on the resulting program using the widely-used software model checker CBMC [20].

I implemented this methodology and experimented with its practical applicability on code for some Linux device drivers and its interacting QEMU hardware emulator code. This chapter studies three large device benchmarks [57] and demonstrates the efficacy of the model checking methodology for verifying 16 transactions in these benchmarks.

1.4 Thesis contributions

Overall, this thesis takes an important step towards bringing high-level functional modeling techniques to firmware/hardware modeling and exploiting the interaction patterns between the concurrent and unbounded firmware and hardware transactions for automatic verification methodologies, such as test generation and bounded model checking.
Motivated by the shortcomings of existing methods for firmware/hardware co-design, this thesis presents a new modeling framework for firmware systems, i.e., service function based transaction-level model for the co-design of firmware and its interacting hardware components. The key contributions of this modeling framework is that, first, it can be used as a unified formal executable specification model for both a wide range of firmware models and its interacting hardware components. Second, this model enables a useful characterization of their interactions.

This thesis performs an in-depth analysis of specific but common interaction patterns (e.g., the stateless producer-consumer case) that can be observed between interacting firmware and hardware transactions. This thesis analyzes the prevalence of these patterns in practice using Rockbox open-source mp3 player firmware code and Linux device driver code and shows the use of these patterns can be made practical in a variety of firmware/hardware scenarios.

This thesis shows how these specific interaction patterns enable the use of a single-threaded concolic testing framework to generate a complete test set for a firmware transaction even when it is interacting with other concurrent firmware/hardware transactions. To do this, it presents a framework for the interaction-pattern-specific customized algorithms to generate test cases for a firmware transaction interacting with concurrent firmware/hardware transactions using a single-threaded concolic testing tool.

By using the common structural patterns of the model for automated testing of concurrent systems, this thesis avoids exploring asynchronous interleavings of the concurrent transactions. Moreover, the interaction-pattern-specific customized 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.
• This thesis provides automated inexpensive termination checks for BMC of unbounded models for several commonly occurring cases. This thesis presents an algorithm that leverages common code patterns to determine a sufficient BMC bound for the firmware transaction $F$ in the context of infinite loops.

• This thesis describes a general algorithmic framework that reduces the concurrent system with $F$ and its interacting producers to a sequential program $P$ using the bound analysis and the common interaction patterns of firmware and hardware transactions. The code generator automatically constructs $P$, which can be verified by a model checking tool. By combining the bound analysis and the previous sequentialization technique used for automated testing, the single-threaded verification-equivalent code guarantees complete BMC results.

• This thesis shows the efficacy of the testing and model checking methodologies using three published real firmware benchmarks of the Linux-QEMU platform.

1.5 Thesis outline

The rest of this thesis is organized as follows. Chapter 2 presents the previous and current practice of firmware verification. Chapter 3 presents the novel service function-based transaction-level model for the co-design of firmware and its interacting hardware components. Chapter 4 then presents an automated firmware testing methodology using firmware-hardware interaction patterns extracted from the modeling framework. Chapter 5 shows how to determine the completeness bounds for BMC and a complete sequentialization technique using the bound information combined with the interaction patterns. Finally, Chapter 6 presents ongoing and future research directions and then provides some concluding remarks.
Much of the work in this dissertation has been previously published to scientific conferences. A list of these publications is provided in Appendix A.
Chapter 2

Background and Related Work

This chapter presents an overview of past and ongoing research in the field of specification-level modeling methodologies and firmware/software verification. This overview will serve as the foundation for the rest of this thesis. Section 2.1 presents the common concepts of TLMs and their limitations in firmware/hardware co-design. Then, Section 2.2 describes previous efforts to perform firmware and hardware co-design at the specification level. Section 2.3 explains the state of the art in firmware verification and the promise of using software verification techniques to verify TLMs. Moreover, this section describes popular software verification techniques and their known limitations, especially when applied to firmware verification.

2.1 Transaction-level models (TLMs)

This section presents the fundamental design concepts of TLMs in hardware/software modeling techniques.
2.1.1 What is a TLM?: Common concepts

There have been various efforts to develop early design models for both firmware and hardware in a common modeling language such as SystemC [8] that enables their co-simulation. TLMs [14, 40, 71] serve as a middle point design between purely functional descriptions and detailed implementations, such as RTL models. A TLM is a functional model that captures the interaction between components without necessarily providing all the low-level details. Some works [14] define TLMs at different levels such as the specification level, the architecture level, and so on. Specification-level models only describe the functionality of the system and are free of any implementation details. Architectural models include channels for communication and synchronization methods. However, the key idea of TLMs is to abstract away all the implementation details, such as the communication protocol among computation components, and to emphasize the functional operations of the data transfers, such as reading or writing carried out in the communication methods. These details of communication and unnecessary details of computation are hidden in a TLM and added later in the design cycle. Moreover, while every operation is synchronized in low-level implementation models, TLMs are typically asynchronous and clock-free (some TLMs may have lose timing annotations). The mechanism for scheduling the order of concurrent operations of TLMs is non-deterministic. Physical parallelism can faithfully be represented by the set of all possible schedulings.

Advantages of TLMs

All of the abstractions described above enable TLM simulations to run much faster than RTL simulations, with a difference of multiple orders of magnitude. For example, simulation times for encoding/decoding a picture in the MPEG 4 format is about a thousand times faster in TLM than in RTL models [25]. Also, a TLM can serve as an executable golden model, and can be used as a reference for synthesizing RTL models. Moreover, a
well-defined TLM framework enables the reuse of some components (both models and test-benches) at different stages of the development cycle. Lastly, TLMs make it easier for the system-level designers to try alternate architectures for communication while maintaining support for a common abstract instance.

Limitations of TLMs

The biggest weakness is the fact that there is no clear definition of TLMs. This prevents easy usage of predefined TLMs. Furthermore, due to the unclear definition of TLMs, the usage of TLMs in various design stages, e.g., modeling, validation, refinement, exploration, and synthesis, cannot be systematically developed. As a result, the advantages of TLMs are not practically beneficial during system development. In order to eliminate such ambiguity, there have been various attempts to explicitly define TLMs. Each definition is typically adopted for a different design purpose.

Moreover, in practice, any simulation engine runs a deterministic scheduler, i.e., specific values are used for the non-deterministic thread execution schedulings (delays). Hence, only a small subset of all the possible schedulings between concurrent components is explored during simulation, and there can be missed bugs. Even if the scheduler can in principle simulate the complete set of schedulings, the number of the schedulings is too large. Hence, many researchers have been trying to cover the set of schedulings efficiently using various techniques such as partial order reduction while aiming for complete verification results [51].

2.1.2 Various TLMs

There are a variety of notions of TLMs used in hardware/software modeling. This section shows how each TLM is adopted for a specific design purpose. Among the earliest uses is in
the context of databases where a transaction is an atomic execution of operations accessing a shared database [5].

In the hardware modeling context, Ghenassia [40] introduced TLMs as an early SoC design methodology at a higher abstraction level (above RTL) in the design flow. The abstraction level lies between the cycle-accurate model and the untimed functional model in their TLMs. They allow the addition of timing annotations to the untimed functional TLMs so that early analysis of the model is possible, yet they avoid extremely slow simulation speeds as in the cycle-accurate models.

More specifically, an SoC design is composed of various components where each component has a finite set of states and a set of concurrent behaviors (or processes). In Ghenassia’s TLMs, a TLM module represents each of these components, and the modules communicate with each other through channels. Depending on the accuracy level, a channel can take various forms, such as a bus structure. However, the communication and the computation of the modules are clearly separated. In their TLMs, the states are modeled as a finite set of variables. Also, the concurrent behaviors are modeled by a set of concurrent processes (or threads). In this model, a transaction is the set of data transfers (communication) between modules through abstract channels. Such communication protocols are encapsulated within channels to separate communication from computation in TLMs. Replacing low-level signals or buses with transactions raises the design abstraction level. Example transactions are word transfers between two registers, or image transfers between two memory buffers.

TLM in SystemC

In addition to a clear definition of TLMs, a concrete programming language is needed in practice to implement the models. However, neither traditional hardware description languages, aiming for lower-level models, such as Verilog or VHDL, nor popular programming languages in which the parallelism cannot easily be supported well, such as C/C++, are a fit for the existing TLMs. Hence, a number of modeling languages for describing the high-level abstraction models have been introduced over the previous decades. SystemVerilog [98] extends the HDLs. SpecC [38] is a system description language based on the ANSI C programming language. Recently, SystemC [8] has become the industry standard modeling language for TLM because of its advantage in hardware-software co-design and its ability to represent multiple levels of abstractions [105][100]. It is a C++-based system language supporting parallelism. Since SystemC is basically a C++ library, it has the advantage that a standard compiler can be used.

A SystemC TLM comprises a set of parallel components (representing modules) and connections between them. A component may have ports for communication with other modules. Each component’s behavior is defined by a set of possibly concurrent processes. The key concept in SystemC TLMs is the scheduler, which manages the concurrent execution of the processes/threads. A SystemC executable includes (1) the static architectures with the components and connections and (2) the associated simulator that runs the processes by realizing parallelism. SystemC also provides synchronization methods for the concurrent processes.

SystemC provides the Open SystemC Initiative (OSCI) [89] library that enables standard TLM modeling. This library provides a set of classes to model hardware from RTL to function level, user-defined communication mechanisms, discrete-event simulation kernel for concurrency, and so on.
SystemC is widely used and supported in the industry now. Not only can the existing tools (compilers, debuggers, etc.) for C++ be reused, but there are also various tools that support co-simulation of SystemC models with other HDL models. Not surprisingly, integrating a SystemC model (at the function level) with C/C++ code is straightforward. All of this support combined with the ability to manage concurrency is beneficial for firmware/hardware co-simulations and developing embedded systems.

The main challenge associated with using SystemC TLMs is that the System-C scheduler typically does not/cannot consider the prohibitively large number of possible interleavings between the concurrent hardware and firmware. Thus, this simulation is incomplete with an unknown coverage of interleavings, thus missing possible bugs.

### 2.1.3 Concerns in TLMs for firmware

As the previous section discussed, there are various forms of TLMs at various abstraction levels. However, the widely used TLMs, such as SystemC TLMs or Ghenassia’s TLMs [40], are suitable for hardware rather than firmware/software, as their detailed semantics match the underlying hardware components. Hence, this thesis introduces a novel definition of TLM for firmware and its interacting hardware while considering the following points of view for modeling firmware at the function level.

**The basic unit of work**

Previously introduced TLMs are typically viewed as reference models for hardware systems. Here, a transaction, the basic unit of work in TLMs, is essentially a communication function call between the various hardware components. However, this previous concept of transactions may not be the best match in firmware systems. Due to the fundamental difference between firmware and hardware systems, it may be preferrable to re-define a
transaction, or the unit of work, for firmware systems so that a divide-and-conquer strategy can be used for both development and verification.

**The reactive nature of firmware**

As explained in Chapter [1], firmware functions are often reactive. Event-driven state machines can be a good candidate for expressing this nature (the environmental input can be seen as an event from the environmental software/hardware layers) while supporting concurrency at the same time. Event-driven state machines can be implemented in software languages, such as C/C++ with some additional supports. For example, previous work [67] provides function pointers to specify the state machines explicitly. However, the resulting code can easily be too complicated for both programmers and compilers. This also results in bugs in the program.

**Non-deterministic delays**

This thesis aims to introduce an executable specification-level TLM for both firmware and hardware. At the specification level, TLMs are asynchronous and clock-free, and the scheduling mechanism of such TLMs is non-deterministic to support parallelism. Fortunately, this is easy to model. Two concurrent threads accessing the same memory space can simply assume a non-deterministic execution speed with respect to each other. One can assume that it does not know how many clock cycles at the lower level would be consumed to execute a transaction, so it would simply assume that it is non-deterministic. This enables the creation of a semantic connection between the specification level and the lower level of the systems.
Communication scheme

Specification level models typically model data transfer among processes through the accessing of variables. This makes use of high-level languages, such as C/C++, suitable for implementing the specification-level models.

2.2 Various forms of specification-level modeling of firmware and hardware

The previous section introduced the most widely used TLM forms for hardware designs. This section studies various previously introduced forms of modeling frameworks for firmware and hardware co-design at the specification level. This section focuses more on the relationship between firmware and hardware and their high-level data computation.

2.2.1 High-level specification modeling of firmware and hardware for their concurrent development

Recently, many works have emphasized the importance of high-level specification modeling of firmware and hardware for their concurrent development. Jerraya et al. [59] introduced a high-level parallel programming model for the hardware/software interface in the case of a heterogeneous multiprocessor system-on-chip (MPSoC) design. The model provides a set of functions (implicit and/or explicit primitives) that can be used by software to interact with hardware in the heterogeneous MPSoC design. Heinen et al. [50] introduced a formal specification of the hardware/firmware interface for consistency in system verification.
2.2.2 Integrating dataflow with finite-state machines

To model concurrent systems at the high level, many approaches integrate dataflow with finite-state machines (FSMs). These approaches support concurrency and sequential data computations at the same time. *charts (pronounced “star charts”) [41] is one of the early approaches that integrate FSM semantics with concurrency semantics. This modeling work suggests that flexibility in concurrency models allows an arbitrary nesting of dataflow charts. Besides the *charts, there are similar modeling approaches, such as SysteMoC [62], California Actor Language (CAL) [33], and Extended Codesign Finite-State Machines (ECFSMs) [92], etc. Falk et al. [34] use the SysteMoC C++ library (based on SystemC) as the model of computation for the FSMs. Here, the actions of the FSM correspond to high-level aspects of the specification, e.g., packet and interrupt types, connection modes, etc. Using this modeling framework, they enable measurement of functional coverage, high-level performance estimation, prototyping for firmware/hardware, etc., in the domain of networking.

2.2.3 Reactive models

As indicated in Chapter 1, modern firmware is inherently asynchronous and concurrent in order to interact with real-world inputs and to increase the performance of the system. Hence, many works naturally model a firmware system as a set of asynchronous concurrent tasks running repeatedly. Time-bounded periodic programs are one of the commonly used models for real-time firmware systems [15, 16]. In this model, there is a set of asynchronous tasks running periodically, and there typically exists a scheduler which manages preemptive schedulings based on their priorities. In practice, many real-time OSs, such as OSEK [82] and RTEMS [90], support these periodic programs. As an example, the nxt/OSEK-based [82] LEGO Mindstorms controller for a robot has four periodic tasks: a TapeMover (moving the tape), a Reader (reading the current symbol), a Writer (writing the
current symbol), and a Controller (managing the other three tasks). Each task has its own period and priorities. Note that these works do not consider their hardware environments even though the hardware functionalities can be defined as a set of concurrent tasks as well.

To sum up, in this thesis, firmware-hardware co-design in the specification level has been developed to emphasize the following: (1) supporting concurrency, where the unit of concurrency is a sequential data computation which can easily be modeled using FSMs, and (2) focusing on the unit of work, or the specific task of the reactive firmware system. Based on these aspects, this thesis provides a novel form of TLMs at the specification level for both firmware and hardware in Chapter 3.

2.3 Firmware verification

This thesis introduces novel specification-level co-verification techniques for complicated firmware-hardware systems with a large search space. Hence, this section presents an overview of past and ongoing work in firmware verification. Furthermore, it explains the applicability of well-developed software verification techniques in verifying firmware-hardware TLMs.

2.3.1 The state of the art in firmware verification

Verification is a mathematical process to prove the absence of errors in the target program. This section introduces a number of firmware verification methodologies. Model checking tools, such as SLAM [4] and KISS [86], formally verify device drivers. SLAM finds Windows API usage errors in device drivers written in C programs using a static analysis engine. SLAM is built on top of the Static Driver Verifier (SDV) tool. In this work, the environment model represents the OS initialization and invokes the device drivers and
the kernel API calls. However, SLAM handles only sequential models. KISS is an automated model checker for concurrent C programs built on top of SLAM, which is meant for sequential C programs. KISS successfully detected 30 race conditions in Windows NT device drivers, but it does not completely cover the input programs as its concurrency analysis bounds the number of context switches.

Kumar et al. [66] extract abstract models of device firmware written in ESP [67] so that these abstract models can be applied to model checkers, such as SPIN [54]. Such an abstraction can be essential for obtaining simpler models to easily check system-wide properties like absence of deadlocks. In fact, they found seven deadlock bugs with the abstract models in VMMC firmware [30].

Chaki et al. [15] introduce time-bounded verification techniques to verify logical properties of the real-time embedded systems, such as user-specified assertions, race conditions, or API usage rules. The embedded systems are modeled as periodic programs with predefined priorities and periods in this work. They provide a sequentialization technique that takes into account inter- and intra-hyper-period temporal separation between tasks of the concurrent programs. This results in a dramatic increase in scalability while reducing false positives.

However, these works explained above do not take the hardware environment into account. At best, they have an interface environment for simulating inputs. Ignoring the effect of the interacting hardware can result in an over-approximation of the set of possible firmware behaviors, and thus false positives, i.e., infeasible behaviors, during firmware verification. To faithfully take the hardware environment into account, hardware/software co-verification for SoCs is typically done by integrating a C/C++ simulator and an FPGA emulator [80]. However, setting up the co-simulation environment is known to be a difficult job. Moreover, the communication via even a flexible interface, such as shared registers, has a large amount of overhead.
Hardware/software integration and co-verification are easier with Bluespec [81]. Bluespec is a hardware tool, which enables synthesis of RTL hardware models using a high-level functional hardware description language. In this language, hardware functionalities are described using rules and methods. Bluespec also enables early hardware emulation, architectural exploration, and verification. Further, it supports firmware development at the block, subsystem and system level. With this, firmware models can be integrated and run with RTL IPs at high speed (MHz speeds). This can be leveraged for simulation-based firmware verification. Although pre-silicon firmware development or verification is possible using Bluespec, Bluespec does not support hardware or firmware verification above RTL. Singh et al. [97] verified Bluespec SystemVerilog (BSV)-based hardware designs at the specification level. They convert the BSV-based hardware designs into PROMELA-based specification models (PROMELA [58] is a high-level language for specifying system descriptions supported by the SPIN [54] model checker). Then, using SPIN, they check the safety properties and the equivalence between the specifications and the implementations.

### 2.3.2 The promise of using software verification techniques for the high-level models

Unified models for both firmware and hardware at the specification level introduced in this thesis can overcome the limitations of previous firmware verification work. These specification-level models enable the use of well-established software verification techniques for firmware and its interacting hardware components. The complicated integration of two different types of simulators/emulators can be avoided as well. Chapters 4 and 5 introduce how this thesis exploits the benefits of using software verification techniques for firmware verification with its interacting hardware components. Hence, the following section discusses various types of popular software testing and verification methodologies.
Testing

In hardware and software systems, testing aims to find bugs in the code and to assess how well the code behaves as intended in its specification. However, guaranteeing the correctness of a program using testing is difficult as it requires testing the model under all possible inputs. Still, testing is the most popular verification method in practice. Most companies verify a program’s correctness via testing, and thus there is significant research in automated testing methods. Testing is typically done by simulating the target system with input test vectors where each test case represents a testing scenario, or a program behavior. Hence, generating a set of test cases with the aim of maximizing code coverage for the target program is the testing goal. These input test cases can be acquired in various ways: by regression testing, which accumulates test cases over a long time period, by random selection, which randomly chooses test cases, or by symbolic simulation of the target program \([64]\). Symbolic testing methods execute the target program symbolically to cover all feasible paths related to the given symbolic variables. In conjunction with symbolic execution, an automated theorem prover or constraint solver is used to generate concrete inputs (test cases) for the symbolic variables.

The concept of concolic testing (concrete and symbolic hybrid software testing technique) was introduced in DART \([44]\) by Godefroid et al. and extended by the testing tool CUTE \([96]\). KLEE \([11]\) (renamed and improved from EXE \([12]\)) provides unit testing of C programs and concolic testing, and jCUTE \([94]\) supports concolic testing for multithreaded Java programs. Microsoft Research’s SAGE \([45]\) also uses a concolic testing technique together with fuzz testing to detect security bugs in Windows applications.

Testing methodologies for sequential programs have grown very sophisticated and there exist various methodologies and tools for this. However, testing of concurrent systems has presented difficulties, which cannot easily be solved with previous testing techniques. There can be too many interleavings between the concurrent threads. The number of inter-
leavings easily blow up even with small-sized input programs. Hence, a number of studies have been trying to systematically explore the possible interleavings based on the previous testing method for sequential programs and some special techniques, such as partial order reduction [47] and bounded context switches [86]. These concurrent program analysis techniques are explained in depth in Section 2.3.3.

Although testing is one of the most common bug-finding methods in practice, it is inherently incomplete: testing is considered a scalable verification method for large-scale systems, but it does not mathematically prove the completeness of the program. Therefore, exploring/executing all possible input test vectors is nearly impossible. This gets worse with concurrent programs. Only a small set of interleavings is explored among all possible interleavings between the concurrent threads. Hence, for some critical software, such as security applications, relying only on testing can be hazardous.

**Formal verification**

While testing still does not guarantee complete verification results, formal verification methods are exhaustive and guaranteed to search for all possible violations of the property under verification. Formal verification uses formal mathematical methods to prove or disprove the correctness of the target system with respect to a certain formal property. Traditionally, this is done by abstract static analysis. The key idea of static analysis is to efficiently compute approximate but sound guarantees. Static analysis techniques typically apply a monotonic function iteratively until a set of program variable values saturates, i.e., until the fixed point is reached and no more program state needs to be explored. However, this method can only show the absence of simple errors as it typically uses context-insensitive analysis, i.e., static analysis techniques trade precision for efficiency by using abstraction. Moreover, this abstract static analysis is not able to generate counterexamples in many cases due to the precision loss. In contrast, the model checking methodology...
has grown to be the most widely used formal verification methodology, overcoming these limitations of the static analysis.

Model checking

One of the most commonly used formal verification methods is model checking. Model checking is an algorithmic method that determines if the target system always satisfies a correctness property \[19\]. Such properties are in the form of logical formulas in temporal logic, usually either safety or liveness properties. Safety properties assert that nothing bad happens. Liveness properties guarantee something good eventually happens. Note that it is hard to assert a liveness property within a finite execution of the target system because the “good” event might occur after the execution ends. The target system is modeled as a transition system with states (evaluation of the values of the variables, the program counter, etc.) and transitions. Transitions represent the progress from one state to another state. Model checking algorithms check the correctness of the system by exhaustively examining the reachable states of a program from the initial states and the transitions. Then, if the model checker finds a state violating the given property, it produces a counterexample.

Model checking methodologies prove complicated correctness properties of the programs and generate counterexamples, i.e., diagnostic execution traces demonstrating the error. However, model checkers often suffer from the state-space explosion problem \([29]\). The state space of a software program can grow exponentially with respect to various parameters, e.g., the number of variables, the size of data types, function calls, dynamic memory allocation, and so on. This is exacerbated by concurrency because it is required to consider all possible interleavings between the concurrent threads while exploring the program control location state space. The number of interleavings is exponential in the number of statements. The state-space explosion problem is the key issue in model checking, and the
following model checking approaches include techniques to efficiently manage the large state space.

- Explicit-state model checking

Explicit-state model checking algorithms explore the state space by recursively exploring state transition graphs starting from the initial states in various manners, e.g., depth-first, breadth-first, heuristic, etc. As the state space is explored, property violations can be checked on the fly. Hence, model checking can terminate early without exploring the entire state space. However, there can be issues with shortage of memory. For this reason, explored states are compressed and stored using hash tables [55]. This technique sometimes causes hash collisions and misses error states as a result, but in reality, this rarely happens (0.1%) with billions of states.

- Symbolic model checking

Symbolic model checking [75] proves the correctness of the input program for a range of inputs. Symbolic model checking algorithms avoid building state graphs by using implicit representations of set of states. Example representations are symbolic formulas, such as Binary Decision Diagrams (BDDs) [10], propositional logic for finite sets [7], or finite automata for infinite sets [63]. A BDD shares maximal number of nodes to eliminate redundant nodes from a Boolean decision tree. A BDD representation is canonical for a fixed variable order. Hence, it enables functional equivalence checking efficiently [73]. However, it can grow quickly. Also, selecting the right order of variables is important for space efficiency, but generating a variable ordering that results in small-sized BDDs can be time-consuming [7]. In symbolic model checking, properties can be verified by manipulating these formulas.

Alternatively, bounded model checking (BMC), a symbolic model checking methodology that uses SAT procedures, is widely used. It was introduced in 1999 by Biere et al. [7], and now is the most popular methodology in the semiconductor industry.
BMC explores the reachable states within a bounded number of steps, for example $k$: the BMC tools construct a Boolean formula that is satisfiable if and only if there is a counterexample of length $k$. BMC is much more efficient than symbolic model checking using BDDs in terms of space since it does not compute sets of visited or reachable states. Also, it does not require knowledge about the data structures of the program. However, BMC is usually useful for programs without deep loops.

While explicit-state techniques work well with error detection, symbolic methods are suited for proving correctness with a large number of program variables and large data types. However, complicated programs with a large number of program paths and variables have a large state space. The following approach (abstraction refinement) analyzes an abstraction of a program, which provides a simpler abstraction of the state space.

- Abstraction refinement

In counterexample-guided abstraction refinement (CEGAR) [19], the model checker first verifies the abstract model. If verification fails, an abstract counterexample is generated. If the abstract counterexample does not occur in the concrete program, it refines the abstract model to a more precise abstraction. Then, the model checker repeats the process until it determines the abstraction is safe or finds a feasible counterexample. This loop has four phases: abstraction, verification, simulation, and refinement.

Microsoft Research’s SLAM [4] is one of the successful pioneers for this technique. It was able to verify dozens of predefined properties of Windows Device drivers, and the Static Driver Verifier tool, a tool based on SLAM, is currently part of the Windows Driver Development Kit. BLAST [53] uses lazy abstraction, i.e., the refinement selectively refines only the relevant parts. While both tools use general-purpose the-
orem provers and BDDs, SATABS [21] uses a SAT solver for symbolic executions and constructing abstract models.

2.3.3 Challenges in applying software verification techniques to firmware verification

The two major challenges in current and future software analysis are concurrency and the environment problem. Firmware models under test need to capture a nontrivial hardware environment that the firmware code concurrently runs in. Furthermore, firmware functionalities are often unbounded as they may be repeatedly executed an unbounded number of times. Unboundedness of the target code is also a key challenge of many model checking techniques, such as BMC, because without a threshold, the completeness of verification is not guaranteed.

This section describes these challenges and the state of the art of the various attempts in the software verification field to solve the problems, especially for firmware (low-level software) interacting with hardware components.

Concurrency analysis

One of the biggest challenges for software verification is concurrency of the software code. It has been studied for many years, but few tools are able to analyze concurrent programs, and these suffer from poor scalability.

For concurrency analysis, a number of verification tools examine interleaved executions. Traditionally, partial order reduction has been used to mitigate issues of the state-space exploration of concurrent programs [47]. For some concurrent instructions, the order of their execution may result in the same state, i.e., their execution order does not matter to prove the properties. Hence, the set of interleavings that does not affect the property can
be grouped into a single class. A model checker using this partial order reduction method generates only one representative interleaving for each class while constructing the state transition graphs. This can reduce the state space exponentially with respect to the number of threads in the best case. SPIN [54], JPF [48], Bandera [24], etc., analyze multithreaded programs using the partial order reduction technique. The Verisoft [43] tool also systematically explores the state space of concurrent programs using partial order reduction. It does not store information about the visited states and, therefore, is a stateless search. Verisoft theoretically could be applied to SystemC. Some other works [9, 51] have used partial order reduction to cover complete interleavings of asynchronous SystemC models. SCOOT [9] is a model checking tool for SystemC based on POR. It used the commutativity of transitions between processors to reduce unnecessary interleavings. Although the partial order reduction technique covers complete behaviors of the programs, the state space can still grow exponentially in the worst case.

To overcome the scalability problems, recent methods bound the number of context switches between the concurrent threads. Lal and Reps [68] reduced concurrent programs to sequential ones under a given context switch bound with different program abstractions. CHESS [79] repeatedly executes idempotent tests, and systematically searches interleavings each time while bounding the number of context switches. KISS is a static analysis tool for multithreaded C programs [86]. It translates a concurrent program into a sequential one by bounding the number of context switches using a single stack and a nondeterministic scheduler. It simulates a large subset of the program behaviors. This sequentialization method is scalable, but the verification result is inherently incomplete as it limits the number of context switches. Many of these works are able to catch meaningful errors but still miss many scenarios as the bound is usually low. Moreover, a recent work [56] showed that bounded context switching technique often effectively increases both the state space and time thereby cannot be considered as reduction.
Beyond these algorithms, there exist additional algorithms that examine concurrent programs: Cook et al. analyze the interleavings of Boolean programs with unbounded thread creation by using an over-approximated set of reachable states [22]. Instead of examining the feasible interleavings, some works [52, 60] use rely-guarantee reasoning by performing modular verification, i.e., by verifying each thread individually using summarization of the behavior of all other threads. Edelstein et al. [31] used a heuristic to increase the probability of observing race conditions by injecting seeded delays instead of exploring all interleaving.

In spite of these various efforts, in practice, existing well-developed and popular verification techniques, such as symbolic testing methods or BMC, are mostly used for sequential code, and they cannot directly consider the interaction of other hardware/firmware threads with the target firmware thread during validation.

The previous analysis techniques for concurrent programs, such as partial order reduction, are either scalable or complete, but not at the same time. On the other hand, this thesis introduces sequentialization methods that are both complete and scalable for a set of input programs with specific interaction patterns (the stateful/stateless transactions and the producer-consumer relationship) between the firmware and hardware transactions that can be analyzed from the TLM. Specifically, this thesis shows how these patterns, along with the firmware and hardware transactions, are used to automatically generate a sequential program that is test-equivalent to the target firmware transaction and that can be used with standard sequential program testing tools or model checkers.

2.3.4 Environment problem

A formal model is required to formally verify the design under test at any level. In software analysis, the input model often includes the environment: libraries, other applications, or hardware components in which the software code runs. As programs often heavily count on
behaviors of their environment, the environment models are needed to accurately analyze the program behaviors. However, the size of such an environmental model is often non-trivial, and manual effort is needed to implement the environment. Furthermore, a significant amount of work is required for integrating the environmental model and the software model. This problem is even worse in the interfaces between hardware and firmware than the interfaces in a homogeneous system. Unfortunately, combining separate verification techniques for firmware and hardware leads to poor verification results because of the difference in languages and tools, data sharing methods, or concurrency. To this end, today’s trend in firmware development is to utilize virtual prototyping. For example, a high-level software model for the interacting hardware components is employed for co-validation of hardware and software (firmware) interfaces [57]. This thesis also follows this trend and introduces a unified modeling language for firmware and hardware interfaces at the specification level.

2.3.5 Unboundedness problem

Firmware code often contains (unbounded) loop structures because of its reactive nature. BMC is a suitable verification technique for such code, as it discovers counterexamples of deep bugs after exploring a large number of iterations/unrollings of such loop structures within a given bound $n$. However, without a threshold for $n$ that guarantees the completeness of verification, this method cannot prove correctness. A key challenge in verifying unbounded programs lies in computing such a completeness threshold.

More recently, SAT-based methods [76, 77], such as interpolation based methods, have been applied to verify unbounded models. However, in these methods, the verification engines need to implicitly store the set of reachable states in every transition. This is expensive and generally limits the applicability of these techniques for practical programs. Hence, one of the key contributions of this thesis introduced in Chapter 5 is a new technique
to determine a sufficient BMC bound to prove the property or find a violation. It is a static analysis approach that exploits characteristics of common code patterns found in firmware to provide inexpensive termination checks for unbounded verification.

### 2.4 Chapter summary

This chapter presented the necessary background for understanding the goals and mechanisms presented in this thesis. The subsequent chapters explore solutions to the problems posed by this chapter; namely, the need for scalable analysis techniques of the concurrent firmware systems and the need for termination checks for unbounded firmware TLMs.
Chapter 3

Service function based TLM

3.1 Introduction

Chapters 1 and 2 highlighted, among other things, the problem that co-verification of firmware and hardware is difficult due to their fundamental differences and their concurrency. This chapter first introduces the service function-based TLM that is suitable as a high-level unified modeling methodology for both firmware and hardware. The TLM is composed of concurrent firmware and hardware transactions where each transaction is a service function with a finite duration.

Second, in structuring the transactions in the form of service functions, this chapter shows that the validation challenges mentioned in Chapters 1 and 2 can be alleviated by exploiting specific interaction patterns between the transactions. Specifically, this chapter studies the specific but commonly occurring interaction patterns between the firmware/hardware transactions. Then, this chapter shows how the interaction patterns enable single-threaded concolic testing to be used for multi-threaded testing for a large and important class of interaction patterns (the stateless producer-consumer case discussed in this chapter). In
case of the specific interaction pattern, our method guarantees complete testing results for practical-sized examples.

This chapter is organized as follows:

- Section 3.2 briefly explains the running firmware example in this chapter, the Rockbox MP3 player system. Rockbox provides a non-trivial example of public domain firmware code.

- Section 3.3 presents a novel service function based transaction-level model for the co-design of firmware and its interacting hardware components. We then show how this model enables a useful characterization of their interactions.

- Section 3.4 shows how specific interaction patterns (e.g., the stateless producer-consumer case) enable the use of a single-threaded concolic testing framework to generate a complete test set for a firmware thread even when it is interacting with other concurrent firmware/hardware threads.

- Section 3.4 also demonstrates the practical applicability of the proposed modeling methodology through its application to Rockbox and the Linux device driver code with its interacting QEMU emulated hardware code. This section shows the practical applicability of the proposed concolic test generation technique through the use of a public domain concolic test generator, KLEE [11], on the firmware threads in Rockbox [11].

Overall, this chapter takes an important step towards bringing high-level functional modeling techniques to firmware/hardware modeling and exploiting the interaction patterns between the concurrent firmware and hardware threads for automatic test generation.
3.2 Rockbox case study

Our running example Rockbox [1] is composed of application code and firmware code, as shown in Fig. 3.1. The application code consists of hardware-independent MP3 player software, such as playback, encoding, decoding, etc. On the other hand, the firmware code consists of the device-independent part and the device-specific part (Rockbox categorizes both parts as firmware, but in this thesis only the device-specific part is classified as firmware). The device-independent part includes a kernel, a multi-threading scheduler, and device driver threads, which access the hardware device using assembly-level driver API. For the device-specific part, Rockbox supports a wide range of MP3 players, hence there are various firmware back-end assembly-level APIs for the target devices. This chapter uses the iAudio X5 device with ColdFire SCF5250 [36] microprocessor as the target device. The main Rockbox firmware is the device driver threads: the power thread for power management, the USB thread that handles USB insertion/removal events, the backlight handling thread, the advanced technology attachment (ATA) disk management thread, etc. Each thread executes its service function in response to an input from the hardware or application-level software. As an example of a service function, the power thread’s job in each call is to (1) update the charger connection status, and (2) keep watching the battery.
level by reading the device (PCF50606 controller chip [84]) status. Hence, these service actions in each call depend on the inputs from the PCF50606 device. Similarly, the hardware device provides service functions in response to the firmware. For instance, one of the iAudio X5 chips [88], PCF50606, which manages the physical power supply, provides a number of services. One example service of this chip is converting the analog battery level to digital and updating the value on the shared register in response to the physical voltage level change. Another service is to update the charger connection status. The next section explains the novel service function-based TLM using this example.

3.3 Transaction-level model

In this section, we formally define a novel TLM framework that exploits several useful transaction interaction patterns.

3.3.1 Model definition

We define the TLM framework based on the service functions.

Service function

The functionality of a firmware program can be organized as a set of services. A service function responds to specific inputs, possibly from the hardware or software environments, and provides a corresponding service. To illustrate the service function through the Rockbox example, the service function of the power thread responds to the charger connection status change and/or the battery level change. The service function updates the charger connection status and reads the physical battery level in order to perform necessary actions, such as preparing for battery-saving mode in every execution. Another illustrative
example is the USB driver thread. 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 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. A service function is modeled as a transaction in the service function-based TLM in this thesis.

Transaction level model (TLM)

A transaction is formally defined in the form of a high-level state machine (HLSM) \([104]\).

**Definition 1** (HLSM). An HLSM is defined as

\[
(S, I, O, V, s_0, v_0, \omega, \nu, \delta),
\]

where \(S\) is a set of states, \(I\) is a set of inputs, \(O\) is a set of outputs, \(V\) is a set of variable values, \(s_0 \in S\) is the initial state, \(v_0 \in V\) is the initial variable value, \(\omega : S \times I \times V \rightarrow O\) is the output function, \(\nu : S \times I \times V \rightarrow S\) is the state transition function, and \(\delta : S \times I \times V \rightarrow V\) is the variable update function.

HLSMs extend FSMs by distinguishing the data states from the control states. This is related to the notion of extended finite-state machines (EFSMs) \([18]\). HLSMs treat the data state \(V\) as separate from the control state \(S\). This allows HLSMs to describe algorithmic computation as a series of data updates (possibly conditional). In the following, \(s \in S\) (possibly subscripted) represents a specific value of state. \(v\) represents a specific value from \(V\). Moreover, HLSMs allow arithmetic operations for \(\omega, \nu, \) and \(\delta\). Each state of HLSMs computes state transition functions and/or performs arithmetic computations for data update.
**Definition 2** (Transaction). A transaction $T$ is an HLSM with a start state $s_s \in S$ with no incoming transition and an end state $s_e \in S$ with no outgoing transition.

Formally, a transaction is an untimed HLSM. The distinctions between an HLSM and a transaction are (1) a start state in a transaction without any incoming transitions, and (2) a clear end state with no outgoing transitions. These characteristics are important for capturing the service function nature of our transactions. A firmware service function starts in response to an external input from hardware or application/OS and ends at the completion of the service task. This is modeled by means of a single departure from the start state, and a terminating end state.

For a transaction, $V$ represents both data values that are local to the transaction and data values that are shared with other transactions. Thus, $V = L \times G$ where $L$ is the set of local variable values and $G$ is the set of shared variable values. Let $v = (l, g)$ where $v$, $l$, $g$ represent variables that take values from $V$, $L$, $G$, respectively. Note that $V$ may be composed of several different sets of variable values $V_1, V_2, \ldots, V_k, \ldots, V_p$, i.e., $V = V_1 \times V_2 \ldots \times V_k \ldots \times V_p$. Equivalently, $v = (v_1, v_2, \ldots, v_k, \ldots, v_p)$, where $v_k$ represents a (either local or shared) variable that takes a value from $V_k$. Different transactions communicate through the shared variables. A state $s$ in a transaction reads variable $v_k$ if some variable update in $s$ depends on $v_k$, or some transition from $s$ depends on $v_k$. $s$ writes $v_k$ if it modifies the value of $v_k$.

**Definition 3** (Transaction Execution Model). A transaction instance $t_i$ is an execution of a transaction $T$. When $T$ executes, $t_i$ starts from the start state $s_s$ and continues execution until it reaches the end state $s_e$. $t_i$ is followed by the next instance $t_{i+1}$.

In other words, the execution of a particular path of a transaction is a transaction instance. The notion of a transaction instance is borrowed from Mahajan et al. [71], but our transaction instance is not overlapped with the previous instance of the same transaction. Instead, transaction instances of a transaction execute sequentially. Hence, the value of $v$ at the end
of $t_i$’s execution equals the value of $v$ when the next instance $t_{i+1}$ starts. Only one instance of a transaction can be active at a time while instances of different transactions can be executed concurrently. Note that, depending on the values of $i$ and $v$, transaction instances may follow different paths through the transaction. Hence, each instance is not necessarily identical with other instances of the transaction.

**Definition 4.** A TLM $\mathcal{T}$ is a pair $(\Omega, \Phi)$, where $\Omega$ is a set of concurrent transactions and $\Phi$ is a set of the initial values of $v$ of each transaction $T \in \Omega$.

Note that since $v = (l, g)$ and $g$ is shared across transactions, $\Phi$ must have consistent values of $g$ across transactions.

A TLM can consist of both firmware and hardware transactions where each transaction is a specific service function. Their interactions are captured through the shared variable $g$. For instance, the two services of the PCF50606 chip can be modeled as two hardware transactions providing the physical battery level and the charger status to the power firmware transaction through the shared variables. These three transactions run concurrently. *This uniform modeling of the hardware and software as transactions provides a uniform analysis framework for studying their interaction patterns as well as a consistent modeling method for both firmware and hardware.*

The execution of our TLM model, comprising firmware and hardware transactions, is an asynchronous interleaving of state updates which come from the infinite streams of the firmware/hardware transactions.

The shared variables $g$ provide the interactions between concurrent transactions. However, concurrency can result in many possible interleavings between the transactions that can impact the values of $g$ and thus the paths taken in the transaction instances. Consider two concurrent transactions $T_1$ and $T_2$ where $T_1$ reads $v$, and $T_2$ writes $v$ in every instance. Note that, in our TLM model, individual reads/writes to shared variables are atomic, but the entire transaction is not. In this example, $t_1$ (an instance of $T_1$) reads the value updated
by the last instance of $T_2$ executed before $t_1$. This last instance of $T_2$ is determined by the relative speed of $T_1$'s instances and $T_2$'s instances and their interleaving. Thus, to get the complete set of possible behaviors of the transactions, all possible interleavings need to be considered.

### 3.3.2 Analyzing data dependencies between transactions

We now show how specific interaction patterns between transactions can be exploited to reduce the number of interleavings. In Rockbox, we observed that there are a few interesting interaction patterns between transactions based on the data dependencies. We first define the notions of data dependence and data dependence graph and describe the specific interaction patterns of interest based on this. While these terms are well known in the compiler context, it is useful to define them in the current TLM context.

**Definition 5 (Data Dependence).** Variable $v_l$ in state $s_m$ in transaction $T_n$ is data-dependent on variable $v_i$ in state $s_j$ in $T_k$, if $s_j$ updates $v_i$, and this update determines the update of $v_l$ in $s_m$.

**Definition 6 (Data Dependence Graph).** The data dependence graph $D$ for a given TLM, $\mathcal{T}$, is a directed graph with vertices $(v,s,T)$ where $v,s \in \mathcal{T}$ and $T \in \mathcal{T}$ and $v$ is updated in $s$ in $T$. An edge $((v_i,s_j,T_k),(v_l,s_m,T_n)) \in D$ exists if and only if $v_l$ in $s_m$ in $T_n$ is data-dependent on $v_i$ in $s_j$ in $T_k$.

In Fig. 3.2, $v_l$ in $s_m$ is data-dependent on $v_i$ in $s_j$. Note that the above definition is not constructive, i.e., it does not provide an algorithm for determining data dependence. That is beyond the scope of this thesis[1]. In the above, $T_k$ and $T_n$ may be the same transaction and $v_l$ is updated by a transaction instance subsequent to when $v_i$ is updated in a previous instance. This form of data dependence is referred to as a cross-instance data dependence.

---

[1]: available in the compiler literature [65]
Different interaction patterns within and across transactions can now be defined in terms of data dependence and the data dependence graph.

**Definition 7** (Transaction Independence). Transaction $T_n$ is independent of transaction $T_k$ iff for each $(v_i, s_j, T_k)$ there is no path to any $(v_l, s_m, T_n)$ in $D$.

Thus, for validating $T_n$, $T_k$ is immaterial. This happens often in firmware transactions. Many firmware transactions are independent of the hardware transactions they interact with when they are only responsible for setting the state of the hardware transactions.

**Definition 8** (Producer Transaction). Let $U = (v, s_j, T)$ be the set of all vertices in $D$ with a write to $v$ in $T$. Further, for each $u \in U$, the only vertices in $D$ with paths to $u$ are in $T$. In this case $T$ produces $v$.

Intuitively, in this scenario, $T$ produces data that is independent of other transactions. Thus, we do not need to consider the interleaving of other transactions with $T$ in computing this data. In Fig. 3.2, the variable $v_i$ is only written in $s_j$, and the only edge to $s_j$ is from $s_k$, which is within $T_k$. Thus, $T_k$ produces $v_i$.

**Definition 9** (Producer Consumer Interaction). Let $((v_i, s_j, T_k), (v_l, s_m, T_n))$ be an edge in $D$ and let $T_k$ produce $v_i$. In this case, $T_n$ is said to consume $v_i$. 

54
In Fig. 3.2, $T_k$ is the producer and $T_n$ is the consumer of $v_i$. Note that a producer-consumer interaction is defined specific to the value produced. Thus, it may be possible for $T_1$ to produce $x$ and consume $y$ and $T_2$ to produce $y$ and consume $x$. Thus, neither of the transactions in a producer consumer relationship may be independent of the other.

**Definition 10 (Stateful Transactions).** A transaction, $T$, is said to be stateful iff (1) there is a cross-instance edge $((v_i, s_j, T), (v_l, s_m, T)) \in D$ or (2) there is a path from $(v_i, s_j, T)$ to $(v_l, s_m, T)$ that goes through some vertex $(v_p, s_q, T_r)$ and $T \neq T_r$.

Intuitively, the first case in this definition corresponds to $v_l$ using a value written to $v_i$ in a previous instance of $T$. In the second case, since $v_p$ is from some other transaction, depending on the relative speeds of the execution of individual instances, $v_l$ can use a value that is written to $v_i$ in a previous instance of $T$. In Fig. 3.3, both $T$ and $T_r$ are stateful. In a stateful transaction, an instance of the transaction depends on previous instances and provides values for future instances. Thus, multiple transaction instances need to be unrolled to cover all possible scenarios for this transaction. For example, assuming the initial states of both $a$ and $b$ are zero, without unrolling $T$ and $T_r$ multiple times, the condition $(a > 5)$ will never be explored.

A transaction is said to be stateless if it is not stateful. Intuitively, the values produced by a transaction instance of a stateless transaction do not depend on any previous instance of the transaction. Hence, in a stateless transaction, only one unrolling of the transaction is needed to cover all possible scenarios for that transaction. This is quite common with a service function-based TLM. Each instance of a service function for the firmware is in response to some input from the application/OS software or from the hardware. Similarly, each instance of a service function for the hardware is in response to an input from the firmware or the physical environment. These responses are typically independent of previous invocations of this service function. In the next section, we examine the practical prevalence of the statelessness and the producer-consumer relationship.
3.3.3 Prevalence of the interaction patterns in practice

We developed the Rockbox TLM model using the service function based TLMs proposed in this thesis. It consists of a set of firmware transactions and a set of hardware transactions that the firmware transactions depend on in SystemC. For the firmware, we used reverse engineering to model the TLM specification out of the C/assembly open-source firmware implementation. The hardware transactions were written by manual analysis of the relevant datasheets \[27, 84, 88\]. We modeled six firmware and three hardware transactions (Table 3.1) and observed the model’s interaction patterns (Table 3.2). To show that these patterns are not only observed in the specific Rockbox case but are meaningful in general cases, we studied the interaction patterns of several Linux device drivers \[23\] and the corresponding x86 QEMU device emulator code \[70\] as well. Table 3.3 shows the different types of peripherals emulated in x86 QEMU and the number of specific devices emulated for each category. (Here, we selected some important representative categories from QEMU. The complete list has over 53 devices and more than 91961 lines of code, and thus a full analysis of QEMU interaction patterns is beyond the scope of this thesis.) We chose one device per category and analyzed its interaction patterns with the corresponding Linux device driver. The rest of this section explains several interaction patterns that are common in the examples.

Table 3.1: Rockbox transactions and their functionality (FW: Firmware, HW: Hardware)

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Type</th>
<th>Main Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>FW</td>
<td>Monitor charger connection and battery level</td>
</tr>
<tr>
<td>USB</td>
<td>FW</td>
<td>Monitoring USB status</td>
</tr>
<tr>
<td>Backlight</td>
<td>FW</td>
<td>Manage to turn on/off the LCD backlight</td>
</tr>
<tr>
<td>ATA</td>
<td>FW</td>
<td>Stop or restart the ATA bus depending on the USB status</td>
</tr>
<tr>
<td>Button</td>
<td>FW</td>
<td>Post the button value to other threads</td>
</tr>
<tr>
<td>Charger stat</td>
<td>HW</td>
<td>Provide the physical charger status [84]</td>
</tr>
<tr>
<td>Battery level</td>
<td>HW</td>
<td>Convert the analog battery level to digital [84]</td>
</tr>
<tr>
<td>USB detect</td>
<td>HW</td>
<td>Detect the physical USB presence [27]</td>
</tr>
<tr>
<td>ATA transfer</td>
<td>FW</td>
<td>Transfer data from/to the disk</td>
</tr>
</tbody>
</table>
Table 3.2: Relationship of the interacting Rockbox transactions (p-c means producer-consumer)

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Shared variable</th>
<th>Related Transaction</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>batvolts, GPI56, usb_state</td>
<td>Battery level, Charger status, USB</td>
<td>Both stateless &amp; p-c, Both stateless &amp; p-c, Both stateless &amp; p-c</td>
</tr>
<tr>
<td>USB</td>
<td>id</td>
<td>USB detect, Backlight</td>
<td>Both stateless &amp; p-c, Both stateless &amp; p-c, Both stateless &amp; p-c</td>
</tr>
<tr>
<td>Backlight</td>
<td>remote_hold_button</td>
<td>Button</td>
<td>Stateless-stateful &amp; p-c</td>
</tr>
<tr>
<td>ATA</td>
<td>sleeping, spinup</td>
<td>ATA transfer</td>
<td>Not p-c, Both stateless &amp; p-c</td>
</tr>
<tr>
<td>Button</td>
<td>-</td>
<td>-</td>
<td>Stateful</td>
</tr>
</tbody>
</table>

Table 3.3: Linux-QEMU interacting transaction pairs (p-c: producer-consumer, SL: stateless, SF: stateful)

<table>
<thead>
<tr>
<th>Category</th>
<th>Num of devices supported</th>
<th>Representative device</th>
<th>Linux Transaction</th>
<th>Related QEMU Transaction</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage</td>
<td>5</td>
<td>ATAPI IDE</td>
<td>ide_transfer_pc(SF)</td>
<td>cmd_read(SL)</td>
<td>p-c</td>
</tr>
<tr>
<td>Graphics</td>
<td>2</td>
<td>Cirrus CLGD 54xx VGA</td>
<td>cirrusfb_bitBLT(SL)</td>
<td>cirrus_vga_iowrite(SF)</td>
<td>not p-c</td>
</tr>
<tr>
<td>Network</td>
<td>5</td>
<td>NE2000 ethernet card</td>
<td>block_input(SL), block_output(SL)</td>
<td>ne2000_iowrite(SL), ne2000_ioread(SL)</td>
<td>p-c</td>
</tr>
<tr>
<td>Port</td>
<td>2</td>
<td>16550A UART</td>
<td>rx_chars(SL), tx_chars(SF)</td>
<td>serial_iowrite(SF), serial_ioread(SL)</td>
<td>not p-c</td>
</tr>
<tr>
<td>Bus</td>
<td>13</td>
<td>USB UHCI</td>
<td>uhci_readw (SL), uhci_writew(SL)</td>
<td>uhci_iowritew(SF), uhci_ioreadw(SL)</td>
<td>not p-c</td>
</tr>
<tr>
<td>Input</td>
<td>4</td>
<td>Microsoft serial mouse</td>
<td>sermouse_interrupt(SF)</td>
<td>msemouse_event(SL)</td>
<td>p-c</td>
</tr>
<tr>
<td>Sound</td>
<td>4</td>
<td>Intel 82801AA</td>
<td>codec_read(SL), codec_write(SL)</td>
<td>nabm_read(SL), nabm_write(SF)</td>
<td>p-c</td>
</tr>
<tr>
<td>Clock</td>
<td>2</td>
<td>i8253 PIT</td>
<td>i8253_read (SF)</td>
<td>pit_iowrite(SF), pit_ioread(SF)</td>
<td>p-c</td>
</tr>
<tr>
<td>Interrupt handler</td>
<td>3</td>
<td>i8259 interrupt handler</td>
<td>irq_pending(SL), mask_irq(SL), unmask_irq(SL)</td>
<td>pic_iowrite (SF), pic_ioread(SL)</td>
<td>p-c</td>
</tr>
</tbody>
</table>
**Independence:** Often the transaction interaction is one-way, i.e., one is independent of the other when the two transactions are interacting. For example, in the Rockbox backlight transaction (Table 3.1), the function setting the backlight brightness only writes to the hardware device but does not read from it. Thus, the backlight thread is independent of the LCD device. Also, the battery level transaction in Table 3.1 the hardware transaction which provides battery value, only writes the shared register for the firmware transaction (power) but does not read from the power transaction. Thus, the battery-level transaction is independent of the power transaction. Six of seven Rockbox transactions (third column of Table 3.2) are independent of the interacting firmware transactions (first column). For QEMU, seven of the 15 Linux transactions are independent of the QEMU device transactions.

**Producer-Consumer Relationship:** The most common case in both Rockbox and Linux-QEMU is that a shared variable is shared only between one firmware and one hardware transaction, or one firmware and another firmware transaction, and this sharing has the producer-consumer interaction pattern. In these cases, the shared variables are I/O pins or registers in a device that are shared with a dedicated device driver, or global variables across the firmware transactions. This pattern is easy to find in embedded systems since many I/O pins or registers are dedicated to a specific purpose and written by a master module. In fact, six of seven pairs of interacting transactions in the Rockbox and six of nine devices in Linux-QEMU have a producer-consumer relationship. In Table 3.1 the power transaction has three producers, battery level, charger status, and USB, but shares different variables with each producer. The USB transaction shares a variable with two producers, and the ATA transaction shares a variable with a producer.

**Statelessness:** As shown in Table 3.2 and 3.3 stateless transactions occur frequently since a firmware transaction instance tends to be interested in only the current status of the device and hence processes a fresh set of device input values in each instance. For example, the power thread reads the current battery level, which is naturally stateless because the past
battery level does not decide the current value. In Table 3.1, eight of nine transactions are stateless, and 19 of 30 Linux or QEMU transactions are stateless.

3.4 Using transaction interaction patterns in firmware testing

This section shows how the transaction interaction patterns presented in the previous section can be exploited to simplify test generation as well as increase test coverage. We show how the interaction patterns can be analyzed, and the results of the analysis can be encoded as constraints to be used during concolic test generation. This enables using a single-threaded test generator such as KLEE to generate tests for a firmware transaction while accounting for other interleaving hardware and firmware transactions.

3.4.1 Automatic test generation using KLEE

Concolic testing is a technique which enables automated test generation. We start with a brief overview of concolic testing and an explanation of how the KLEE concolic testing tool works.

Concolic Testing: Concolic testing [72] combines symbolic execution [64] with concrete execution. It uses a constraint solver with the symbolic execution to cover all feasible paths related to the given symbolic variables. The user is responsible for identifying a set of variables as symbolic and instrumenting her test code. Then the concolic testing tool executes the code concretely except for branches controlled by the symbolic variables. For these branches, the tool explores both paths while attaching the corresponding constraints of the symbolic inputs for each path. At the end of each path, the tool asks its solver to
```c
void main(void)
{
    unsigned int i, ret;
    make_symbolic(&i);
    if (i > 0) ret = i;
    else ret = 1/i;
}
```

Code 3.1: An example instrumented code for KLEE

KLEE is built on top of the LLVM compiler. It generates test cases automatically while detecting errors at dangerous operations, such as division by zero. Code 3.1 is an instrumented test code for KLEE with a symbolic variable \( i \). KLEE executes the code as usual until it hits a symbolic value (line 4). Then, it forks a new execution process and adds the constraint \( i > 0 \) on the true path and \( i \leq 0 \) on the false path (Fig. 3.4). The two paths are executed in the two distinct processes. The true path ends after line 4, and the solver picks a random value, say 10, for \( i \) among those values satisfying \( i > 0 \). On the false path with the path constraint \( i \leq 0 \), KLEE continues execution, eventually reaches line 5, and detects a dangerous operation, division by zero. Now, KLEE adds \( i == 0 \) as a constraint to the solver, and generates the error case. KLEE continues the other path, \( i \leq 0 \land i \neq 0 \), and generates the test case \( i == -10 \) at the end. As shown, KLEE explores all feasible paths for this case and generates the interesting test cases.

**Concolic Testing and Concurrency:** Concolic testing has been used largely for unit testing single-threaded code, not a multi-threaded program. The survey paper [87] on concolic testing tools indicates that only jCUTE [96] supports concurrent programs, but it is not open-source. There have been approaches to extend concolic testing to multi-threaded programs [91 95]. These involve searching for all possible interleavings (or equivalently,
schedules) of concurrent threads. However, due to the large number of possible interleavings, this tends not to scale. The partial order reduction (POR) \[47\] technique can help mitigate this. It generates only one representative interleaving among all interleavings, which result in the same state. However, even with POR, the number of interleavings can still be very large. In this thesis, we show how the transaction interaction patterns can be used to address this issue.

3.4.2 Test case generation for stateless producer-consumer transactions

As seen in the previous section, stateless transactions and the producer-consumer interaction pattern are prevalent in our benchmark transactions. Also, as discussed, if the firmware is the producer, the firmware transaction is independent of the hardware transaction. Hence, the most interesting case is when the firmware transaction is the consumer. Here, what/when the hardware transaction produces can impact the paths that the firmware transaction takes, and thus impact the test cases of the firmware. We now show how the interactions with the hardware transaction can be modeled through additional constraints provided to the constraint solver and used for test case generation of the firmware transaction. The overall procedure is as follows. The transactions to be tested are minimally instrumented manually for two purposes. The first is to identify certain variables as symbolic. These are the variables that need to be explored to cover various program paths in the firmware. This is standard in concolic testing. The second is to add constraints that capture the producer-consumer interactions among the transactions. This is specific to our methodology. As we will show, these constraints completely eliminate the need to consider interleavings for this interaction pattern.

**Example Transaction:** We will illustrate the procedure using a simple example that captures the essence of the procedure. This example focuses on two different producer-
consumer interactions, each with one shared variable, between two stateless transactions. However, the procedure can extend to multiple stateless producers sharing multiple variables with multiple consumers.

In Code 3.2 and 3.3, $T_1$ is a producer for $x$ and a consumer for $y$ at the same time. $T_2$, on the other hand, consumes $x$ and produces $y$. Assume that the initial values of $x$ and $y$ are 0 and 1, respectively. Note that $T_1$ is executed as follows: while(1){ $T_1()$; }. $T_2$ is executed in a similar infinite while loop as well, and the executions of $T_1$ and $T_2$ are in parallel. In Code 3.2, the symbolic variable $rand$ is used to take either one of the paths in each execution. If $result$ is 0, it is considered to be an error. While these transactions, as constructed, are not computationally very meaningful, their structure has been selected to illustrate the key points of the test generation procedure while keeping the transactions themselves small. By manual analysis, we can easily conclude that in the concurrent execution of $T_1$ and $T_2$, each of $r_1$ and $r_2$ could independently store 1, 2, or 3. The order of write does not matter. $r_1 = 3, r_2 = 2$ is possible in an instance of $T_2$, since a new instance of $T_1$ keeps getting executed and $T_2$ can read any of the values produced by the new instance of $T_1$. Note that $r_1 = 0$ or $r_1 = r_2 = 0$ is also possible in case $T_2$ executes first before $T_1$ ever executes, as the initial value of $x$ is zero. Accordingly, we see that the condition in line 5 in Code 3.3 will never be true. This means $T_1$ will not read 0 from $y$. It is important that the transaction instances do not have to be unrolled for testing since $T_1$ and $T_2$ are stateless. We now show how the results of this analysis can be captured through instrumenting the code in KLEE.

**Instrumenting the input code:** The overall goal of instrumenting the producer is tracking the different values that could be produced by the producer for use by its consumer. In the instrumented version, $T_1$, the producer for $x$, takes an integer pointer $wx$ as an argument in Code 3.4 ($wx$ is short for writex). The value of $x$ updated by $T_1$ will be stored in $wx$ later, so that the test generator code (Code 3.5), which calls $T_1$, can access this value. In
```c
void T1() {
    int rand, result;
    klee_make_symbolic(&rand);
    if (rand < 0) {
        x = 1;
    } else {
        x = 2;
        x = 3;
    }
    result = y;
    assert(result != 0);
}
```

```c
void T2() {
    int r1, r2;
    r1 = x;
    r2 = x;
    if (r1 < 0 || r2 < 0) {
        y = 0; // error
    } else {
        y = 7;
    }
}
```

Code 3.2: Producer for \( x \) / consumer for \( y \)

Code 3.3: Consumer for \( x \) / producer for \( y \)

LINE 1, \( w[NUM\_WRITE] \) saves all values of \( x \) written by \( T1 \) during the execution of an instance. \( write\_len \) tracks the number of multiple writes on \( x \) during the instance, and it can be either 1 or 2 at the end of a path taken. Lines 12-14 let KLEE pick a constrained-random number between 0 to \( write\_len - 1 \), to pick one of the written values during the path. Thus, \( w[c] \) can be either 1, 2, or 3. In other words, every time \( T1 \) is called, \( w_x \) will record one of the three values. Note that, in general, the producer transaction may have many possible paths, and each of these paths may have many possible writes to the shared variable. With this instrumentation, for each path, we gather only one of the written values during the path. This is enough because later on, our test generator code runs the producer as many times as the maximum number of writes on its shared variables (detailed explanation is provided later). This will generate all possible write values since KLEE tries to explore all possible values for the shared variables.

Similarly, \( T2 \) (Code 3.4), the producer of \( y \), is instrumented in the same way in terms of \( y \) (line 22-23). We directly store the written value in \( w_y \) rather than introducing \( write\_len \) and \( c \) here because \( T2 \) only writes \( y \) once along any of the feasible paths.

In the above instrumentation, \( w[] \) stores all possible values written to the shared variable. As we saw in Code 3.2, this set of values, not the order in which they were written, is what
Code 3.4: Instrumented T1

matters in terms of interaction with the consumer. The consumer can read any one of these values with every read. Also, the size of $w[]$ is known statically in this case. If this is not the case, then $w[]$ can be dynamically allocated.

On the other hand, in the instrumented consumer Code 3.4, the consumer of $x$, $T_2$, takes integer pointers $rx_1$ and $rx_2$ as arguments for every read of $x$ in the transaction ($rx$ is short for read$x$). The pointers $rx_1$ and $rx_2$ are to store the values that the transaction reads in order to deliver them to the test generator code. Every time the transaction is supposed to read from $x$, the value is instead read from a symbolic variable, e.g., $temp_1$ or $temp_2$ (line 19, 21). This forces the read value to explore all possible values as it is symbolic. Later, as shown in the next section, this symbolic value will be constrained to the possible written values of $x$ gathered from the instrumented producer $T_1$. 
int main (){
    int st.x = 0; // initial state of x
    int num_st;
    int wx1, wx2, wx3, wy; // pointers for writes
    int rx[2]; // pointers for read x
    int ry1, ry2, ry3; // pointers for read y
    int read_len = 2;
    T1(&wx1, &ry1);
    T1(&wx2, &ry2);
    T1(&wx3, &ry3);
    T2(&rx[0], &rx[1], &wy);
    klee_make_symbolic(&num_st);
    klee_assume(0 <= num_st <= read_len);
    int i;
    for (i=0; i<num_st; i++)
        if (rx[i] != st.x) klee_silent_exit(0);
    for (i=num_st; i<read_len; i++)
        if (rx[i] == wx1 && rx[i] == wx2 && rx[i] == wx3) klee_silent_exit(0);
    if (ry1 != wy) klee_silent_exit(0);
    if (ry2 != wy) klee_silent_exit(0);
    if (ry3 != wy) klee_silent_exit(0);
}

Code 3.5: The test generator code

The Test Generator: In Code 3.5, the test generator code combines $T_1$ and $T_2$ into a single thread code and adds some constraints that connect the values shared by the producer and consumer. This code first executes $T_1$, the producer of $x$, as many times as the total number of writes to $x$ in $T_1$ (line 8-10). This allows KLEE to assign independent values for each write. From each call of $T_1$, it collects one of the write values from the producer in $wx_1$, $wx_2$, and $wx_3$. Since constrained-random values are picked by KLEE, if the algorithm calls $T_1$ as many as the number of total writes, it can collect all possible write values from the producer.

Next, the consumer of $x$, $T_2$, is called in line 11. Now, what we want is to make $T_2$ read one of the collected values written by $T_1$. In line 18, if $rx[i]$, one of the read values, is not the same as one of the collected values of $x$, KLEE exits without generating any case for that path. This forces KLEE to select one of the collected values. Also, the case that $T_1$ is not even executed before $T_2$ executes is covered with $T_2$ reading from the initial state of $x$. KLEE picks the number of how many times the consumer reads from the start state,
num_st, again randomly but constrained by the total number of reads (line 12, 13), and forces \(rx[0]\) to \(rx[num_st - 1]\) to read from the start state as in line 16.

Similarly, the written value of \(y\) was collected from \(T2\) in line 11, and the read values from \(y\) (\(ry1\), \(ry2\), and \(ry3\)) are forced to be the same as the collected \(y\) value in line 19-21. Note that we omitted the start state of \(y\) for brevity. Also, we only ran \(T2\) once since we know there is only one write to \(y\) during any path of \(T2\) and there is only one read from \(T1\). In general, the producer transaction is run as many as the maximum number of each shared variable’s total number of writes from its producer, so that all the reads can get independent write values.

Note that there are multiple paths in the consumer of \(x\), \(T2\). As the values written by the producer \(T1\) are stored in the symbolic variables \(wx1\), \(wx2\), and \(wx3\), *KLEE will try and find all possible values that can be written to \(wx1\), \(wx2\), or \(wx3\) in order to exercise all the paths in the consumer.* It may be possible that there is no value for \(wx1\), \(wx2\), or \(wx3\) that can exercise some path, in that case no test case will be generated for that path. In this example, since \(T1\) only produces 1, 2, or 3, for \(x\) of which the initial value is 0, the true branch path in \(T2\) can never be exercised. In contrast, if the consumer was unit tested alone, then this path would be exercised by assigning some value to \(x\), even though that value could never be produced by the producer. This path would be classified as a bug, but it would be a false positive as this path cannot be exercised in the full system context. False positives due to under-constrained environments are a key problem in software testing, and eliminating them is important. This is the key value of this integrated testing procedure. *Note also that the service function aspect of the transactions was key to enabling the characterization of the transaction interaction pattern as stateless producer-consumer, which was critical to completely capture the space of shared values needed for the test case generation.*

**General Algorithm:** Code 3.6 is the general algorithm for generating test cases of stateless consumer transaction with multiple producer transactions. Here, \(T1\) is the target firmware
testGenerator($T_1,\ldots,T_n$) {
  for each $T_i$ where $1 \leq i \leq n$ {
    for ($x = 0$; $x < \max(nw_{ijk} \forall j,k)$; $x++$)
      $T_i(\ldots&wv_{ijkx},\ldots&rv_{ijklx},\ldots);$ 
  }
  for each $rv_{ijkx}$ {
    klee_make_symbolic(&num_st_{ijkx});
    klee_assume(0 $\leq$ num_st_{ijkx} $\leq$ read_len_{ijkx});
    for ($l = 0$; $l < read_len_{ijkx}$; $l++$) {
      if ($rv_{ijklx} = INIT_{ijk}$) klee_silent_exit(0);
    }
    for ($l = num_st_{ijkx}$; $j < read_len_{ijkx}$; $j++$) {
      if ($rv_{ijklx} = wv_{ijk1}$ & $rv_{ijklx} = wv_{ijk2}$ & $\ldots rv_{ijklx} = wv_{ijk\max(nw_{ijk}\forall j,k)}$) 
        klee_silent_exit(0);
    }
  }
}

Code 3.6: The general test generator algorithm

transaction, and $T_2,\ldots,T_n$ are the producer transactions of $T_1$, where each transaction is instrumented as explained in the previous section. $v_{ijk}$ represents the $k$-th variable among the shared variables produced by $T_i$ and consumed by $T_j$. We use fresh variables for multiple readings on the same variable, i.e., $rv_{ijk1}$ and $rv_{ijk2}$ are multiple readings (r stands for read) on $v_{ijk}$. In contrast, we gather only one value of $v_{ijk}$ written by $T_i$ multiple times during an instance, and save into $wv_{ijk}$. Note that we assume that static analysis of the data dependence graph is already able to determine the interaction patterns. For example, $nw_{ijk}$, the total number of writes on $v_{ijk}$ in all paths of its producer $T_i$, or $read_{len}_{ijk}$, how many times $v_{ijk}$ is read during an instance of its consumer, is given.

Lines 2-5 in the Code 3.6 runs all transactions including $T_1$ and its producers. Each runs as many times as the maximum number of writes on its shared variables. This forces generation of all possible write values since KLEE tries to assign different values as much as possible for the $wv_{ijkx}$ of each instance ($x$ represents each instance). Lines 7-8 decide among multiple reads from the same variable $v_{ijk}$ during an instance, how many read from
its initial states, not from its producer. Lines 9-11 force the variables to have the corresponding initial values as decided. Line 10 drives KLEE to generate results only satisfying desired conditions by killing the cases satisfying the negative conditions. Here, $INIT_{ijk}$ is the initial value of $v_{ijk}$. The rest of the code matches values of all $r_{ijk}$ and $w_{ijk}$. A fresh variable $r_{v_{ijklx}}$ can read from any values of $w_{v_{jikx}}$ generated from all producer instances.

This algorithm forms the template which can be automatically instantiated with specific instances. To automate the algorithm, tracking accesses to the shared variables in both producer and consumer code and instrumenting the code to store the set of values are needed.

To sum up, the algorithm captures the effect of the producers on the target transaction as a set of constraints instead of explicitly exploring all their interleavings. The algorithm executes the producer transactions as many times as they can generate all possible values for the shared variables, thus the target transaction can consume these values where each case corresponds to a feasible path. This algorithm guarantees the complete coverage of the consumer since it collects all values of shared variables written by the producers. Also, the testing result is sound since we constrain the consumer with only realizable values from the producers. Lastly, in case only a subset of the interaction patterns among transactions have the stateless producer-consumer relationship, this algorithm can be used to partially test the system.

### 3.4.3 Limitations

This chapter focuses on the stateless-producer-consumer pattern only. However, the procedure presented here shows the possibility of a scalable and complete sequentialization method while avoiding explicitly exploring the interleavings. In the following chapters, this procedure is extended to other combinations of the interaction patterns. This also adds complexity as it will require exploring some transaction interleavings and unrollings.
Further, determining the interaction patterns and the set of readers and writers in the in-
teracting transactions required data dependence analysis. Another limitation is the manual
instrumentation process. For our Rockbox study we were able to analyze data dependence
and instrument the example code manually. Automated processes to (1) determine the
interaction patterns and (2) generate the sequentilization code are presented in the next
chapter.

3.4.4 Experimental results

We applied the proposed procedure to the set of the producer-consumer stateless transac-
tions listed in Table 3.2 For fair experiments, we only tried to extract a unit of work, or
a transaction, from Rockbox but did not change its code and functionality. (In this work,
the physical input values, such as physical battery values, are made symbolic. Also, since
KLEE only allows C input, we made a C version of the Rockbox transactions.) For com-
parison, we performed unit testing using KLEE on the consumer transaction by itself first.
For instance, unit testing of the Power transaction in Table 3.4 resulted in 800 test cases.
For unit testing, initial values of the inputs and the shared/local variables were made sym-
bolic. The results of the unit testing on consumers are shown in the third column of the
Table 3.4.

Next, we performed the testing with the producer transactions. The Power transaction
has three producer transactions; three shared variables constrained randomly in the unit
testing are now to be constrained by the producers. We gradually introduced the producers
one by one to see the effect of introducing these producers. For example, we tested the
Power transaction only with the battery-level transaction first while still making the shared
variables with charger status and USB transaction symbolic. As a result, we got 404 cases.
Next, we tested the Power transaction (consumer) and the two producers, battery level and
charger status, together and got the same 404 cases. Finally, we tested with all three of
Table 3.4: Test cases of Rockbox transactions

<table>
<thead>
<tr>
<th>Consumer</th>
<th>LOC (Original/Instrumented)</th>
<th># of test cases of unit testing</th>
<th>Run time (sec)</th>
<th># of symbolic variables</th>
<th>Producer</th>
<th>LOC (Original/Instrumented)</th>
<th># of test cases with the producers</th>
<th>Run time (sec)</th>
<th># of symbolic variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>Power</td>
<td>973/991</td>
<td>800</td>
<td>302.56</td>
<td>8</td>
<td>Battery level</td>
<td>89/105</td>
<td>404</td>
<td>241.00</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Charger status</td>
<td>129/141</td>
<td>404</td>
<td>252.71</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>USB</td>
<td>807/824</td>
<td>128</td>
<td>39.82</td>
<td>3</td>
</tr>
<tr>
<td>USB</td>
<td>807/821</td>
<td>18</td>
<td>1.86</td>
<td>6</td>
<td>USB detect</td>
<td>27/30</td>
<td>14</td>
<td>1.97</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Backlight</td>
<td>1003/1017</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ATA</td>
<td>1538/1568</td>
<td>14</td>
<td>1.78</td>
<td>7</td>
<td>ATA transfer</td>
<td>226/243</td>
<td>19</td>
<td>2.36</td>
<td>3</td>
</tr>
</tbody>
</table>

the producers and got 128 cases. By adding the producers, the test cases decrease since the producers restrict the possible values of shared variables more accurately than how the concolic testing tool constrained them during unit testing. The remaining 404-128 (= 276) cases correspond to paths that can never be exercised in the full system context and these may lead to false positives if only unit testing of the firmware was used. For ATA, the number of test cases increased slightly when we tested with the producer transaction. This is because some of the values written by the producer are read multiple times along the same path in the consumer. With the producer, these values can be distinct. However, when unit-testing with KLEE alone, they will all be the same as KLEE does not consider the volatility of the shared variable. Thus, in this case, integrating the producer and the consumer during test generation provides for increased coverage.

### 3.5 Related work

This section presents the TLM modeling frameworks and concurrent program testing approaches most related to ours.

#### 3.5.1 Transaction-level models

Perhaps closest in spirit to our TLM is the microarchitecture-level TLM by Mahajan et al. [71]. They provide an intrinsic higher-level modeling framework for the functional al-
algorithms to be implemented by the RTL hardware models. From this data-centric model, they enable RTL synthesis and verification of the models across levels. A transaction is a unit of work performing data computation with clearly defined functionalities in this paper. This shares the explicit start and end characteristics of our service function-based transactions. However, their model is suitable for hardware rather than firmware/software as their detailed concurrency semantics match the underlying hardware components. For example, they allow multiple instances of a transaction executed at the same time for pipelining unlike our TLMs.

3.5.2 Concolic testing for concurrent programs

There have been various efforts to extend concolic testing to support concurrent programs. Large-sized real-world programs are often concurrent. Testing these programs is notoriously hard due to their exponential number of interleavings. CUTE [96] and jCUTE [94] are concolic test engines for C and Java, respectively. In these testing tools, they combined concolic execution with a variant of dynamic partial order reduction in order to systematically generate test inputs to handle multi-threaded programs [95]. They determine the exact (data and lock) race conditions between concurrent events. Then, they permute the events to explore only one representative schedule. These tools showed their practical aspects with experiments on several open-source software programs, including the java.util library of Sun JDK 1.4. The combination of partial order reduction and concolic testing provides one of the first techniques to test complex concurrent programs efficiently. Rungta et al. [91] takes possible error states of the target concurrent system as an input, and uses guided symbolic execution to check reachability of the error states with continuous refinements.

In contrast, our work directly uses the interaction patterns to add constraints to a single-threaded concolic testing program to capture the effect of all interleavings. While the existing concolic testing tools have already been effectively used to test various (mainly
single-threaded) software programs, more research is still needed to scale symbolic execution to concurrent programs. Also related is the work on formal verification of device drivers [4, 86] but that is beyond the testing focus of this chapter.

3.6 Chapter summary

This chapter introduces a novel TLM for modeling both the firmware and hardware components where the transactions correspond to service functions. The notion of service functions provides an intuitive way of modeling firmware and hardware functionalities at the specification level. Moreover, the TLM naturally captures specific firmware-hardware interactions. This chapter provides in-depth analysis of several common interaction patterns that are observed in real-world firmware examples. In particular, we focus on the stateless producer-consumer interaction pattern. We then show how single-threaded concolic testing generation tools, such as KLEE, can be used to generate tests for firmware with this interaction pattern. This test generation can potentially generate a complete test set with no false positives. We demonstrate the applicability of this modeling and testing methodology through a non-trivial practical case study of the Rockbox MP3 player. We model three hardware and six firmware transactions, characterize their interaction patterns, and generate complete test cases for the six pairs that have the stateless producer-consumer interaction pattern.

While the test generation part of this chapter focused on the stateless producer-consumer case, it underlines a broader principle that can be brought to bear to simplify test generation for concurrent execution. The main idea exploited here is the higher-level information captured by the service function transaction model, which enabled static code analysis to determine the various interaction patterns. The next step in this research is to explore the cases that, unlike the clean stateless producer consumer case, require exploration of
transaction unrolling and interleaving. Further analysis of the data dependences for these cases can help bound the unrolling/interleavings needed. Also, analyzing the interaction patterns between transactions and instrumenting the target programs to generate a single-threaded code are manually done in this chapter. Hence, the following chapter expands this work to provide an automated testing method with a high coverage of the complex interaction patterns.
Chapter 4

Automated Firmware Testing using Firmware-Hardware Interaction Patterns

4.1 Introduction

Chapter 3 described how to achieve a complete and scalable testing methodology for one particular transaction interaction pattern (e.g., the stateless producer-consumer case). 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 Chapter 3 - which significantly limits its practical utility.

This chapter addresses both these limitations. It considers both stateful and stateless transactions, and provides 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. This chapter shows how this coverage can be determined for a large space of interaction patterns. Conversely the testing method 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 hardware/firmware transactions to generate a custom sequential program \( P \) that considers the transaction interactions. \( P \) is test-equivalent to the target firmware code \( F \), i.e., (i) the set of tests of \( P \) can be directly applied to \( F \) and (ii) the test coverage for \( F \) can be determined using the test coverage for \( P \).

The test-generation methodology in this chapter is fully automated. The testing framework is built using KLEE [11], a concolic testing tool, and Frama-C [26], a static software analyzer. The practical applicability of this framework is tested on Linux device driver code and the interacting QEMU code. QEMU is the hardware emulator code that includes a wide range of devices. Horn et al. [57] have generated two large usable benchmarks by extracting several device models that can run in standalone mode by eliminating the complex dependencies in QEMU. This chapter demonstrates the efficacy of our method for 15 transactions in these benchmarks.

This chapter is organized as follows:

- Section 4.2 introduces a running TLM example.
- Sections 4.3, 4.4 present frameworks for the interaction-pattern-specific customized algorithms to generate test cases for a firmware transaction interacting with concurrent firmware/hardware transactions using a single-threaded concolic testing tool. The algorithms introduced here use the interaction patterns to avoid exploring asynchronous interleavings of the concurrent transactions. Also, the 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.

- Section 4.5 demonstrates the automatic detection of the specific interaction patterns of interest for the transactions using a static analyzer based on Frama-C. This section also describes the code generator that uses these interaction patterns along with the firmware and hardware transactions to generate a test-equivalent sequential program. This sequential program can then be used with a standard sequential concolic testing tool. Lastly, this section describes how the concolic test generator KLEE is modified to provide code coverage in this context.

- Section 4.6 demonstrates the efficacy of the testing process on 15 transactions from two published real firmware benchmarks, TMP105 and Ethoc, from the Linux-QEMU platform. Lastly, this section highlights a bug discovered by this testing process in a published benchmark.

4.2 A running example of the service-function based TLMs

This section introduces a Rockbox [1] TLM example, the running example throughout this chapter.

Fig. 4.1 represents a simplified version of the Rockbox [1] iPod serial protocol transaction in the form of an HLSM. In Fig. 4.1, an instance of the SERIAL0 transaction initiates, and at $f_a$, it checks if $rx$ is ready. If it is ready, then it takes the true path (right branch) and goes to the $f_b$ state. Here, $f_b$ 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 $f_k$. Hence, the path that an instance takes is determined by the current data state and
the state transition function. After this instance is finished at end state, the next instance
can begin. 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. 4.1 the SERIAL0 and serial_bitrate transactions
share the autobaud variable. serial_bitrate writes the variable in gb and gc, and SERIAL0
reads the variable in fb and fc.

Figure 4.1: Rockbox SERIAL0 firmware transaction and its producer firmware transaction
serial_bitrate
4.2.1 Transaction interaction patterns occur in the running example

This section explains the interaction patterns that occurred in the running example.

Stateless vs. stateful transactions

The serial_bitrate transaction (Fig. 4.1) is stateless. \( g_a \) 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 \( g_a \), one can possibly exercise either path of the transaction in the first iteration. To consider the effect of the serial_bitrate transaction 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. 4.1, if an instance of SERIAL0 reaches \( f_h \), it increases the stateful variable 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 \( f_j \) and sets the badbaud value to zero. The next instance may read this new value.

Producer and consumer transactions

Both SERIAL0 and serial_bitrate are producers for autobaud in Fig. 4.1. SERIAL0 is a consumer for autobaud.

4.2.2 Transaction interaction patterns and test generation: Previous work and its limitations

This section briefly explains how the previous chapter efficiently sequentialized concurrent stateless transactions for the purpose of testing.
Avoiding all interleavings

In general, testing concurrent threads needs full exploration of all the interleavings to consider all possible paths through code. However, the firmware testing method based on the proposed TLM introduced in Chapter 3 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. Chapter 3 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. Chapter 3 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.

Limitations

The previous 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 Chapter 3 presents 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 gen-
eration of $P$ was manual. This lack of a broad consideration of interaction patterns and automation limits the practical application of the previous chapter.

The following sections address the gaps in previous chapter as follows.

- This chapter covers a much larger set of interaction patterns. In particular, it explores testing stateful transactions with stateless producers and testing stateless transactions with stateful/stateless producers. Also, for all cases, in addition to the producers, the target transaction $F$ may write to the shared variables.

- This chapter determines 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 $P$.

### 4.3 TLM based concolic test generation for stateful transactions: Overview

This section describes the main ideas for handling stateful transactions and avoiding generating all possible interleavings of $F$ with its interacting transactions.

### 4.3.1 Challenges with stateful transactions: A motivating example

The following example illustrates the challenges in testing stateful transactions.
The example transaction

Consider the example in Code 4.1. This code is a C version of the SERIAL0 transaction in Fig. 4.1 that has been simplified for presentation of the key ideas (Fig. 4.2 is a reduced version of Fig. 4.1). In this example, newpkt, autobaud, and badbaud are 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 temp was not 0xFF, line 8 increases badbaud by reading the old badbaud value updated by the previous instance. Therefore, badbaud is a stateful variable. Next, the instance 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.
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. There are six paths in Fig. 4.2, an HLSM version of Code 4.1. Traditional unit testing for SERIAL0 will find test cases for all six paths. However, some test cases may require values for the stateful variables that cannot be obtained from any sequence of executions from the starting state. Further, note that, due to the reactive nature of transactions, each transaction is continuously executed in a forever loop in its response to requests as follows:

```c
// 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 baud is 10, and during the first iteration, the instance passes through the bold path from st to fh. In fh, the baud value is read, increased by 1, and updated to 11. The following if condition becomes true. Next, the instance traverses fi and end. In the next iteration, suppose the bold path is followed again. In fh, now baud becomes 12 and the if condition is true again. The dashed path from fh to end is not taken again. In fact, from here on, none of the following instances can take the dashed path. This is because the only state which updates baud is fh, and fh can only increase baud. Hence, the path from st to end via fb, fd, and fh is an infeasible path. On the other hand, if the initial value of baud 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. Instead, it is desired to determine the path coverage obtained for a fixed number of iterations. This will also enable stopping at the fewest iterations needed to obtain full path coverage. The rest of this chapter shows how this can be accomplished with our algorithms.

### 4.3.2 Overview of the methodology

Given a TLM with a target firmware transaction \( \mathcal{F} \) and its interacting hardware and firmware transactions, the goal is to automate the test generation for \( \mathcal{F} \) for a large class of the interaction patterns. Our methodology for this is depicted in Fig. 4.3.

First, the Frama-C based automatic transaction analyzer takes as input a TLM with \( \mathcal{F} \). It outputs: (1) whether \( \mathcal{F} \) 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 \( \mathcal{F} \). It generates a sequential program \( \mathcal{P} \) that is test-equivalent to \( \mathcal{F} \). A test generation algorithm using KLEE is used with \( \mathcal{P} \). These test cases can be directly used for \( \mathcal{F} \). Further, the test generation program also reports the coverage obtained for \( \mathcal{F} \). The code generator that generates \( \mathcal{P} \)
handles stateless and stateful transactions differently. Section 4.4 describes the specific algorithms for these cases.

### 4.4 Stateful and stateless test generation algorithms

This section introduces two testing algorithms: a testing algorithm for stateful transactions and a testing algorithm for stateless transactions.

#### 4.4.1 Testing stateful transactions

We first explain the key idea of the testing algorithm for stateful transactions.

**Main idea**

The test generation algorithm faces multiple sources of complexity when $\mathcal{F}$ 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 $\mathcal{F}$ are stateless, then the prior work in Chapter 3...
which uses KLEE for constraining the values for the shared variables in $\mathcal{F}$ to the values that can be produced by its producers, can be used \[2\]. For now, this chapter will consider the restriction to this case. This restriction will be relaxed in the next section where stateful producers are involved. Also, unlike Chapter \[3\] this chapter permits $\mathcal{F}$ to also write the shared variables. The main contribution of this chapter is to show how $\text{C1}$ is handled using a concolic testing based algorithm, and how this is considered with the solution for $\text{C2}$ to constrain the values from the producer transactions. The main idea is to use concolic testing by iteratively unrolling $\mathcal{F}$, 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. A path here refers to a path in a single instance of the transaction. 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 some pre-assigned bound $k$ on the number of iterations in the unrolling is reached. At this point, the path coverage is computed.

This section now illustrate the algorithm on the reduced \texttt{SERIAL0} stateful transaction in Code \[4.1\] interacting with the stateless \texttt{serial_bitrate} producer in Fig. \[4.1\]. The stateful shared variable \texttt{autobaud} is also written by this concurrent producer. For inputs from the physical environment, constrained values are assigned based on the environmental constraints, such as for the \texttt{temp} value (line 2) and the \texttt{pkt} value (line 15) in Code \[4.1\].

**STEP1 : Early Pruning of Infeasible Paths** First, the algorithm performs unit testing for the stateful transaction \texttt{SERIAL0} using KLEE. Instead of the stateful variables \texttt{autobaud}, \texttt{newpkt}, and \texttt{badbaud}, the algorithm substitutes symbolic variables \texttt{autobaud_sym}, \texttt{newpkt_sym}, and \texttt{badbaud_sym} and calls \texttt{SERIAL0}. \texttt{SERIAL0} is run with symbolic values starting from a single thread. However, through its execution, KLEE explores the six possible paths in six different processes. For example, the unique process \texttt{proc1} in Fig. \[4.4\] is dedicated for \texttt{path1} in Table \[4.1\]. Similarly, \texttt{path2} in Table \[4.1\] is executed in \texttt{proc2},
Table 4.1: Path conditions for the six paths produced in STEP1 of Fig. 4.4

<table>
<thead>
<tr>
<th>Path</th>
<th>via</th>
<th>Path condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>path1</td>
<td>( f_b, f_k, f_i, f_m )</td>
<td>( pc_1 = \neg \text{newpkt}<em>{sym} \land \neg \text{autobaud}</em>{sym} &gt; 0 \land \neg \text{pkt} )</td>
</tr>
<tr>
<td>path2</td>
<td>( f_b, f_k, f_m )</td>
<td>( pc_2 = \neg \text{newpkt}<em>{sym} \land \neg \text{autobaud}</em>{sym} &gt; 0 \land \neg \text{pkt} )</td>
</tr>
<tr>
<td>path3</td>
<td>( f_b, f_d, f_f, f_k, f_i, f_m )</td>
<td>( pc_3 = \neg \text{newpkt}<em>{sym} \land \neg \text{autobaud}</em>{sym} &gt; 0 \land (\text{temp} == 0xFF) \land \neg \text{pkt} )</td>
</tr>
<tr>
<td>path4</td>
<td>( f_b, f_d, f_f, f_k, f_i, f_m )</td>
<td>( pc_4 = \neg \text{newpkt}<em>{sym} \land \neg \text{autobaud}</em>{sym} &gt; 0 \land (\text{temp} == 0xFF) \land \neg \text{pkt} )</td>
</tr>
<tr>
<td>path5</td>
<td>( f_b, f_d, f_h )</td>
<td>( pc_5 = \neg \text{newpkt}<em>{sym} \land \neg \text{autobaud}</em>{sym} &gt; 0 \land (\text{temp} != 0xFF) \land (\text{badbaud}<em>{sym}++ \land \text{badbaud}</em>{sym} &lt; 6) )</td>
</tr>
<tr>
<td>path6</td>
<td>( f_b, f_d, f_h, f_i )</td>
<td>( pc_6 = \neg \text{newpkt}<em>{sym} \land \neg \text{autobaud}</em>{sym} &gt; 0 \land (\text{temp} != 0xFF) \land (\text{badbaud}<em>{sym}++ \land \text{badbaud}</em>{sym} &gt;= 6) )</td>
</tr>
</tbody>
</table>

etc. For path1, in Fig. 4.5 only the concrete values of autobaud and newpkt satisfying \( pc_1 \) can exercise it. This condition \( pc_1 \) is built up while KLEE traverses path1. Every time KLEE reaches a branch, it adds (conjuncts) the condition of the branch that it takes to \( pc_1 \) (Fig. 4.5). Hence, in proc1, after running the instrumented SERIAL0, the algorithm can get the corresponding path condition \( pc_1 \). Similarly, each of the six processes has a corresponding path condition after STEP1. Table 4.1 shows the path conditions for all six paths. If a path condition, e.g., \( pc_1 \), cannot be satisfied by the constraint solver then 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. In proc1, after getting \( pc_1 \), 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 \texttt{serial_bitrate}, which writes the shared variable autobaud. In composing the transaction code for the producer \texttt{serial_bitrate}, specialized library functions to read/write the shared variables are used (these library functions are a part of our implementation). Additionally, statically assigned shared variables are used. 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, the algorithm can automatically gather the written values from `serial_bitrate` by calling the producer multiple times. The 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, the algorithm is 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. This accurately captures the producers’ behaviors for use by the target transaction.
Note that the set of output values from the producers can be either a finite set or a range of values. In the former case, the shared registers representing device status values (e.g., pressed buttons) updated by the hardware producers would have only 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 a single value within the particular range can be chosen.

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 six 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, the stateful variables autobaud, newpkt, and badbaud are substituted with concrete variables autobaud_conc, newpkt_conc, and badbaud_conc in SERIAL0. 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. 4.4). Note that, in STEP1, the algorithm runs SERIAL0 with symbolic values to represent all pos-

```
Figure 4.5: path1 of Table 4.1. This is one of the paths in the Rockbox SERIAL0 example from Fig. 4.1
```
sible 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. 4.4), the first instance starts from `st` to `fb`, then goes to `fd` since (`newpkt` && `autobaud` > 0) is true (Fig. 4.2). 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 \( pc1 \land (newpkt\_sym == 1) \land (autobaud\_sym == 1) \land (badbaud\_sym == 4) \) is satisfiable or not (the algorithm selected a constrained value for `pkt` so that `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`, this is repeated. 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 `fk` via `fb`, `fd`, and `ff`. Assuming the `pkt` is 0, the instance goes to `fi`, `fm`, and `end` (`path3`). After the second iteration, the algorithm checks \( pc1 \land (newpkt\_sym == 1) \land (autobaud\_sym == 1) \land (badbaud\_sym == 5) \) again. The formula is still unsatisfiable, so the algorithm keeps unrolling. In the third iteration, since `autobaud` is now 0, `iter3` goes through `fb` and `fk`. 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 \( pc1 \land (newpkt\_sym == 1) \land (autobaud\_sym == 0) \land (badbaud\_sym == 5) \), 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, the algorithm terminates the search.

Note that as the stateful transaction is unrolled, the algorithm considers 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, SERIAL0 may update autobaud to 0 at $f_i$. Then, the next instance is supposed to take the false branch from $f_b$. However, serial_bitrate can write 2 on autobaud in the interim and this drives the next SERIAL0 instance to $f_d$ 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 $f_b$, SERIAL0 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. In other words, the algorithm continues to unroll only if there is no possible set of input/producer values that can exercise the given path with the current number of iterations. KLEE is slightly modified to unroll the target transaction only the least number of times instead of exploring all possible number of iterations for the path in this work. Thus, the algorithm requires only the least possible unrolling for each path and provides the shortest possible test for each path.

**Pseudocode**

Code 4.2 provides the framework for the testing algorithm. In Fig. 4.3, once the automatic transaction analysis is done to determine the interaction patterns and other related information, our code generator composes a customized program based on this framework 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 each producer p{
    for (j=0; j<NUM OF WRITES_on_v_in_p; j++){
      Call producer p
      klee_make_symbolic(&choose); klee_assume(0 <= choose < number of write during the instance);
      wv[j]=s_hw_v.writes[choose]; //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 4.2: Testing stateful transactions with stateless producers

stateful transactions. In Code 4.2, $x[i]$ are the stateful variables, and $NUM\_SFV$ is the number of stateful variables. Lines 1-6 are for STEP1. Lines 1-5 set $x\_sym[i]$ as symbolic and substitute $x\_sym[i]$ for $x[i]$. Then, Line 6 calls the stateful transaction so that all paths of the target transaction are explored in multiple processes. Mode 0 is to use the unit testing ability of KLEE without considering the effect of the initial state and the producers. 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, all possible written values for $v$ from the producers are gathered and stored 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 $x[i]$ to run the target transaction concretely. 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 $pc \land \forall i \in NUM_{SFV}(x_{sym}[i] == x_{conc}[i])$. $pc$ 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 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, the other case does not matter. 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 that are not needed. 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 stateless producers for each shared variable.
1. call the stateless transaction

\[ pc1 \rightarrow \ldots \rightarrow pc3 \]

(proc1)(proc2)(proc3)

2. unroll the stateful producers

\[ r1 \leftarrow w1 \]
\[ r2 \leftarrow w2 \]
\[ r3 \leftarrow w3 \]
\[ r4 \leftarrow w4 \]
\[ r5 \leftarrow w5 \]

3. check \( pc3 \land (w_i = r1) \land (w_j = r2) \)

- \( ex. pc3 \land (w1 = r1) \land (w2 = r2) \)
  at the first iteration

- \( ex. pc3 \land (w2 = r1) \land (w3 = r2) \)
  at the second iteration

Figure 4.6: Illustration of the algorithm for testing stateless transactions with stateless/stateful producers

4.4.2 Testing stateless transactions

We now introduce the testing framework for stateless transactions.

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, all paths including possible infeasible paths of the stateless target transaction are explored, then checked 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. 4.6, 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 this example, there is a shared variable between the stateless transaction and the stateful producer in Fig. 4.6 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 \( wj \) \((i \leq j)\). Hence, after the path exploration, each process along with a path
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(&t, sizeof(t), "t");
            klee_assume(t>=old); klee_assume(t<WRITE_LEN);
            c[i] = 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 4.3: Testing stateless transactions with stateless/stateful producers

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 \( w_1 \) and \( w_2 \). \( i \) and \( j \) are constrained to \( 1 \leq i \leq j \leq 2 \). Then, if \( pc3 \land (w_i == r_1) \land (w_j == r_2) \) is satisfiable, the algorithm stops iterating and break out of the loop as path3 is feasible. If not, the algorithm keeps unrolling, and after the second iteration, \( i \) and \( j \) are constrained to \( 1 \leq i \leq j \leq 5 \). 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, \( r_1 \) or \( r_2 \) either reads from the last value updated by itself or the stateful producers.

**Pseudocode**

The pseudocode in Code 4.3 is written for a single shared variable and a single producer for ease of understanding. Our full implementation supports 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 symbolic value dedicated for each read. $NUM\_READ$ is the number of reads from the shared variables written by the stateful transaction along the particular path (it is 2 in case of proc3 in Fig. 4.6). $y\_sym[0], \ldots, y\_sym[NUM\_READ - 1]$ are the symbolic values 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 producers. Then the algorithm enters the while loop and calls the stateful producer in line 5. Lines 7-12 simply pick indices $c[0], c[1], \ldots, c[NUM\_READ - 1]$ constrained to satisfy $0 \leq c[0] \leq c[1] \leq \cdots \leq c[NUM\_READ - 1] \leq WRITE\_LEN$ where $WRITE\_LEN$ is the number of total writes on the shared variable from the series of the producer instances so far. Then, in line 16, the algorithm checks if the following condition is satisfiable: $pc \land (y\_sym[0] == (prod\_writes[c[0]] \lor \text{any of stateless producers' update})) \land \ldots \land (y\_sym[NUM\_READ - 1] == (prod\_writes[c[NUM\_READ - 1]] \lor \text{any of stateless producers' update}))$ where the $prod\_writes$ array stores the produced values from the stateful producer in order. $\text{any of stateless producers' update}$ refers to the shared variable values updated by the stateless producers gathered in line 2. 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 producers goes on until the condition is satisfied or it reaches the $k$ bound. Again, the simple prior work case of stateless variables shared with stateless producers in Chapter 3 is naturally included in the above.

4.4.3 Summary

Instead of explicitly exploring all the interleavings with the producers, the algorithms introduced in this chapter show how the effect of the producers and the initial states can be
added as a set of constraints to the target transactions. Then, a single-threaded standard
testing tool can be used to verify the sequentialized code. The algorithms also guarantee
minimal number of unrollings for the stateful transactions.

4.5 Implementation

Frama-C [26] is an open-source plugin-based platform written in ML that provides various
static analyses for C code. Each plugin 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). Fig. 4.7 shows the PDG of the SERIAL0
transaction. In Fig. 4.7 the bold edges represent DDG. In this figure, the \(i^{th}\) instance
executes path1. In \(f_k\), pkt is written. Then, \(f_m\) reads this pkt value and writes this value
into newpkt. Thus, there is a data dependence from \(pkt = iap\_getc(temp)\) to \(newpkt =
pkt\) instruction. The formal definitions of stateful transactions and the producer-consumer
relationship [2] are based on the DDG of the transactions. These definitions are used with
the PDG plugin to identify the shared variables and interaction patterns between the target
transaction and its interacting transactions.

We adapted the Frama-C source for our work. If a transaction is analyzed with the original
Frama-C, the PDG plugin would analyze 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. The
following points are the specific analyses done using PDG in this work.

**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. 4.7 there is an edge between
instances \(i\) and \(i + 1\) as \(f_b\) in the \((i + 1)^{th}\) instance reads autobaud value updated from
$f_t$ in the $i^{th}$ instance. Therefore, $SERIAL0$ is stateful. Indirect stateful transactions, in which variable values flow between instances through other transactions, can be detected by analyzing DDGs of multiple transactions in the similar manner.

**Producer-consumer relationship:** The dependence table obtained from the Frama-C PDG provides information about the other 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, it is needed 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 configuration file for the target transaction and a customized
algorithm for it based on the stateful/stateless algorithm framework as appropriate. The algorithm is executed by our modified version of KLEE.

4.6 Experiments

For the experiments, two published benchmarks from Horn et al. [57] (Table 4.2) were used. These benchmarks consist of Linux device drivers [23] and the corresponding QEMU device emulator code [70] written in C. QEMU is an open-source machine emulator supporting a wide range of hardware components including devices such as storage devices, network cards, etc. Horn et al. extracted several device models (Table 4.2) to run stand-alone, i.e., without dependencies 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 chapter, the QEMU and Linux benchmarks can be organized as a set of transactions. Table 4.3 shows TMP105 transactions and their interaction patterns. Table 4.4 describes Ethoc transactions. Some transactions in the tables do not have producers because they act as producers themselves for other transactions. For the stateful transactions with stateful producers case that is not handled completely, the producers are treated as stateless by allowing the stateful variables to have any initial value, giving us over-approximated results for this case.

The experiments show the efficacy of the 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.

Table 4.2: Linux-QEMU device driver benchmarks

<table>
<thead>
<tr>
<th>Device</th>
<th>Main Functionality</th>
<th>LOC of the benchmark</th>
</tr>
</thead>
<tbody>
<tr>
<td>TMP105 [103]</td>
<td>A temperature sensor</td>
<td>2460</td>
</tr>
<tr>
<td>OpenCores ethernet mac</td>
<td>An Interrupt (or polling) driven</td>
<td>4130</td>
</tr>
<tr>
<td>(Ethoc) [78]</td>
<td>Ethernet MAC with a DMA ring</td>
<td></td>
</tr>
</tbody>
</table>
Table 4.3: Interaction patterns of TMP105 transactions

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Interaction Patterns</th>
<th>Interacting Producers</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmp105_tx</td>
<td>stateful + no prod</td>
<td>-</td>
</tr>
<tr>
<td>readword</td>
<td>stateful + no prod</td>
<td>-</td>
</tr>
<tr>
<td>tmp105_set</td>
<td>stateless + stateful prod</td>
<td>tmp105_tx</td>
</tr>
<tr>
<td>show_temp</td>
<td>stateful + stateful prod</td>
<td>tmp105_tx, tmp105_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>tmp105_tx, lm75_resume</td>
</tr>
<tr>
<td>lm75_resume</td>
<td>stateless + stateful/stateless prod</td>
<td>tmp105_tx, lm75_suspend</td>
</tr>
</tbody>
</table>

Table 4.4: Interaction patterns of Ethoc transactions

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Interaction Patterns</th>
<th>Interacting Producers</th>
</tr>
</thead>
<tbody>
<tr>
<td>open_eth_moder</td>
<td>stateful + stateless prod</td>
<td>reg_write</td>
</tr>
<tr>
<td>open_eth_desc_write</td>
<td>stateful + stateless prod</td>
<td>reg_write</td>
</tr>
<tr>
<td>open_eth_receive</td>
<td>stateful + stateful prod</td>
<td>open_eth_moder</td>
</tr>
<tr>
<td>open_eth_reg_read</td>
<td>stateless + no prod</td>
<td>-</td>
</tr>
<tr>
<td>ethoc_rx</td>
<td>stateful + no prod</td>
<td>-</td>
</tr>
<tr>
<td>ethoc_tx</td>
<td>stateful + no prod</td>
<td>-</td>
</tr>
<tr>
<td>ethoc_interrupt</td>
<td>stateful + stateful prod</td>
<td>ethoc_rx, ethoc_tx, ethoc_poll</td>
</tr>
</tbody>
</table>

4.6.1 Test generation results

The result of test generation for the stateful transactions in Tables 4.3 and 4.4 are provided in Table 4.5. The testing algorithm could cover nine out of 10 stateful transactions completely within the bound \( k = 6 \). In the open_eth_moder transaction, the customized testing code 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. The testing algorithm could cover these branches by finding satisfying stateless variable values only. Hence, the testing code does not have to unroll these stateful transactions multiple times. Similarly, the testing algorithm could cover the open_eth_receive transaction after an iteration, but this transaction has a while loop inside
Table 4.5: Testing results of the stateful/stateless transactions

<table>
<thead>
<tr>
<th>Type</th>
<th>Transaction</th>
<th>k-bound</th>
<th>Time (sec)</th>
<th>Number of test cases</th>
<th>Complete</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stateful</td>
<td>tmp105_tx</td>
<td>1</td>
<td>5.584</td>
<td>12</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>20.105</td>
<td>17</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>50.647</td>
<td>22</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>readword</td>
<td>1</td>
<td>0.112</td>
<td>4</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0.104</td>
<td>6</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>show_temp</td>
<td>1</td>
<td>1902.55</td>
<td>3193</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>open_eth_moder</td>
<td>1</td>
<td>5.012</td>
<td>79</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>46.163</td>
<td>262</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>326.724</td>
<td>777</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>3627.475</td>
<td>2235</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>time-out</td>
<td>10409</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>open_eth_desc_write</td>
<td>1</td>
<td>0.196</td>
<td>20</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0.224</td>
<td>20</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>0.476</td>
<td>20</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>2.424</td>
<td>84</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>21.617</td>
<td>596</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>6</td>
<td>301.987</td>
<td>3156</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>open_eth_receive</td>
<td>1</td>
<td>time-out</td>
<td>122221</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>ethoc_rx</td>
<td>1</td>
<td>0.276</td>
<td>19</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>ethoc_tx</td>
<td>1</td>
<td>0.180</td>
<td>8</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>ethoc_interrupt</td>
<td>1</td>
<td>20.725</td>
<td>387</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>SERIAL0</td>
<td>1</td>
<td>0.188</td>
<td>30</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>0.324</td>
<td>59</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>0.720</td>
<td>116</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
<td>1.808</td>
<td>229</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>5</td>
<td>5.440</td>
<td>457</td>
<td>Yes</td>
</tr>
<tr>
<td>Stateless</td>
<td>tmp105_set</td>
<td>1</td>
<td>0.144</td>
<td>39</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>set_temp</td>
<td>1</td>
<td>28.802</td>
<td>60</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>lm75_suspend</td>
<td>1</td>
<td>7.768</td>
<td>40</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>13.281</td>
<td>40</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
<td>18.749</td>
<td>44</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>lm75_resume</td>
<td>1</td>
<td>5.712</td>
<td>25</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td>2</td>
<td>9.109</td>
<td>27</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>open_eth_reg_read</td>
<td>1</td>
<td>0.0092</td>
<td>1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

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. \textit{SERIAL0} transaction is tested as well in this experiments.

The result of testing the stateless transactions in Tables 4.3 and 4.4 are in Table 4.5. Some transactions, such as \textit{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.
Figure 4.8: Simplified `open_eth_moder_host_write` transaction in which a bug is found

### 4.6.2 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. 4.8 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 `ma`. If true, the transaction is allowed to reset the `MODER` register. In this case, the transaction sets the `MODER` register to `0000A000h` (`mb`). 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` (`mc`). Hence, each instance will not set `MODER` to `0000A000h` at the end as the data sheet [78] 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 `ma`, `mb`, `mc`, and `me`, 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.
4.6.3 Limitations

This work is driven by the primary motivation of avoiding exploring an exponential number of interleavings to consider the impact of interacting transactions on $\mathcal{F}$. This chapter has shown how their impact can be considered through the algorithms presented in Section 4.4. This work generates a custom sequential program $\mathcal{P}$ that captures the interactions of other transactions for specific cases as a set of constraints. In particular, this chapter 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, it is needed 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].

4.7 Chapter summary

Validating firmware is challenging as it needs to consider the concurrency between firmware and its interacting hardware/firmware components. Chapter 3 introduced a uniform transaction-based modeling framework for firmware/hardware, which 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 interleavings of transactions during test generation. This chapter extends this work to consider a much larger class of patterns that include transactions that may be stateful. This extended class covers the most common interactions patterns seen in practice. Further, this chapter provides 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 $F$, our code generator automatically produces a
sequential program $P$ that composes $F$ with its interacting transactions. This enables a
slightly modified version of the KLEE concolic testing tool to be used for generating tests
for $P$, which can be directly used for $F$. 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.
Chapter 5

Completeness Bounds and Sequentialization for Model Checking of Interacting Firmware and Hardware

5.1 Introduction

Chapter 4 provided a sequentialization method based on the transaction interaction patterns. However, it cannot cover the case where both the target firmware transaction $F$ and its producers are stateful, and does not compute sufficient bounds for the input unbounded transactions. Instead, Chapter 4 used a given constant bound to verify unbounded transactions. Hence, the testing result is incomplete. Many software model checkers discover counterexamples of deep bugs only after exploring a large number of iterations/unrollings of such loop structures. Bounded model checking (BMC) [7] searches for property violations within a given bound $n$ of unrollings. However, without a threshold for $n$ that guarantees the completeness of verification, this method cannot prove correctness. A key challenge in using BMC lies in computing such a completeness threshold. A key contribu-
tion of this chapter is a new technique to determine a sufficient BMC bound to prove the property or to find a violation. It is a static analysis approach that exploits characteristics of common code patterns found in firmware to provide inexpensive termination checks for unbounded verification.

Based on this bound analysis, this chapter presents a new compositional form of sequentialization. The previous interaction pattern-based sequentialization method is paired with the bound analysis technique to improve the previous work in Chapter 4. As a result, the general sequentialization technique covers all common interaction patterns (it is a unified framework for both stateless/stateful $F$ with any type of producers). It does not miss any errors when completeness bounds can be successfully computed. This sequentialization method can be independently combined with a broad range of verification techniques.

This chapter applies BMC on the resulting program using the widely-used software model checker CBMC [20].

Our fully automated methodology is outlined in Fig. 5.1. As shown, our BMC bound analysis takes as inputs the target firmware code $F$ and the result of transaction interaction pattern analysis, and provides the sufficient BMC bound for $F$. The producer bounds are computed by producer bound analysis in a similar way. Based on these bounds, our sequentialization code generator creates the single-threaded program $P$ for use with the CBMC model checker. All boxes shown in bold are implementations of contributions made in this chapter.

The practical applicability of this methodology is shown in experiments on code for some Linux device drivers and interacting QEMU hardware emulator. Three large device benchmarks [57] are studied in this chapter. The efficacy of our methodology is demonstrated by verifying 16 transactions in these benchmarks.

This chapter makes the following contributions:
It provides automated inexpensive termination checks for BMC of unbounded models for several commonly occurring cases. This chapter presents an algorithm that leverages common code patterns to determine a sufficient BMC bound for the firmware transaction $F$ in the context of infinite loops. (§5.3-5.4)

It describes a general algorithmic framework that reduces the concurrent system with $F$ and its interacting producers to a sequential program $P$ using the bound analysis and the common interaction patterns of firmware and hardware transactions. Our code generator automatically constructs $P$, which can be verified by a model checking tool. (§5.5)

It shows the efficacy of our model checking methodology on 16 transactions with 46 properties from three published real firmware benchmarks of the Linux-QEMU platform. (§5.6)

## 5.2 Background

This section provides the necessary background for understanding this chapter.
5.2.1 Service function-based TLMs

This chapter is based on the service function-based TLM introduced in Chapter 3 with a few assumptions: (1) all loops inside a transaction are bounded – this is common for firmware and hardware components, and (2) in the producer-consumer relationship, the shared variables are only written by their producers, i.e., a shared variable consumed by a transaction is stateless. This is common in embedded systems since many shared registers are written by a single master module. Thus, all stateful variables referred to in this chapter are local to the transaction under analysis.

Programming language syntax and semantics for TLM

This section presents a simple programming language to describe the TLM.

\[
x, y \in \text{Vars} \\
e \in \text{Expr} \quad ::= \quad x \mid \text{const} \mid e \ \text{arithop} \ e \mid e \ \text{bitop} \ e \mid (e)
\]

\[
\phi \in \text{Bool} \ \text{expr} \quad ::= \quad \text{true} \mid \text{false} \mid e \ \text{relop} \ e \mid \phi \lor \phi \mid \phi \land \phi \mid \neg \phi
\]

\[
s, t \in \text{Stmt} \quad ::= \quad \text{skip} \mid x := e \mid \text{havoc} \ x \mid \text{assert} \ false \\
\quad \mid \text{if} (\phi) \ \text{then} \ Sseq \ \text{else} \ Sseq \ | \text{while} (\phi) \ \text{do} \ Sseq
\]

\[
Sseq \quad ::= \quad \text{Stmt}^{*}(+ \text{ represents one or more})
\]

\[
P \in \text{Program} \quad ::= \quad Sseq
\]

Variables are either local to a transaction or shared by multiple transactions. \text{arithop}, \text{bitop}, and \text{relop} denote arithmetic, bitwise, and relational operators, respectively. The \text{skip} statement denotes a no-op, and \text{havoc} \ x \ statement assigns \ x \ a non-deterministic value from an appropriate set. Reading from a shared variable is modeled via \text{havoc}, and \text{assert true} is implemented via \text{skip}. A property violation \text{assert} \ \phi \ is expressed as \text{if} (\phi) \ \text{then} \ \text{assert false}.
5.2.2 Program dependence

For the rest of the chapter, a program is represented by its dependence graph. The formal definition of the dependence graph is provided in this section.

**Definition 11** (Control flow graph). A control flow graph (CFG) \( G_F = (V, E, v_I, v_F) \) is a directed graph with a distinguished initial vertex \( v_I \) (with no predecessor) and a final vertex \( v_F \) (with no successor) \[85\]. The vertices \( V \) represent the program statements and the edges \( E \) represent potential transfers of control between \( V \).

**Definition 12.** A vertex \( v \) in \( G_F \) post-dominates vertex \( u \) iff all paths in \( G_F \) from \( u \) to \( v_F \) go through \( v \).

**Definition 13** (Dependence graph). A dependence graph \( G_D \) has the same vertices as \( G_F \) and two sets of edges, control dependence edges and data dependence edges, defined below.

**Definition 14** (Control dependence edge). There is a control dependence edge from \( v_c \) to \( v \) in \( G_D \) if there exists a path \( P \) from \( v_c \) to \( v \) in \( G_F \), and \( v \) post-dominates all vertices between \( v_c \) and \( v \) in \( P \) (exclusive) but does not post-dominate \( v_c \).

**Definition 15** (Data dependence edge). A data dependence edge \( e \) exists from vertex \( v_{\text{def}} \) to vertex \( v_{\text{use}} \) in \( G_D \) if (1) there is at least one path in \( G_F \) from \( v_{\text{def}} \) to \( v_{\text{use}} \), (2) a value is assigned to a variable \( x \) in \( v_{\text{def}} \) and \( v_{\text{use}} \) uses \( x \) in an expression, and (3) \( x \) is not assigned at any vertex between \( v_{\text{def}} \) and \( v_{\text{use}} \) along some path in \( G_F \).

**Definition 16.** For a vertex \( v \), \( \text{data}(v) \) is the set of vertices where a data dependence edge exists from each of \( u \in \text{data}(v) \) to \( v \).

**Definition 17.** For each vertex \( v \), \( \text{ctrl}(v) \) is the unique vertex where there exists a control dependence edge from \( \text{ctrl}(v) \) to \( v \).

\( \text{ctrl}(v) \) is the most immediate conditional branch to \( v \). Note that \( \text{ctrl}(v) \) is \( \emptyset \) if there is no conditional branch to \( v \).

Further, standard Def-Use terminology is used to define some terms. \( D(u) \) is the variable defined in vertex \( u \). \( U(v) \) is the set of variables used in vertex \( v \).
TLM structure and its dependence graph

A transaction $T$ is represented as follows: $init(); while(1)T();$. $init()$ is a sequence of statements that initialize the variables of $T$. We denote this set of vertices as $V_{INIT}$. Within the transaction body $T$, the direction of a data dependence edge can be against the control flow. In other words, even if a vertex $u$ precedes $v$ in the control flow graph, there can be a data dependence edge from $v$ to $u$. This represents the situation where the data flows from $v$ in an iteration (instance) to $u$ in some future iteration of the outer while loop. We use the expression $\text{old}(x)$ to refer to the value of $x$ at the entry to an instance, i.e, the value of $x$ written in an earlier instance. Note that such an $x$ is a stateful variable. A data dependence edge in $G_D$ from $v_{def}$ where $D(v_{def}) = x$ to vertex $v_{use}$ where $v_{use}$ uses $\text{old}(x)$ is an edge across two or more instances. We refer to this edge as an inter-instance edge.

5.2.3 Bounded model checking (BMC)

Bounded model checking (BMC) [7] is a widely used bug-finding method that examines paths of bounded length to search for violations of a given property. Given the transition relation of a target system, a property, and a bound $n$, the transition relation is unrolled $n$ times to obtain a formula, which is satisfiable if and only if there exists a violation for the property within bound $n$. However, this technique is inconclusive if the formula is unsatisfiable, since there may be counterexamples longer than $n$. Hence, a key challenge lies in determining a sufficiently small completeness threshold $k$ that guarantees completeness of verification. In other words, if there is no violation up to completeness bound $k$, then the property is proved to be correct. To limit unbounded path enumeration, loop unwinding techniques are also used. Instead of unwinding the transition relation, each loop is separately unwound up to the bound. In the TLM, each firmware transaction execution is in an infinite loop (referred to as the outer loop to distinguish it from loops within the transactions). This chapter determines sufficient bounds for guaranteeing completeness of BMC.
even with unbounded TLMs. The outer unbounded loop can then be unwound up to its completeness bound for use with a standard BMC tool, such as CBMC [20] for complete verification.

5.3 Completeness bounds: Key ideas

The first part of this chapter provides a bound on the number of unrollings that is sufficient to find either a property violation (counterexample) or prove that the property is not violated even with an unbounded number of unrollings. This bound is defined as the sufficient BMC bound.

**Definition 18** (Sufficient BMC bound). For a property violation at vertex \( v \) in a transaction executed in an infinite loop, if there is no path to reach \( v \) by unrolling the transaction up to the sufficient BMC bound \( B(v) \), then there will be no path to reach \( v \) longer than \( B(v) \) unrollings.

\( B(v) \) does not guarantee the reachability of \( v \), but guarantees that the model checker does not need to unroll more than \( B(v) \).

Computing \( B(v) \) is related to the reachability of \( v \). In the \( k \)-unrollings of a transaction \( T \), the reachability of a vertex \( v \) in the \( m \)th instance is denoted as \( R_m(v) \). \( R_m(v) \) is determined as follows:

**Lemma 1.** \( R_m(v) = (R_m(ctl(v)) \land \phi_m(ctl(v), v)) \) where

\( R_m(\emptyset) = \text{true} \) (terminating condition), and

\( \phi_m(ctl(v), v) \) denotes the evaluation of the branch condition from \( ctl(v) \) to \( v \) in the \( m \)th instance.

**Proof.** If \( R_m(ctl(v)) \) is true, then the branch condition at vertex \( ctl(v) \) is reachable in the \( m \)th instance. Also, if this branch condition is evaluated to true, i.e., \( \phi_m(ctl(v), v) \) is true, \( v \) is reached in the \( m \)th instance.
\[ B(v) \] can be formally defined using the reachability formula.

**Definition 19 (Reachability formula).** For the property violation vertex \( v \) (assert false), if \[ \exists k \text{ s.t. } \forall l, \exists m \text{ s.t. } l > k \geq m, R_l(v) \Rightarrow R_m(v), \] then \( B(v) = k. \)

This definition is the key to finding \( B(v) \). It ensures that if \( R_m(v) \) is false for all \( m \leq k \), then \( R_l(v) \) is also false for all \( l > k \). Intuitively, if the property violation \( v \) is not found up to the instance \( k \), it is not needed to check beyond the \( k \)th instance. The first part of this chapter explains how to find such \( k \) satisfying the reachability formula in Definition 19, given Lemma 1 and the structural information of the TLM model.

In the service function-based TLM, the states of stateful variables persist between instances and determine the behavior of future instances. In other words, the reachable states of the stateful variables in the \( k \)th instance decide the reachability of a particular vertex. **Hence, stateful variables with inter-instance data dependence edges are key to computing the bound. In particular, the dataflow patterns through inter-instance edges of relevant stateful variables are used to compute the sufficient BMC bound.**
5.3.1 Dataflow chains in TLMs

The rest of this section introduces different types of dataflow code patterns of the stateful variables. Code 5.1 is a synthetic example of a transaction with various dataflow code patterns that can be found in real world firmware code. This example is used throughout this chapter.

Types of a vertex

A vertex in the TLM is one of the three types:

<table>
<thead>
<tr>
<th>Types of $v$</th>
<th>Reachability</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) $v \in V_{INIT}$ of the transaction</td>
<td>init only</td>
</tr>
<tr>
<td>(b) $ctrl(v) = \emptyset$</td>
<td>unconditional</td>
</tr>
<tr>
<td>(c) Vertex with non-null $ctrl(v)$</td>
<td>may be conditional</td>
</tr>
</tbody>
</table>

An (a) vertex is executed once. For (b), $R_k(v)$ is true for any $k$. In case of (c), vertex $v$ which control-depends on a branch vertex $u$ is executed when $u$ is reached and $\phi(u, v)$ is true. As an illustration, a simple transaction is constructed in Code 5.2 by extracting code related to the variables $old_1, old_2, a, b,$ and $c$ from Code 5.1. It receives fresh non-deterministic values for $new$ and $a$ in every instance from its environment. Also, suppose that $I1$ is an assert false statement. Fig. 5.2 represents the dependence graph of this code. In Fig. 5.2 the vertices above the horizontal line are (a) type vertices. $I1$ and if $(c > old_2)$ are (c) type vertices. The rest are (b) type vertices. In this figure, an intra-instance data dependence edge labeled as $d$ represents a data dependence edge which is not inter-instance. An edge labeled as $id$ is an inter-instance data dependence edge. An edge labeled as $c$ is a control dependence edge.
Dataflow chain

The dataflow of a variable is represented by its chains. Each chain represents the *lifetime of a certain assignment* of the variable.

**Definition 20 (Chain).** A chain for a variable $x$ at an if vertex $v$ ($x \in U(v)$) in a transaction $T$ is a finite sequence of connected data dependence edges
- starting at a source vertex $v_{src}$ where data($v_{src}$) = \emptyset
- passing through a vertex $v_{def}$, where $D(v_{def}) = x$, with edge from $v_{def}$ to $v$, and
- ending at $v$.

For example, there are four chains defined at vertex if($old1 == old2$) in Fig. 5.2. One is defined for $old1$ variable, and three are defined for $old2$ variable. A source vertex is in the form of $x := const$ or havoc $x$. The length of a chain is defined as the number of the edges.

all_chains($v$) are all data chains that affect the reachability of $v$.

**Definition 21 (all_chains($v$)).** all_chains($\emptyset$) = $\emptyset$. For a vertex $v$, if $v$ is an if vertex, ch$_{base}$ is a union of the set of chains at $v$ and the set of chains at ctrl($v$). If $v$ is not an if vertex, ch$_{base}$ is ch$_{base}$ of ctrl($v$). all_chains($v$) is the set of chains consisting of ch$_{base}$ and all_chains($u$) for each vertex $u$ in ch$_{base}$.

For example, all_chains(I1) in Fig. 5.2 is the set of chains defined at vertex if($c > old2$) and the chains defined at vertex if($old1 == old2$).

**Bounded vs. unbounded chains**

The length of a chain can be unbounded when there is a cycle in the chain. For example, the lengths of all the chains in Fig. 5.2 are bounded, but the chain at vertex if($len == 4$) in Code 5.3 has a cycle due to the variable len and thus, is unbounded. A cycle in the data dependence chain indicates that there is a variable $x$, which data-depends on old($x$). This chapter refers to such a variable as a self-modifying variable.
Definition 22 (Self-modifying variable). Let $v$ be a vertex in a chain such that there is a path from $v$ to $v$. Let $x = D(v)$. $x$ is a self-modifying variable.

The data dependence edge from $v$ goes back to $v$ (possibly through other vertices) forming a cycle. $x$ may read its value from an earlier instance and use it to compute a new value of itself.

5.4 Completeness bound: Algorithm

Based on the common code patterns, this section presents an automated algorithm that determines a sufficient BMC bound for the firmware transaction $\mathcal{F}$.

5.4.1 Sufficient BMC bound for bounded chains

For a vertex $v$, if the chains in $\text{all\_chains}(v)$ are bounded, any reachable state of a variable in $U(v)$ can be reached within a bounded number of instances as the state is derived from the source vertices of the bounded chains. $B(v)$ has the following properties: Within
$B(v)$, (1) all reachable states of each variable $\in U(v)$ can be reached and (2) all reachable combinations of the states of the variables $\in U(v)$ can be reached. (2) implies that the evaluation of the branch condition at $v$ (if $v$ is an if vertex) will have all the reachable outcomes within $B(v)$, i.e., if the condition can ever be true, then it will be evaluated to true within $B(v)$. Note that this is a conservative bound. Hence, a reachable vertex $w$ such that $\text{ctrl}(w) = v$ can be reached in some instance $m \leq B(v)$, and $\phi_l(v, w) \Rightarrow \phi_m(v, w)$ is true for $m \leq B(v), l > m$.

Computing for (1) is relatively easy as for each variable $x \in U(v)$, all reachable states of $x$ are reachable within $\max(it + B(u))$ for $\{u|u \in \text{data}(v), D(u) = x\}$ instances ($it$ is for the case where the data dependence edge between $u$ and $v$ is inter-instance, and $it = 0$ otherwise). However, it is necessary to consider all combinations of reachable states of variables in $U(v)$ needed to evaluate $D(v)$ in $v$. Finding all reachable combinations of all variables in $U(v)$ is trickier. For example, for $x, y \in U(v)$, suppose we look for a combination $(x, y) = (c_x, c_y)$, which is required for the branch condition to be true for a branch vertex $v$. When $x$ is $c_x$, $y$ may not be $c_y$ in the same instance. By the time $y$ becomes $c_y$, $x$ may not be $c_x$ anymore, and we do not know when $x$ becomes $c_x$ again. To solve this, the following lemma is used.

**Lemma 2.** For a vertex $v$ with bounded all chains($v$), if $v$ is executed in the $i^{th}$ instance, then either (1) $v$ can be made to execute between instance $(i+1)$ and $(i+B(v))$ (inclusive), or (2) $v$ is never executed again.

**Proof.** Case 1. No source vertex of the chains in all chains($v$) is a type-(a) vertex: Suppose that the first time $v$ is executed is in the $j^{th}$ instance. Then $j \leq B(v)$ by Definition [19] All the source vertex values of the chains in all chains($v$) between the first and the $j^{th}$ instance can be assigned to the same source vertices between instance $(i + 1)$ and $(i+j)$ (inclusive). This is possible because first, the source vertices are either $x := \text{const}$ or havoc $x$ so that choosing a specific value for havoc is possible, and second, all the type-(b)
source vertices are reachable always. Accordingly, a type-(c) source vertex executed in the
ingstance $k$ can be executed in the instance $(i + k)$. Hence, $v$ can be executed in the instance
$(i + j)$.

Case 2. Some source vertex of the chains in all.chains($v$) are type-(a) vertices: There are
two sub-cases. First, $v$ can be reached in a way without the source values in $V_{INIT}$ within
$j$ instances. $j$ should satisfy $j \leq B(v)$ as any reachable combination of values in $U(v)$
should be possible within $B(v)$. Then the proof of this case is the same with Case 1 and
relies on all source vertices being either $x := \text{const}$ or havoc $x$. Second, if $v$ can only
be reached with some sources in $V_{INIT}$, which never occur again, $v$ can never be reached
again. This case will correspond to (2) of Lemma 2. □

This lemma holds because of the repetitive nature of transactions. With bounded chains,
a state of a variable can show up repeatedly within a bounded number of instances. In
Fig. 5.2, by assigning 0 and 1 to the source havoc new in the first and second instance,
respectively, old1 == old2 is true in the third instance. If 0 and 1 are assigned to new in the
third and fourth instances again, old1 == old2 is true in the fifth instance. Hence, vertex
if($c > old2$) can be reached repeatedly where the duration is no greater than $B(\text{if}(c > old2))$. If all reachable states of a variable $x$ can be reached in $B_x$ instances and repeated
thereafter, then all reachable state combinations of $x$ and $y$ need no more than $B_x \times B_y$
instances.

Code 5.4: Pseudocode for computing $B(v)$

```plaintext
B_c = (ctrl(v)==0)?0:B(ctrl(v));
B_d = 1;
For each $x \in U(v)$
   $B_x = 1;
   For each $u$ s.t. $u \in data(v)$ \&\& $D(u) = x$
      $it(u,v) = (\text{inter-instance edge between } u \text{ and } v)?1:0;$
      $B_x = \max(B_x, it(u,v)+B(u))$
   $B_d *= B_x;$
\text{return } B_c * B_d;
```
Using the lemma, the algorithm to compute $B(v)$ is presented in Code 5.4. In Code 5.4, first, within $B_c$, $\text{ctrl}(v)$ can be reached (line 1) and all reachable states of the $D(\text{ctrl}(v))$ variable can be reached. Second, all states of $D(v)$, or all reachable combinations of the variables $x \in U(v)$, can be reached within $B_d$. $B_d$ is computed using $B_x$ as defined in Code 5.4. All reachable states of $x$ can be reached within $B_x$ as $x$ receives values from a set of vertices $\{u|u \in \text{data}(v), D(u) = x\}$. A particular value of $x$, or $c_x$, that $v$ can read is reachable within $B_x$ instances. Similarly, a reachable value of $y$ in $U(v)$, or $c_y$, is reachable within $B_y$ instances. Hence, all reachable combinations of $x$ and $y$ need no more than $B_x \times B_y$ instances. Therefore, $B_d = \prod_{x \in U(v)} B_x$. Lastly, for $v$ to be reached while $U(v)$ variables have a particular combination at the same instance within $B(v)$, $B(v)$ should be $B_c \times B_d$.

This recursive algorithm may result in an infinite loop if there is a cycle composed of one or more control dependence edges and one or more data dependence edges (this cycle is different from the cycles in the chain composed of only the data dependence edges). The algorithm stops the computation and returns when it meets a visited vertex due to such cycles as that control vertex has already been seen.

Note that this algorithm does not assume any restriction on the value of source vertices, or any constraints between the source vertices. However, this is a naive algorithm to calculate the sufficient BMC bound. The computed bound can blow up quickly. Hence, a special but common case that often makes $B(v)$ smaller in practice is introduced.

**Definition 23 (Independent chains).** Two chains are independent if

1. they do not have common vertices, or

2. they end at the same vertex $v$ and there is exactly one path through their shared vertices and that path ends at $v$, or

3. the source vertices of either chain are type-(a) or type-(b) vertices.
For \((2)\), \textit{chain} 1 does not affect any vertex in \textit{chain} 2 that is not shared. For \((3)\), the source values of a chain are reachable independent of other chains. If the chains in \textit{all\_chains}(v) have source vertices of type-(a) or type-(b), as the source assignments occur in every instance, \(v\) can be reached in every subsequent instance after \(B(v)\) by giving the same assignments to the sources in every instance. For example, in Fig. 5.2, the chains at \(\text{if}(c > \text{old}2)\) are all independent.

**Lemma 3.** For a vertex \(v\), if \(R_k(v)\) is true, \(R_l(v)\) is true for all \(l > k\) if the chains in \textit{all\_chains}(v) are bounded and independent.

**Proof.** Follows from the fact that the source values can be set independently in each instance. \(\Box\)

Using Lemma 3, Code 5.4 can be adjusted as follows:

**Lemma 4.** (1) \(B_d\) is reduced to \(\max(BD, BI)\) where

- \(W = \{ w \mid w \in \text{data}(v) \text{ s.t. the chains in all\_chains}(w) \text{ are independent of each other} \}\), and

- \(BI = \max(it(w, v) + B(w), \forall w \in W)\)

- \(B_x = \max(it(u, v) + B(u)), u \in (\text{data}(v) - W) \text{ and } D(u) = x\)

- \(BD = \prod_{x \in (D(\text{data}(v) - W))} B_x\)

(2) \(B(v)\) is reduced to \(\max(B_c, B_d)\) where

- the chains in \textit{all\_chains}(w), \(\forall w \in \text{data}(v)\) are independent of each other, or the chains in \textit{all\_chains}(\text{ctrl}(v)) are independent of each other, and

- the chains in \textit{all\_chains}(w), \(\forall w \in \text{data}(v)\) are independent of the chains in \textit{all\_chains}(\text{ctrl}(v)).

**Proof.** Similar to Lemma 3 \(\Box\)

This algorithm can compute \(B(I1) = B(\text{if}(c > \text{old}2))\) in Fig. 5.2. Each chain at \(\text{if}(c > \text{old}2)\) defined for \(c\) is independent of the chains in \textit{all\_chains}(\text{if}(c > \text{old}2)).\) Similarly, the
chains defined for old2 at if(c > old2) are independent. Hence, BI for if(c > old2) is \( \text{max}(4, 3) = 4 \). Intuitively, any reachable value for c or old2 is reachable within the fourth instance. Finally, \( B(\text{if(old1 == old2)}) \) is \( \text{max}(2, 3) \). Hence, \( B(I1) = \text{max}(4, 3) \).

Before going to the next section, we clarify the following point: The shared variable values from the producers are modeled as non-deterministic. With such over-approximation, there is no trace to the property violation longer than the computed BMC bound of the consumer transaction.

5.4.2 Sufficient BMC bound for unbounded chains

When there is a cycle in the data dependence chain due to a variable \( x \) (i.e., \( x \) is a self-modifying variable), as shown in the example in Fig. 5.3, the number of instances needed to reach all reachable states of \( x \) are not easy to bound. There can be various updates of the self-modifying variables, and it is difficult to predict their behavior unless there is a particular pattern. Hence, this work focuses on monotonic variables, a special but common type of self-modifying variables.

Monotonic variables

Definition 24 (Monotonic variables). For a monotonic variable \( x \), the increment \( \Delta x = x_{i+1} - x_i \), where \( x_i \) is the value of \( x \) in the \( i^{th} \) instance for \( i > 0 \), is a constant.
While a variable may be monotonically changing in other ways (such as $x = x \times A$ where $A > 0$), the above arithmetic progression definition is useful due to its prevalence in benchmarks and ease of analysis.

If a monotonic variable does not stop changing at some bound, it has an unbounded-chain. In practice, in the infinite loop context, variables are often designed to change monotonically over a certain range. Hence, a variant of a monotonic variable is defined.

**Definition 25.** A **wrap-around monotonic variable** $x$ increases/decreases until it reaches some limit in an instance, then it wraps around back to its initial (or intermediate) value.

Non-wrap-around monotonic variables are never assigned the source value again, as shown in Fig. 5.3. In this example, to satisfy $x == C$, at least $C + 1$ instances are needed. After that, the predicate takes the branch $x! = C$ only. A wrap-around monotonic variable repeatedly goes back to its source value whenever it reaches some limit, as shown in Fig. 5.4. Hence, in this case, $x == C$ branch is taken every $C + 1$ instances. Any branch condition from the if vertex can be evaluated to true within $C + 1$ instances for both types of the monotonic variables. This is applied in the sufficient BMC bound computation.

**Fitness values**

Previous loop analysis research [106, 46] showed how to compute fitness values, which reflect the number of iterations needed to satisfy a target predicate in certain forms, given inductive loop invariants. When our algorithm in Code 5.4 visits a branch vertex (e.g., if($x == C$)) with monotonic variables, it accesses Table 5.1 which calculates the fitness values (e.g., $C + 1$) for the unbounded cycles with simple update patterns in rows (3) to (10). $C$ represents a constant, and $A$ is the invariant increment/decrement of the monotonic variables. $B$ is the initial value. If the algorithm finds a matching predicate form and the update pattern of the self-modifying variable in the table, then it adds the corresponding fitness value to the bound of the update vertex (e.g., $x++$) instead of adding 1 (representing
Table 5.1: Fitness value calculation table

<table>
<thead>
<tr>
<th>Vertex ( v )</th>
<th>Update pattern</th>
<th>Fitness value</th>
</tr>
</thead>
<tbody>
<tr>
<td>uses stateless vars only</td>
<td>(1) ( x&amp;=C ) or ( x</td>
<td>=C )</td>
</tr>
<tr>
<td>bounded chains only</td>
<td>(2) ( x==C )</td>
<td>calculated using Code 5.4</td>
</tr>
<tr>
<td>-</td>
<td>(3) ( x+=A ) or ( x-=A ) ( C\geq B )? ((C-B)/A+1: ) infeasible</td>
<td>1</td>
</tr>
<tr>
<td>( x==C )</td>
<td>(4) ( x+=A ) ( B\geq C )? ((B-C)/A+1: ) infeasible</td>
<td>( (C-B)/A+1 )</td>
</tr>
<tr>
<td>( x!\neq C )</td>
<td>(5) ( x-=A ) ( C!=B )? ((C-B)/A+1: ) infeasible</td>
<td>( (C-B)/A+1 )</td>
</tr>
<tr>
<td>( x\geq C )</td>
<td>(6) ( x+=A ) or ( x-=A ) ( C!=B )? ((C-B)/A+1: ) infeasible</td>
<td>( (C-B)/A+1 )</td>
</tr>
<tr>
<td>( x&lt;C, ) ( x\geq C )</td>
<td>(7) ( x+=A ) ( B-C&gt;0)? ((B-C)/A+1: ) infeasible</td>
<td>( (B-C)/A+1 )</td>
</tr>
<tr>
<td>( x\leq C )</td>
<td>(8) ( x-=A ) ( B-C&gt;0)? ((B-C)/A+1: ) infeasible</td>
<td>( (B-C)/A+1 )</td>
</tr>
<tr>
<td>( x\leq C )</td>
<td>(9) ( x+=A ) ( C-B&gt;0)? ((C-B)/A+1: ) infeasible</td>
<td>( (C-B)/A+1 )</td>
</tr>
<tr>
<td>( x\leq C )</td>
<td>(10) ( x-=A ) ( C-B&gt;0)? ((C-B)/A+1: ) infeasible</td>
<td>( (C-B)/A+1 )</td>
</tr>
</tbody>
</table>

the inter-instance edge). As an example, Code 5.3 (and Fig. 5.5) is an extracted code from Code 5.1. Vertex if \( len == 4 \) in Fig. 5.5 is a summarized vertex, and we use the bound 5, which is calculated based on its update pattern and the initial value. Here, \( B(I1) = B(if(old1 == old2)) = B(if(len == 4)) \times max((1 + B(old1 = |new| + 1)), (1 + B(old2 = old1 + 1))) \). \( B(if(len == 4)) = 5 \). Hence, \( B(I1) \) is \( 5 \times max(2, max(2, 5)) \), or 25. Intuitively, since \( len \) becomes greater than 4 in five instances, 5 is needed for \( old2 \) to be assigned (line 4), and 5 is needed for line 3 to be executed after \( old2 \) gets the valid value. If \( len \) is a non-wrap-around monotonic variable, once \( len \) becomes 4, it will never be 4 again, hence multiplying 5 twice is not necessary.

Figure 5.5: The dependence graph of Code 5.3
The algorithm also detects infeasible paths during this process. Suppose the update pattern is (4) in Table 5.1 but the variable’s initial value is greater than $C$, then $x == C$ is never true. In such cases, the algorithm detects infeasibility of the path and terminates the computation. This algorithm is limited to the specified patterns, but these are practically useful as the experiments show that this work could compute the sufficient BMC bound for 41 out of 46 properties.

5.4.3 Implementation

We have implemented a bound calculation tool for TLM. It is built on top of Frama-C [26], an open-source plugin-based platform where each plugin performs a specific static analysis on C code. The program dependence graph plugin is used for the input transaction code. Fig. 5.6 is an overview of our automated procedure to calculate the sufficient BMC bound. From the error vertex ($\text{assert}(0)$) of the input transaction $T$, our tool traverses the control flow backward following Code 5.4 optimized with Lemma 4. Whenever visiting an if vertex with self-modifying variables, their fitness values are determined from part B. This process is repeated until there is no more dependent vertex, the bound is infeasible, or not computable due to the special code patterns that our algorithm is not able to handle. The final result is the sufficient completeness bound for $T$ in cases when the process terminates successfully.
5.5 Unified sequential firmware-hardware model

The goal of this section is to achieve automated model checking of firmware transactions with concurrent hardware transactions.

5.5.1 Producer-consumer relationship

In Chapter 4, the producer-consumer relationship was exploited to reduce concurrent programs to sequential ones. The basic idea was as follows: to verify a firmware transaction $\mathcal{F}$, explore its producers separately, gather the reachable shared variable values, and constrain $\mathcal{F}$ with these values in a correct order. However, the previous chapter was unable to analyze stateful consumers with stateful producers, and it used a fixed constant bound for the transaction executions, thereby not exploring the complete behavior of both producers and consumers. This chapter exploits the producer-consumer relationship for sequentialization and introduces a new general sequentialization method, which handles both stateful/stateless transactions and uses complete bounds for the common cases arising from the code patterns discussed in the previous sections. This sequentialization is verification-equivalent, i.e., the correctness of the concurrent system with $\mathcal{F}$ and its interacting producers can be determined using the correctness of the sequentialized program $\mathcal{P}$, and thus can be used for model checking as well as test generation.

5.5.2 Producer analysis

Even though the producer is executed for an unbounded number of iterations, a finite sufficient bound can be determined to explore all possible values it can produce. This is then used for generating efficient constraints comprising a vector of producer values for the sequentialization. The (slightly modified) bound analysis is applied to get a sufficient bound
for vertices writing the shared variables (instead of the property violation vertex) to collect shared variable values generated by the producers.

**Initial states vs. repeating states**

Among the writes (to the shared variables) executed by a producer, some occur only a bounded number of times due to the initial stage of the transaction, and some occur infinitely often. Code [5.1] shows the effect of the initial stage. Suppose that \(I_1, I_2,\) and \(I_3\) have writes \(w_1, w_2,\) and \(w_3,\) respectively (representing different writes). \(old \) value becomes positive during the first instance. Hence, \(w_2\) is executed only once in the very first instance due to the initial value of \(old\). After the initial stage, each transaction provides a particular service repeatedly. Each execution may perform slightly different tasks but the transaction will eventually repeat the same task, or equivalently, the same write again. The following lemma describes how to find a write that occurs infinitely often.

**Lemma 5.** If a vertex \(v\) in a transaction \(T\) is reachable, it can be executed infinitely often if all \(chains(v)\) are bounded, and no source vertex of all \(chains(v)\) is in \(V_{INIT}\).

This lemma is proved by the case 1 in the proof of Lemma [2]. It is easy to find variables satisfying these conditions as they mostly have bounded-chains (in the unbounded context, most variables are modeled as bounded, such as wrap-around monotonic variables). In the infinite loop context, some states controlled by the initial state may appear only a bounded number of times, but some states will appear infinitely often. For example, in Code [5.1], \(w_1\) (\(I_1\)) is executed infinitely often.

**The vector of producer values**

The goal of this section is to generate the vector of producer values, which is bounded but has complete information of the producer using the sufficient producer bound. Since some writes occur infinitely often and some occur only finite times, this section defines the
sufficient producer bound to capture all feasible behaviors of the producer in the infinite context.

**Definition 26** (Sufficient producer bound $B_{PROD}$). *There is a trace to observe all possible producer values and each of the producer values, which occur only a bounded number of times during the initial stage within $B_{PROD}$ instances of the producer.*

For example, in Fig. 5.7, $w_1$ occurs twice, and $w_2$ and $w_3$ occur infinitely often. $B_{PROD}$ should include (1) all three writes and (2) all $w_1$ executions. After $B_{PROD}$, all writes are those that occur infinitely often. By Lemma 5, states depending on stateless, non-self-modifying, or wrap-around monotonic stateful variables can occur infinitely often if the source of their chains are not type-(a) ones. If the source of the chains are $v_{INIT}$, the corresponding producer values will be captured within the producer bound. For the writes depending on non-wrap-around monotonic variables, they either only occur bounded times until the monotonic variables reach some threshold, or occur infinitely many times once the monotonic variables reach some specific values. This producer analysis considers only monotonic self-modifying variables.

<table>
<thead>
<tr>
<th>Value</th>
<th>Repeat?</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w_1$</td>
<td>N</td>
</tr>
<tr>
<td>$w_2$</td>
<td>Y</td>
</tr>
<tr>
<td>$w_3$</td>
<td>Y</td>
</tr>
<tr>
<td>$w_1$</td>
<td>N</td>
</tr>
</tbody>
</table>

< The producer vector >

![Producer Diagram](image)

**Figure 5.7**: An example producer vector and a possible execution of the producer instances and its consumer ($\mathcal{F}$) instances
Table 5.2: Fitness table for non-wrap-around monotonic variables

<table>
<thead>
<tr>
<th>Predicate</th>
<th>Update pattern</th>
<th>Fitness value</th>
<th>Repeat</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x &gt; C, x \geq C$</td>
<td>(11) $x += A$</td>
<td>same as (7)</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>(12) $x = A$</td>
<td>$(B-C)&gt;0?\frac{(B-C)}{A}+1:\text{infeasible}$</td>
<td>NO</td>
</tr>
<tr>
<td>$x &lt; C, x \leq C$</td>
<td>(13) $x += A$</td>
<td>$(C-B)&gt;0?\frac{(C-B)}{A}+1:\text{infeasible}$</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>(14) $x = A$</td>
<td>same as (10)</td>
<td>YES</td>
</tr>
</tbody>
</table>

**Computing the producer bound:** Computing a producer bound is based on the sufficient BMC bound computation method. However, for the producer, we want to observe multiple executions of a certain vertex (e.g., $w_1$ occurs multiple times within $B_{PROD}$ in Fig. 5.7) while the BMC bound aims to observe the target instruction, or the property failure, only once. To reflect this difference, a couple of rows shown in Table 5.2 are added to Table 5.1. While analyzing a producer code with the sufficient BMC bound computation algorithm, whenever our method encounters a non-wrap-around monotonic variable, it uses Table 5.2. In case of (11) and (14), the fitness value used is the same, but the states depending on such predicates can occur arbitrarily often. For (12) and (13), different fitness values are used so that the bound includes all the writes due to the initial stage. Further, if a producer write depends only on the bounded chains but not on $v_{INIT}$, this is marked as repeating.

**Constructing bounded but complete producer vectors:** Using the bound, a producer vector is constructed: this is composed of the concrete values gathered until $B_{PROD}$ in order, plus the recurring writes that occur in instances $> B_{PROD}$. A vector rather than a set is needed to capture the order, which is relevant in how these values are used by the consumer. For each write, $B_{PROD}$ is computed using the algorithm in Code 5.4 (+ Lemma 4) and Table 5.1 extended with Table 5.2. If the write occurs infinitely often, it is marked separately. Then, our producer analysis method unrolls the producer up to the maximum sufficient producer bound to get a producer vector, as shown in Fig. 5.7. To capture the fact that the producer and consumer transactions can run at any possible speed relative to each other (in the TLM, concurrent transactions run asynchronously), it is sufficient for $\mathcal{F}$ to read from this vector in order, but not necessarily consecutively (i.e.,
some values may be skipped). Beyond $B_{PROD}$, simulating $F$ to read from any of the infinitely-occurring values can capture any scenario. In Fig. 5.7, $w2$ and $w3$ occur infinitely often, so the order between them beyond $B_{PROD}$ is not relevant.

### 5.5.3 Sequentialization for model checking

This section describes the automated model checking methodology of concurrent transactions based on the bound analysis. In Fig. 5.1, given a TLM including the target firmware transaction $F$, the model is analyzed using a previously introduced tool, the Frama-C based transaction interaction pattern analyzer [3]. This tool is used to (1) determine whether $F$ is stateful or stateless, and the list of its stateful variables, and (2) identify its producers and the shared variables. For the stateless transactions, we simply use the bound 1. Otherwise, using the structural information of $F$, its sufficient BMC bound is computed by our BMC bound analyzer. At the same time, the producers’ bounds are also determined. This information is then used by a code generator that also takes the TLM with $F$ as input. It generates a sequential program $P$ that is verification-equivalent to $F$ with the hardware transactions as its environment. The bounded model checker CBMC then runs $P$.

The key idea of our sequentialization algorithm is the following: for $F$ and its producer $Prod$ (for explanation, only one producer is used but multiple producers/shared variables are supported), instead of considering all interleavings, we analyze the effect of $Prod$ on
the shared variable $x$ and apply it while $F$ is executed, while assuming that any execution speed between $F$ and $Prod$ is possible.

Table 5.3 demonstrates the sequentialization process. The rightmost column is the generated sequential program $P$. It first concretely unrolls the instrumented $Prod$, or $Prod^w$, as many times as its statically calculated producer bond. $Prod^w$ is instrumented in such a way that every time it writes $x$, it updates the producer vector $V_x$ and marks whether the value is infinitely recurring or not. Once $P$ completes filling $V_x$, it starts to unroll $F$ as many times as $F$’s sufficient BMC bound $B_F$. The $k^{th}$ instance of $F$ is described in the first column of Table 5.3 Every time an instance of $F$ reads from $x$, it reads from $x_j$ instead. $x_j$ captures the effect of $Prod$ by the time of this read. $j$ represents each read throughout the iterations.

Once the unrolling of $F$ is done ($k$ reaches $B_F - 1$), it returns to $P$. $N_r$ is the number of total reads during this unrolling. Now $P$ extracts as many as $N_r$ values of $x$ from the producer vector, i.e., $v_j$ for $0 \leq j < N_r$, by executing (b). In (b), the sequentialization method non-deterministically increments $i$ ($i \geq 0$) and assigns $V_x[i]$ to $v_j$. If the index $i$ hits the length of the vector while $j < N_r$, $v_j$ non-deterministically chooses a value from any of the infinitely recurring values in $V_x$, or $V_x[\text{recur, *}]$. Lastly, $P$ enforces $x_j$ to be $v_j$.

This sequentialization method guarantees complete coverage of all interleavings between $F$ and $Prod$, but is limited to the strict producer-consumer relationship. As this method does not explicitly interleave between $F$ and $Prod$, the possibly exponential number of interleavings have been avoided.

The boxes shown in bold in Fig. 5.1 are what we implemented. Our code generator (written in Python) writes a customized code based on the pseudocode (Table 5.3), given the results of the three analyzers. These tools provide an automated process to generate the program for BMC.
5.6 Experiments

We used three published benchmarks [57] consisting of Linux device drivers [23] and the corresponding QEMU device emulator code [70] written in C (Table 5.4). These are significant sized benchmarks, each several KLoCs long. In each benchmark, a QEMU device model and the corresponding Linux device driver are co-analyzed. Each benchmark can be organized as a set of transactions. In all 16 transactions from the benchmark set are studied.

Some firmware transactions may not have producers, hence they can be verified by themselves. Some producers depend on other transactions. This case is not completely handled in our analysis, but the producers are treated as independent of the other transactions by allowing the corresponding shared variables to have any value, giving us over-approximated results in such cases. This may result in false positives in the model checking, i.e., some reported bugs may not be real bugs. However, these false positives will be fewer than the case where the hardware produced values are completely unconstrained. In these benchmarks, the authors formalized the firmware/hardware interaction protocols in terms of pre- and post-conditions captured as runtime assertions [57]. We built 17 additional assertions in a similar manner. Altogether, a total of 46 properties were checked in our work to show the efficacy of our bound analysis and the model checking approach for the transactions.

5.6.1 Sufficient BMC bound calculation

This section discusses the practical applicability of our bound analysis. Table 5.5 shows the given properties for which we can compute the bound in stateful transactions (the proper-
ties in stateless transactions simply have the bound 1). In Table 5.5, the first column lists the properties. The second column reports the calculated bound, and the third column shows the data-flow patterns of the variables observed during the bound calculation. Patterns (1)-(10) are shown earlier in Table 5.1. The remaining columns are discussed in Section 5.6.3.

In all, the bounds for 41 out of 46 properties are successfully computed. For the properties ETH11, 14, 17, our method could not calculate the bound, since one of the related predi-

<table>
<thead>
<tr>
<th>Prop.</th>
<th>B</th>
<th>Pattern</th>
<th>Max/Ave</th>
<th>dep(%)</th>
<th>bsf(%)</th>
<th>mono(%)</th>
<th>sl(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>I2C1</td>
<td>1</td>
<td>(1)</td>
<td>1/1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>I2C2</td>
<td>1</td>
<td>(1)</td>
<td>1/1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
</tr>
<tr>
<td>I2C3</td>
<td>2</td>
<td>(2),(4),(7)</td>
<td>2/1.66</td>
<td>0</td>
<td>25</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>I2C4</td>
<td>6</td>
<td>(1),(2),(4),(7)</td>
<td>3/2</td>
<td>0</td>
<td>25</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>I2C5</td>
<td>6</td>
<td>(2),(4),(7)</td>
<td>4/2.25</td>
<td>0</td>
<td>25</td>
<td>50</td>
<td>25</td>
</tr>
<tr>
<td>I2C6</td>
<td>2</td>
<td>(1),(9)</td>
<td>1/1</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>I2C7</td>
<td>2</td>
<td>(1),(9)</td>
<td>1/1</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>I2C8</td>
<td>4</td>
<td>(1),(7),(9)</td>
<td>3/1.31</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>I2C11</td>
<td>60</td>
<td>(1),(2),(4),(7)</td>
<td>4/2.28</td>
<td>77.78</td>
<td>20</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>I2C12</td>
<td>60</td>
<td>(1),(2),(4),(7)</td>
<td>4/2.28</td>
<td>77.78</td>
<td>20</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>I2C13</td>
<td>60</td>
<td>(1),(2),(4),(7)</td>
<td>4/2.28</td>
<td>77.78</td>
<td>20</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>I2C14</td>
<td>12</td>
<td>(2),(3),(4),(7),(9)</td>
<td>3/1.66</td>
<td>44.44</td>
<td>28.57</td>
<td>42.86</td>
<td>28.57</td>
</tr>
<tr>
<td>I2C15</td>
<td>12</td>
<td>(2),(3),(4),(7),(9)</td>
<td>3/1.66</td>
<td>44.44</td>
<td>28.57</td>
<td>42.86</td>
<td>28.57</td>
</tr>
<tr>
<td>I2C17</td>
<td>12</td>
<td>(2),(3),(4),(7)</td>
<td>3/1.66</td>
<td>44.44</td>
<td>28.57</td>
<td>42.86</td>
<td>28.57</td>
</tr>
<tr>
<td>I2C18</td>
<td>50</td>
<td>(2),(3),(4),(7),(9)</td>
<td>3/1.66</td>
<td>33</td>
<td>50</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>I2C19</td>
<td>84</td>
<td>(2),(3),(4),(7)</td>
<td>4/2.08</td>
<td>25</td>
<td>33.33</td>
<td>33.33</td>
<td>33.33</td>
</tr>
<tr>
<td>I2C20</td>
<td>4</td>
<td>(7),(9)</td>
<td>3/2</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>I2C21</td>
<td>6</td>
<td>(1),(2)</td>
<td>3/2.33</td>
<td>66.67</td>
<td>50</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>ETH1</td>
<td>2</td>
<td>(2)</td>
<td>2/2</td>
<td>0</td>
<td>33.33</td>
<td>0</td>
<td>66.66</td>
</tr>
<tr>
<td>ETH7</td>
<td>1</td>
<td>(1),(4)</td>
<td>1/1</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>ETH8</td>
<td>2</td>
<td>(2)</td>
<td>3/2.33</td>
<td>0</td>
<td>33.33</td>
<td>0</td>
<td>66.66</td>
</tr>
<tr>
<td>ETH9</td>
<td>3</td>
<td>(2)</td>
<td>3/2.33</td>
<td>100</td>
<td>33.33</td>
<td>0</td>
<td>66.66</td>
</tr>
<tr>
<td>ETH10</td>
<td>5</td>
<td>(3),(7)</td>
<td>5/3</td>
<td>0</td>
<td>0</td>
<td>22.22</td>
<td>77.78</td>
</tr>
<tr>
<td>ETH11</td>
<td>-</td>
<td>Compare with a variable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ETH12</td>
<td>2</td>
<td>(1),(3),(7)</td>
<td>1/1</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>ETH13</td>
<td>2</td>
<td>(1),(3)</td>
<td>1/1</td>
<td>0</td>
<td>0</td>
<td>25</td>
<td>75</td>
</tr>
<tr>
<td>ETH14</td>
<td>-</td>
<td>Compare with a variable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ETH15</td>
<td>121</td>
<td>(3),(9)</td>
<td>12/4.25</td>
<td>28.57</td>
<td>0</td>
<td>40</td>
<td>60</td>
</tr>
<tr>
<td>ETH16</td>
<td>-</td>
<td>Function call condition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ETH17</td>
<td>-</td>
<td>Compare with a variable</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ETH18</td>
<td>11</td>
<td>(3),(9)</td>
<td>11/6</td>
<td>0</td>
<td>0</td>
<td>50</td>
<td>50</td>
</tr>
<tr>
<td>ETH19</td>
<td>5</td>
<td>(3),(6),(7)</td>
<td>5/2.67</td>
<td>0</td>
<td>0</td>
<td>22.22</td>
<td>77.78</td>
</tr>
<tr>
<td>ETH20</td>
<td>6</td>
<td>(3),(7)</td>
<td>6/4.33</td>
<td>0</td>
<td>0</td>
<td>22.22</td>
<td>77.78</td>
</tr>
<tr>
<td>RTC1</td>
<td>9</td>
<td>(1),(2)</td>
<td>3/1.7</td>
<td>40</td>
<td>69.23</td>
<td>0</td>
<td>30.77</td>
</tr>
<tr>
<td>RTC2</td>
<td>9</td>
<td>(1),(2)</td>
<td>3/1.7</td>
<td>40</td>
<td>69.23</td>
<td>0</td>
<td>30.77</td>
</tr>
<tr>
<td>RTC3</td>
<td>2</td>
<td>(1),(2)</td>
<td>2/1.33</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>RTC4</td>
<td>3</td>
<td>(1),(2)</td>
<td>2/1.2</td>
<td>20</td>
<td>80</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>RTC5</td>
<td>2</td>
<td>(1),(2)</td>
<td>2/1.5</td>
<td>0</td>
<td>33.33</td>
<td>0</td>
<td>66.67</td>
</tr>
<tr>
<td>RTC6</td>
<td>2</td>
<td>(1),(2)</td>
<td>1/1</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>RTC7</td>
<td>2</td>
<td>(1),(2)</td>
<td>2/1.5</td>
<td>0</td>
<td>33.33</td>
<td>0</td>
<td>66.67</td>
</tr>
<tr>
<td>RTC8</td>
<td>2</td>
<td>(1),(2)</td>
<td>1/1</td>
<td>0</td>
<td>66.67</td>
<td>0</td>
<td>33.33</td>
</tr>
<tr>
<td>RTC9</td>
<td>2</td>
<td>(1),(2)</td>
<td>2/1.6/</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>RTC10</td>
<td>2</td>
<td>(1),(2)</td>
<td>2/1.6/</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>RTC11</td>
<td>2</td>
<td>(1),(2)</td>
<td>2/1.25</td>
<td>0</td>
<td>50</td>
<td>0</td>
<td>50</td>
</tr>
<tr>
<td>RTC12</td>
<td>-</td>
<td>Function call condition</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RTC13</td>
<td>4</td>
<td>(1),(2),(3)</td>
<td>4/1.83</td>
<td>0</td>
<td>40</td>
<td>20</td>
<td>40</td>
</tr>
</tbody>
</table>

Table 5.5: Bound calculation results for the given properties
Table 5.6: Model checking results of the Linux/QEMU transactions

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Bound</th>
<th>Time (sec)</th>
<th>Producers</th>
<th>Producer bound</th>
<th>Complete?</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmp105_tx</td>
<td>60</td>
<td>20.10</td>
<td>None</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>tmp105_readword</td>
<td>4</td>
<td>0.52</td>
<td>None</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>tmp105_set</td>
<td>1</td>
<td>0.69</td>
<td>tmp105_tx</td>
<td>12</td>
<td>Yes</td>
</tr>
<tr>
<td>show_temp</td>
<td>6</td>
<td>0.66</td>
<td>tmp105_tx</td>
<td>tmp105_set</td>
<td>Yes</td>
</tr>
<tr>
<td>set_temp</td>
<td>1</td>
<td>0.32</td>
<td>None</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>lm75_suspend</td>
<td>1</td>
<td>0.34</td>
<td>tmp105_tx</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>lm75_resume</td>
<td>1</td>
<td>0.34</td>
<td>tmp105_tx</td>
<td>4</td>
<td>Yes</td>
</tr>
<tr>
<td>open_eth_modem</td>
<td>3</td>
<td>491.00</td>
<td>reg_write</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>open_eth_desc_write</td>
<td>3</td>
<td>511.55</td>
<td>None</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td>open_eth_receive</td>
<td>3</td>
<td>500.65</td>
<td>open_eth_modem</td>
<td>3</td>
<td>Yes</td>
</tr>
<tr>
<td>open_eth_reg_read</td>
<td>1</td>
<td>0.91</td>
<td>reg_write</td>
<td>1</td>
<td>Yes</td>
</tr>
<tr>
<td>ethoc_rx</td>
<td>121</td>
<td>11.88</td>
<td>None</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td>ethoc_tx</td>
<td>11</td>
<td>1.47</td>
<td>None</td>
<td>-</td>
<td>No</td>
</tr>
<tr>
<td>ethoc_interrupt</td>
<td>2</td>
<td>563.87</td>
<td>ethoc_rx</td>
<td>1</td>
<td>No</td>
</tr>
<tr>
<td>cmos_write</td>
<td>9</td>
<td>593.75</td>
<td>None</td>
<td>-</td>
<td>Yes</td>
</tr>
<tr>
<td>cmos_read</td>
<td>2</td>
<td>707.91</td>
<td>cmos_write</td>
<td>1</td>
<td>Yes</td>
</tr>
</tbody>
</table>

cates compares a monotonic variable with a non-constant variable. For ETH16, RTC12, our tool did not recognize some complex conditional branches. Some computed bounds in stateful transactions are only 1 because they are only controlled by stateless variables, or the computed fitness value is 0 by Table 5.1.

5.6.2 Automated model checking

Table 5.6 shows the result of model checking the benchmark transactions. The first two columns are the target transactions with the properties in Table 5.5 and their computed bounds. The fourth and fifth columns are the target transaction’s producers and their bounds. The third column is the time taken to model check the transactions. The last column shows whether this model checking result is complete or not, based on our ability to compute the bound. When the bound analysis could not compute the bounds for some properties in a transaction, this experiment used the maximum bound of other properties defined in that transaction. Therefore, results for the open_eth_desc_write, ethoc_rx, and ethoc_tx transactions are incomplete for that reason, i.e., our model checking method may miss a counterexample longer than the used bounds.
Some transactions take a relatively long time even with small bounds because they involve bound
ed but big-sized loops inside the transaction. Model checking for many transactions
was done within a second. Horn et al. [57] performed model checking for the same bench-
mark set using CBMC, but they took hundreds of seconds for most properties. On the
other hand, our work verified much smaller units (transactions) separately with their pro-
ducers. Hence, it is not possible to directly compare the results, but this shows that our
compositional methodology of verifying small units can be significantly faster in addition
to guaranteeing completeness in many cases.

5.6.3 Discussion

Table 5.5 provides several statistics in columns 4-8. For each property \( v \), \( Max \) and \( Ave \)
are the maximum and average lengths of the chains in \( all \_chains(v) \), respectively (for un-
bounded chains, the fitness values are used for the chain lengths). \( dep \) is the percentage of
dependent chains among the chains in \( all \_chains(v) \). \( bsf \), \( mono \), and \( sl \) are the percentage
of stateful variables with bounded chains, monotonic variables, and stateless variables the
property depends on. Note that 95% of the average chain lengths are within 3. Dependent
chains (which are responsible for the product term of \( B(v) \)) are often short and keep the
final bound from exploding. For monotonic variables, the chain length can be relatively
long, but the chains are usually independent of other chains. Further, branches with only
stateless variables or bounded stateful variables occur quite often in these real world bench-
marks. For these reasons, the computed bounds were mostly small.

We used CBMC in this work, but other model checkers or even test generation tools could
be used for finding property violations within the statically determined bounds. However,
we expect model checkers are more suitable for this work as test generation tools (even
those based on symbolic execution) may enumerate too many paths, which are considered
implicitly by a model checker.
5.6.4 Comparison with the partial order reduction method

Our automated model checking method consists of the bound analysis and the sequentialization method. The bound analysis enables complete and scalable sequentialization of concurrent systems for specific interaction patterns. There is no other bounded model checking tool that computes such sufficient bounds, and thus a direct comparison with other methods is not possible. Nevertheless, this section reports the number of states due to interleavings our sequentialization method saves compared to the partial order reduction technique.

For the partial order reduction technique, we used the model checking tool Verisoft [42]. Verisoft systematically explores the state space of concurrent systems using partial order reduction. The state space of a system refers to a directed graph that represents the combined behavior of all the concurrent components of the system. In this graph, a path corresponds to a sequence of operations that can be observed during the system executions. During model checking, Verisoft searches for coordination problems, such as deadlocks or divergences between concurrent components, and for violations of the user-specified assertions. The tool supports nondeterminism for modeling the environment of the target system. As a result of model checking, Verisoft reports the number of states (the nodes of the graph) and transitions (the edges of the graph) explored during partial order reduction, and the time taken. The states here are global states between concurrent processes. The transitions are the possible transitions between these global states.

Since we have a sufficient bound for the unbounded transactions, we could use BMC for model checking (Table 5.6). For Verisoft, the while(1) loops are used as the outer loops of the input transactions. Instead, Verisoft bounds the depth of the search and the delay, as specified by users. The depth in the state space is defined by the number of read/write operations on the shared variables between the concurrent threads executed from the initial global state. The delay is the time given to any process to reach its next operation on the
shared variables. In this experiment, we used 100 for the maximum depth of the search, 5 for the depth of one layer of breadth-first search (BFS), and 10 seconds for the delay. Note that the depth of one layer of BFS is different from the depth of the search. For example, if the depth of one layer of BFS is set to 5, Verisoft will start to explore all the possible paths of length 5 in the state space, then it will explore the paths up to length 10, and so on until it reaches the maximum depth 100. Hence, Verisoft only guarantees a complete coverage of the state space up to some depth.

Table 5.7 provides the model checking results of our unbounded benchmark transactions with their concurrent producers using Verisoft. The results indicate the number of global states we could save. Even if we bounded the depth of the search and the delay, Verisoft model checking process often stopped without completing verification, as it reached the maximum size of the result file of the explored states (10MB) in the \texttt{tmp105_set}, \texttt{show\_temp}, \texttt{open\_eth\_receive}, or \texttt{cmos\_read} cases. We performed two sets of model checking using Verisoft with 10MB and 20MB as the maximum size of the result files. In Table 5.7, columns 2 and 3 show the results with the 10MB files, and columns 4 and 5 show the results with the 20MB files.

For some transactions, such as \texttt{lm75\_suspend}, \texttt{lm75\_resume}, \texttt{open\_eth\_reg\_read}, or \texttt{ethoc\_interrupt}, the number of explored states were very small (note that these states are the number of the explored global states, not including the local ones). This is because the correctness properties to prove are simple expressions with only few shared variables.

Table 5.7: Model checking results of the Linux/QEMU transactions using Verisoft with unbounded outer loop executions

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Number of explored global states (10MB)</th>
<th>Time (sec) (10MB)</th>
<th>Number of explored global states (20MB)</th>
<th>Time (sec) (20MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmp105_set</td>
<td>410000</td>
<td>14559.13</td>
<td>410000</td>
<td>12479.71</td>
</tr>
<tr>
<td>show_temp</td>
<td>85716</td>
<td>965.41</td>
<td>178661</td>
<td>2327.92</td>
</tr>
<tr>
<td>lm75_suspend</td>
<td>5</td>
<td>0.63</td>
<td>5</td>
<td>0.59</td>
</tr>
<tr>
<td>lm75_resume</td>
<td>4</td>
<td>0.63</td>
<td>4</td>
<td>0.58</td>
</tr>
<tr>
<td>open_eth_receive</td>
<td>5006</td>
<td>147.55</td>
<td>5006</td>
<td>193.42</td>
</tr>
<tr>
<td>open_eth_reg_read</td>
<td>2</td>
<td>0.11</td>
<td>2</td>
<td>0.62</td>
</tr>
<tr>
<td>ethoc_interrupt</td>
<td>4</td>
<td>10.57</td>
<td>4</td>
<td>10.94</td>
</tr>
<tr>
<td>cmos_read</td>
<td>9023</td>
<td>71.44</td>
<td>9023</td>
<td>81.52</td>
</tr>
</tbody>
</table>

134
Our BMC-based model checking methodology, on the other hand, implicitly includes all feasible behaviors of the shared variables. For $lm75\_suspend$, $lm75\_resume$, and $open\_eth\_reg\_read$, our model checking results (Table 5.6) also show very short execution times, but the model checking result of $ethoc\_interrupt$ shows the partial order reduction technique can be more effective than our method if the shared variables have a small impact on the correctness properties. We do not report the number of transitions in Table 5.7 as Verisoft does not provide them when it is forced to quit due to lack of space.

Table 5.7 demonstrates the state explosion due to interleavings even when using partial order reduction for practical sized programs. Even if we bound the transactions using our sufficient BMC bounds during the Verisoft model checking process, the number of states explored is still large, and thus a significant amount of storage is required. This adjusted experiment result (with 10MB as the maximum size of the result file) is provided in Table 5.8. Some transactions, such as $tmp105\_set$ or $show\_temp$, show a large number of explored states due to interleavings. Although a direct comparison of Tables 5.8 and 5.6 is not fair, some transactions, such as $open\_eth\_receive$ or $ethoc\_interrupt$, show shorter verification time using Verisoft (Table 5.8) than our model checking method (Table 5.6).

The main reason is that while the small sufficient BMC bounds computed by our bound analysis are used for both experiments, our method does not bound all the loops inside the main transaction loop (some of the results in Table 5.6 report long verification time because of the loops inside). On the other hand, Verisoft bounds the depth of the search for these loops, and thus the result is not complete.

To sum up, the partial order reduction method requires storage of a large number of states, and often this is not feasible with practical examples. Furthermore, the Verisoft result is not complete due to the user-defined bounds for the depth of the search and the delay. On the other hand, our method is able to handle most practical benchmarks and ensures the
Table 5.8: Model checking results of the Linux/QEMU transactions using Verisoft with bounded outer loop executions

<table>
<thead>
<tr>
<th>Transaction</th>
<th>Number of explored global states (10MB)</th>
<th>Number of explored global transitions</th>
<th>Time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>tmp105_set</td>
<td>300583</td>
<td>30614884</td>
<td>6125.11</td>
</tr>
<tr>
<td>show_temp</td>
<td>41530</td>
<td>1906198</td>
<td>299.77</td>
</tr>
<tr>
<td>lm7/suspend</td>
<td>5</td>
<td>10506</td>
<td>0.61</td>
</tr>
<tr>
<td>lm7/resume</td>
<td>4</td>
<td>404</td>
<td>0.61</td>
</tr>
<tr>
<td>open_eth_receive</td>
<td>1799</td>
<td>36000000</td>
<td>69.36</td>
</tr>
<tr>
<td>open_eth_reg_read</td>
<td>2</td>
<td>202</td>
<td>0.09</td>
</tr>
<tr>
<td>ethoc_interrupt</td>
<td>4</td>
<td>107</td>
<td>10.16</td>
</tr>
<tr>
<td>cmos_read</td>
<td>1006</td>
<td>207712</td>
<td>13.41</td>
</tr>
</tbody>
</table>

completeness of the verification results as long as the sufficient BMC bound is successfully computed.

5.7 Related work

This section introduces previous approaches to the loop bound analysis and the sequentialization techniques for verifying concurrent programs.

5.7.1 Loop bound analysis

Our analysis can be viewed as finding the break condition, i.e., the property violation, of the infinite loops. There is prior work on loop analysis, which aims to accelerate determining loop break conditions. In symbolic execution, the presence of loops with fairly small sizes can result in a large number of test cases. To prevent this, Xie et al. [106] introduced Fitnex, a method for guided path search using the fitness function. This work exploited the fitness function to summarize the effect of monotonic variables towards the property violation. Godefroid et al. [46] used pattern matching rules similar to our work on the loop guards for loop summarization but limited to a single loop exit condition and induction variables. However, these loop analysis techniques assume a single predicate for a loop break. Our work, on the other hand, is able to analyze nested branches. Healy et al. [49] provided timing analysis for loops with nested predicates that can affect the loop
exits as well. They compute the range of iterations taken for each basic block of the control flow graph to calculate the minimum/maximum number of iterations associated with each loop. In contrast, our work covers a broader range of controlling variables using variable lifetime information.

5.7.2 Verifying concurrent programs

Our treatment of the shared variables has some similarity to the sequentialization work by Lal et al. [68]. However, their analysis over-approximates variable values for a bounded number of context switches. Our granularity of interactions is much finer at the level of shared variable accesses, which implicitly covers many more interleavings. Some recent works provide sequentialization by extracting the relationship between concurrent threads and using it as a constraint to simulate the concurrency in symbolic execution. Chaki et al. [15, 16] non-deterministically schedule concurrent periodic tasks using constraints between the concurrent threads. These constraints reflect relationship between inter- or intra-tasks similar to ours, but they use timing information while we use dataflow information.

5.8 Chapter summary

Firmware verification is challenging as it needs to consider the concurrency between firmware and its interacting hardware/firmware components. This is compounded by the often non-terminating nature of firmware and hardware components. This chapter shows how commonly occurring code patterns can be used to statically calculate the completeness bounds for use with BMC of firmware properties. The bound information, combined with the interaction patterns between firmware/hardware transactions, is used to sequentialize the target firmware transaction $F$ and its interacting hardware transaction.
to a verification-equivalent single threaded program $\mathcal{P}$. The sequentialization algorithm handles many common interactions patterns seen in practice and the bound (when successfully computed) is guaranteed to check sufficient iterations of both transactions while avoiding expensive exploration of their interleavings. This sequentialized program $\mathcal{P}$ is then verified using BMC. The efficacy of this methodology is demonstrated on 46 properties in 16 firmware transactions in three published benchmarks of the Linux device drivers with the QEMU emulator code for the interacting hardware components.
Chapter 6

Conclusion

This chapter summarizes the contributions of this thesis, and it then describes future research directions. This thesis makes various contributions to the state of the art of firmware verification and concurrent firmware-hardware model analysis techniques. With the growing need for inexpensive and scalable methods to analyze concurrent firmware systems, our research has focused on the development of pattern-based analysis techniques for firmware systems while addressing complete coverage and scalability.

At a high level, our contributions are two-fold. First, we introduce the service function-based TLM, a unified modeling framework for firmware and its interacting hardware at the transaction level. Second, we propose a set of new techniques to sequentialize concurrent firmware TLMs on standard single-threaded verification techniques can be applied. Previous concurrent analysis approaches are either complete but computationally expensive, or scalable but incomplete. Instead, we use the important classes of interaction patterns of firmware and hardware transactions in the well-structured TLM modeling framework for the analysis. As a result, our techniques are scalable and complete at the same time for the set of input programs with the specific interaction patterns. Experimental results have been provided to demonstrate (1) the prevalence of these patterns in practice, and (2) the efficacy
of these techniques using real firmware benchmarks, such as Linux device driver code and its interacting QEMU emulated hardware code.

Overall, this thesis makes the following contributions:

- Chapter 3 applies high-level functional modeling techniques to firmware-hardware modeling and presents a novel service function-based TLM for the co-design of firmware and its interacting hardware components. This is a distinctive modeling technique compared to previous TLM designs as it is based on the service functions, which provide an intuitive way of modeling firmware/hardware functionalities at the specification level. This unified modeling methodology allows firmware/hardware co-verification in the early design stage while avoiding a laborious co-simulation process. Moreover, the well-defined structure of the TLMs naturally enable a useful characterization of firmware-hardware interactions for firmware analysis. This chapter provides in-depth analysis of several firmware-hardware interaction patterns that are commonly observed in real-world firmware examples.

- Chapter 4 shows how the interaction patterns enable the use of a single-threaded concolic testing framework to generate a complete test set for a firmware transaction even when it is interacting with other concurrent firmware/hardware transactions. This chapter presents frameworks for the interaction-pattern-specific customized algorithms, that sequentialize the target firmware transaction and its interacting firmware/hardware transactions. This pattern-based sequentialization method fills a crucial gap in scalability. Previous approaches for analyzing concurrent systems are computationally expensive and impractical for guaranteeing complete coverage. Our algorithms avoid an explicit exploration of asynchronous interleavings of the concurrent transactions. Instead, the interactions between firmware-hardware transactions are implicitly captured by a set of constraints. This work covers the most common interaction patterns seen in practice. Also, the testing generation al-
gorithms guarantee the minimum number of iterations needed to cover all feasible paths or alternatively determine the path coverage for a fixed number of iterations.

Further, this chapter provides a fully automated solution that builds on our modified versions of publicly available static analysis and concolic testing tools. First, we demonstrate automatic detection of specific interaction patterns of interest using a static analyzer based on Frama-C. We also implemented the code generator, which uses the captured interaction patterns to automatically compose a sequential program. This sequential program, which is test-equivalent to the target transaction, can then be used with a standard sequential concolic testing tool.

- Chapter 5 presents a new technique to determine a sufficient BMC bound to prove the property or to find a violation. The verification method in Chapter 4 is able to decide if the verification results of unbounded TLM models are complete or not, only within a given bound. The completeness of the results is guaranteed only when the bound is sufficient, but there is no way to decide the sufficient bound. This chapter fills this gap by providing termination checks for bounded model checking of unbounded models. This is an inexpensive static analysis approach that exploits characteristics of common code patterns found in firmware.

By combining this bound analysis with the sequentialization technique introduced in Chapter 4, Chapter 5 presents a new general sequentialization technique. This new method improves the previous sequentialization technique in the following ways: (1) it is a unified framework covering all common interaction patterns reported while the previous method misses the case where both the target transaction and its producers are stateful, and (2) it guarantees complete coverage when completeness bounds can be successfully computed. For the certain (but common) set of input transactions, our sequentialization method achieves both scalability and completeness at the same time.
There are many interesting possible ways in which the work in this thesis can be further extended.

First, in Chapter 4, the resulting number of test cases in the experiments (Table 4.5) are sometimes substantial. This is due to the way symbolic analysis handles bounded loops inside transactions. Concolic testing tools may explore extremely large (possibly infinite) number of paths even for a single loop when the loop condition involves some unbounded input. To solve this problem, there have been many heuristics. For example, the testing tool SAGE [45] uses counters to limit the number of constraints that can be generated from a branch condition. This heuristic effectively reduces the search space in an unsound manner, i.e., may miss bugs. Godefroid et al. [46] presented a different approach based on loop-invariant generation that enables loop summarization. Applying such works to our testing method can improve the efficiency of our method.

Second, our static bound analysis can be applied to achieve full path coverage. The bound analysis is to determine sufficient bounds for safety properties, but it is not able to decide bounds for complete path coverage of input programs. Our work so far only determines if the complete path coverage is achieved with a given bound. I believe that the sufficient BMC bound analysis can be useful to tackle this problem.

Lastly, while the stateful/stateless patterns cover the complete cases together, our producer-consumer relationship is strictly defined. This work is limited to the simple producer-consumer relationship: we handle only the cases where there is no transaction in the middle between the target transaction and the producers, i.e., the produced values are directly read by the consumers. I believe various complicated producer-consumer relationships can be explored further.
Appendix A

List of Publications


Bibliography


[51] Claude Helmstetter, Florence Maraninchi, and Laurent Maillet-Contoz. Full simulation coverage for SystemC transaction-level models of systems-on-a-chip. *Pro-


