Bbs: A Phase-Bounded Model Checker for Asynchronous Programs

Rupak Majumdar and Zilong Wang(✉)

MPI-SWS, Kaiserslautern, Germany
{rupak,zilong}@mpi-sws.org

Abstract. A popular model of asynchronous programming consists of a single-threaded worker process interacting with a task queue. In each step of such a program, the worker takes a task from the queue and executes its code atomically to completion. Executing a task can call "normal" functions as well as post additional asynchronous tasks to the queue. Additionally, tasks can be posted to the queue by the environment.

Bouajjani and Emmi introduced *phase-bounding* analysis on asynchronous programs with unbounded FIFO task queues, which is a systematic exploration of all program behaviors up to a fixed *task phase*. They showed that phase-bounded exploration can be sequentialized: given a set of recursive tasks, a task queue, and a phase bound $L > 0$, one can construct a sequential recursive program whose behaviors capture all states of the original asynchronous program reachable by an execution where only tasks up to phase L are executed. However, there was no empirical evaluation of the method.

We describe our tool Bbs that implements phase-bounding to analyze embedded C programs generated from TinyOS applications, which are widely used in wireless sensor networks. Our empirical results indicate that a variety of subtle safety-violation bugs are manifested within a small phase bound (3 in most of the cases). While our evaluation focuses on TinyOS, our tool is generic, and can be ported to other platforms that employ a similar programming model.

1 Introduction

In many asynchronous applications, a single-threaded worker process interacts with a task queue. In each scheduling step of these programs, the worker takes a task from the queue and executes its code atomically to completion. Executing a task can call "normal" functions as well as post additional asynchronous tasks to the queue. Additionally, tasks can be posted to the queue by the environment. This basic concurrency model has been used in many different settings: in low-level server and networking code, in embedded code and sensor networks [6], in smartphone programming environments such as Android or iOS, and in Javascript. While the concurrency model enables the development of responsive applications, interactions between tasks and the environment can give rise to subtle bugs.

D. Kroening and C.S. Păsăreanu (Eds.): CAV 2015, Part I, LNCS 9206, pp. 496–503, 2015.
DOI: 10.1007/978-3-319-21690-4_33

Bouajjani and Emmi introduced *phase-bounding* [1]: a bounded systematic search for asynchronous programs that explores all program behaviors up to a certain phase of asynchronous tasks. Intuitively, the *phase* of a task is defined as its depth in the task tree: the main task has phase 1, and each task posted asynchronously by a task at phase i has phase $i + 1$. Their main result is a sequentialization procedure for asynchronous programs for a given fixed bound L on the task phase.

In this paper, we describe our tool Bbs[1] that implements phase-bounding to analyze C programs generated from TinyOS applications, which are widely used in wireless sensor networks. Our empirical results indicate that a variety of subtle memory-violation bugs are manifested within a small phase bound (3 in most of the cases). From our evaluation, we conclude that phase-bounding is an effective approach in bug finding for asynchronous programs.

While our evaluation focuses on TinyOS, our tool is generic, and can be ported to other platforms that employ a similar programming model. We leave certain extensions, such as handling multiple worker threads, and the experimental evaluation of this technique to other domains, such as smartphone applications or Javascript programs, for future work.

2 Sequentialization Overview

We now give an informal overview of Bouajjani and Emmi's sequentialization procedure. Given an asynchronous program, we first perform the following simple transformation to reduce assertion checking to checking if a global bit is set: (1) we add a global Boolean variable `gError` whose initial value is *false*; (2) we replace each assertion **assert**(e) by `gError` = !e; **if**(`gError`) **return**; and (3) we add **if**(`gError`) **return**; at the beginning of each task's body and after each procedure call. The translation ensures that an assertion fails iff `gError` is *true* at the end of `main`.

Intuitively, the sequentialization replaces asynchronous posts with "normal" function calls. These function calls carry an additional parameter that specifies the phase of the call: the phase of a call corresponding to an asynchronous post is one more than the phase of the caller. The sequentialization maintains several versions of the global state, one for each phase, and calls the task on the copy of the global state at its phase. The task can immediately execute on that global state, without messing up the global state at the posting task's phase. Since tasks are executed in FIFO order, notice that when two tasks t_1 and t_2 are posted sequentially (at phase i, say), the global state after running t_1 is exactly the global state at which t_2 starts executing. Thus, the copy of the global state at phase i correctly threads the global state for all tasks executing at phase i.

The remaining complication is connecting the various copies of the global state. For example, the global state when phase i starts is the same as the global state at the *end* of executing phase $i - 1$, but we do not know what that state is

[1] Bbs stands for *Buffer phase-Bounded Sequentializer* and can be downloaded at https://github.com/zilongwang/bbs.

(without executing phase $i-1$ first). Here, we use non-determinism. We guess the initial values of the global state for each phase at the beginning of the execution. At the end of the execution, we check that our guess was correct, using the then available values of the global states at each phase. If the guess was correct, we check if some copy of `gError` is set to true: this would imply a semantically consistent run that had an assertion failure.

We now make the translation a bit more precise. Given a phase bound $L \in \mathbb{N}$, i.e., the maximal number of phases to explore, the sequentialization consists of four steps:

1. Track the phase of tasks at which they run in an execution. Intuitively, the phase of `main`, the initial task, is 1, and if a task at phase i executes **post** $p(e)$, then the new task p is at phase $i+1$. As an example, consider an error trace in Fig. 1, task t_0 is at phase 1, and tasks t_1, t_2 are at phase 2. This tracking can be done by augmenting each procedure's parameter list with an integer k that tracks the phase of the procedure. Consequently, we also replace each normal synchronous procedure call $p(e)$ by $p(e,k)$, and each asynchronous call **post** $p(e)$ by **post** $p(e,k+1)$.
2. Replace each **post** $p(e,k+1)$ by **if**$(k < L)$ $p(e,k+1)$;, meaning that if some task at phase k posts the task p and $k+1$ does not exceed the phase bound L, the task p is immediately called and runs at phase $k+1$ instead of putting it into the task queue.
3. For each global variable g, create L copies of it, denoted by $g[1], \ldots, g[L]$. Set the initial value of the first copy $g[1]$ to the initial value of g, and nondeterministically guess the initial values of the other copies. For each statement of a program, if g appears, then replace it by $g[k]$. Intuitively, the i-th copy of global variables is used to record the evolution of global valuations along an execution at phase i.
4. Run the initial task t_0 at phase 1. When t_0 returns, for each phase $i \in [2, L]$, enforce that the guessed initial values of the i-th copy are indeed equal to the final values of the $(i-1)$-th copy. Finally, a bug is found if some copy of `gError` equals *true*.

Step 4 is better explained through an example. We present how a sequentialized execution in Fig. 2 is related to an error trace of Fig. 1. Suppose that the phase bound $L = 2$ and the above first three steps have been done correctly.

Consider segment (a) in Fig. 2 and segment (1) in Fig. 1. When task t_0 starts, notice that the global state x in segment (1) and its first copy $x[1]$ in segment (a) are always the same because both are initialized to v_0, and in each step of their executions, the way that segment (1) modifies x is the same as the way that segment (a) modifies $x[1]$. In this case, we say that segment (a) uses the first copy of the global state to "mimic" the evolution of the global state in segment (1).

Since the last statement of segment (a) is **if**$(k < L)$ $p(e,k+1)$; and the current phase $k = 1$, segment (b) starts. Notice that segment (b) runs at phase 2 and only modifies the second copy of the global state $x[2]$. Additionally, if

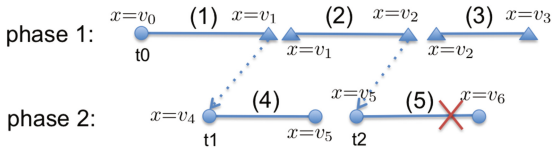

Fig. 1. An error trace before sequentialization. Circles denote the starting or ending points of tasks. Solid lines denote the execution of tasks. Triangles with dashed arrows indicate a post statement that posts a task to the queue; triangles without dashed arrows are statements right after post statements. The cross represents where the assertion fails. This error trace is read as follows: task t_0 runs, posts tasks t_1 and t_2 to the task queue, and completes. Then t_1 and t_2 runs one after another. We divided the error trace into execution segments (1)–(5), ordered by their execution order. Values of the global state x at each segment are shown. E.g., when segment (1) starts and ends, $x = v_0$ and $x = v_1$, respectively. When segment (4) starts and ends, $x = v_4$ and $x = v_5$, respectively. Note that due to the FIFO order, $v_3 = v_4$.

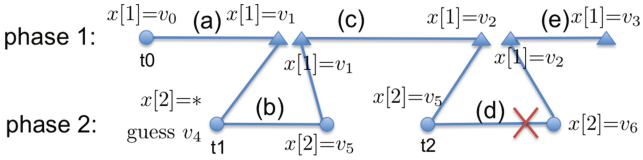

Fig. 2. The sequentialized error trace after sequentialization. Values of each copy of the global state x at each segment are shown. E.g., when segment (a) starts and ends, the first copy $x[1] = v_0$ and $x[1] = v_1$, respectively. When segment (b) starts, the second copy $x[2]$ is guessed to v_4. When segment (b) ends, $x[2] = v_5$.

we assume that the initial value of $x[2]$ are guessed correctly, i.e., v_4, shown in Fig. 2, then segment (b) uses the second copy of the global state to "mimic" the evolution of the global state in segment (4).

After segment (b) completes, the control goes back to phase 1 and segment (c) starts. Note that segment (b) does not modify the first copy x[1], and hence when segment (c) starts, the value of $x[1]$ is still v_1. As a result, segment (c) uses the first copy of the global state to "mimic" the evolution of the global state in segment (2).

After segment (c) completes, segment (d) starts. Note that since segment (c) does not modify the second copy $x[2]$, the value of $x[2]$ is still v_5 at the beginning of segment (d), which is the same as the value of x at the beginning of segment (5). Hence segment (d) uses $x[2]$ to mimic x in segment (5). When segment (d) completes, segment (e) starts to use the first copy $x[1]$ to mimic segment (3).

Finally, When segment (e) completes, by using `assume` statements, we enforce that the initial value for the second copy $x[2]$ is indeed guessed to v_4 in order to satisfy the FIFO order imposed by the task queue. After the enforcement, the sequential execution in Fig. 2 and the error trace in Fig. 1 reach exactly the same set of global states. Hence we conclude that a bug is found.

3 Experimental Evaluation

We first provide a brief introduction to TinyOS applications. We then present the design of Bbs and elaborate on our experimental results.

3.1 TinyOS Execution Model

TinyOS [7] is a popular operating system designed for wireless sensor networks. It uses nesC [6] as the programming language and provides a toolchain that translates nesC programs into embedded C code and then compiles the C code into executables which are deployed on sensor motes to perform operations such as data collection.

TinyOS provides a programming language (nesC) and an execution model tailored towards asynchronous programming. A nesC program consists of tasks and interrupt handlers. When the program runs, TinyOS associates a scheduler, a stack, and a task queue with it, and starts to run the "main" task on the stack. Tasks run to completion and can post additional tasks into the task queue. When a task completes, the scheduler dequeues the first task from the task queue, and runs it on the stack.

Hardware interrupts may arrive at any time (when the corresponding interrupt is enabled). For instance, a timer interrupt may occur periodically so that sensors can read meters, or a receive interrupt may occur to notice sensors that packets arrived from outside. When an (enabled) interrupt occurs, TinyOS preempts the running task and executes the corresponding interrupt handler defined in the nesC program. An interrupt handler can also post tasks to the task queue, which is used as a mechanism to achieve deferred computation and hide the latency of time-consuming operations such as I/O. Once the interrupt handler completes, the interrupted task resumes.

3.2 Bbs Overview

We implemented Bbs to perform phase-bounded analysis for TinyOS applications. Bbs checks user-defined assertions as well as two common memory violations in C programs: out-of-bound array accesses (OOB) and null-pointer dereference.

The workflow of Bbs is shown in Fig. 3. First, given a TinyOS application consisting of nesC files, the nesC compiler `nescc` combines them together and generates a self-contained embedded C file. `nescc` supports many mote platforms and generates different embedded C code based on platforms. In our work, we let `nescc` generate embedded C code for MSP430 platforms.

Bbs takes as inputs the MSP430 embedded C file containing assertions and a phase bound, and executes three modules.

The first module performs preprocessing and static analysis on the C program to instrument interrupts and assertions. Interrupt handlers are obtained from `nescc`-generated attributes in the code. A naive way to instrument interrupts is to insert them before each statement of the C program. However, if a

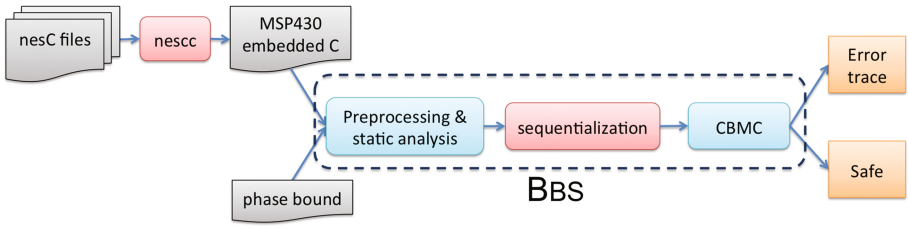

Fig. 3. The workflow of BBS

statement does not have potentially raced variables[2], we do not need to instrument interrupts before it, because the execution of such statements commutes with the interrupt handler: either order of execution leads to the same final state. Thus BBS performs static analysis to compute potentially raced variables and instruments interrupts accordingly.

The second module implements the sequentialization algorithm. The resulting sequential C program is fed into the bounded model checker CBMC [3,4], which outputs either an error trace or "program safe" up to the phase bound and the bound imposed by CBMC.

3.3 Experimental Experience with BBS

We used BBS to analyze eight TinyOS applications in the `apps` directory from TinyOS's source tree. These benchmarks cover most of the basic functionalities provided by a sensor mote such as timers, radio communication, and serial transmission.

In Table 1, we summarize the size and complexity of these benchmarks in terms of (1) lines of code in the cleanly reformatted ANSI C program after the preprocessing stage, (2) the number of types of tasks that can be posted, (3) the number of types of hardware interrupts that are expected, (4) the number of global variables as well as the number of potentially raced variables (found by the static analysis).

In each of the first three benchmarks, we manually injected a realistic memory violation bug that programmers often make. The rest five benchmarks were previously known to be buggy [2,5,8,9]. The TestSerial benchmark contains two bugs and each of the rest has one bug. We ran BBS on these benchmarks to see whether it could find these bugs efficiently within small phase bounds.

Experimental Results. All experiments were performed on a 2 core Intel Xeon X5650 CPU machine with 64GB memory and 64bit Linux (Debian/Lenny). Table 2 lists the analysis results, showing that BBS successfully uncovered all bugs that are injected in the first three benchmarks, as well as all previously known bugs in the rest five benchmarks. We report the type of bugs, the minimal

[2] A potentially raced variable is accessed by both tasks and interrupt handlers, and at least one access from both is a write.

Table 1. TinyOS benchmarks

Benchmark	LOC	Tasks	Interrupt types	Global variables	Potentially raced global variables
TestAdc	6738	9	2	100	19
TestEui	7467	13	3	138	17
TestAM	11259	13	5	154	27
BlinkFail	3153	3	1	64	5
TestSerial	6590	10	3	127	17
TestPrintf	6882	13	3	136	18
TestDissemination	13004	17	5	166	37
TestDip	17091	25	7	243	49

Table 2. Experimental results

Benchmark	Bug type	Min phase	Time		Error trace
			Seq. (s)	CBMC (s)	(in steps)
TestAdc	NullPtr	2	3.92	15.92	2014
TestEui	OOB	2	3.97	12.78	9425
TestAM	NullPtr	3	5.88	342.99	12925
BlinkFail	OOB	3	2.55	2.69	3773
TestSerial	OOB	4	3.75	23.92	13531
	User-defined	4		39.01	14161
TestPrintf	OOB	3	3.78	30.32	14154
TestDissemination	NullPtr	3	5.95	843.68	17307
TestDip	NullPtr	3	7.69	681.81	20274

phases that are required to uncover the bugs, the time used in both sequentialization and CBMC, and the lengths of error traces. Notice that all bugs were found within small phase bounds, that is, at most 4 phases. This result indicates that the phase-bounded approach effectively uncovers interesting bugs within small phase bounds for realistic C programs.

References

1. Bouajjani, A., Emmi, M.: Bounded phase analysis of message-passing programs. STTT **16**(2), 127–146 (2014)
2. Bucur, D., Kwiatkowska, M.Z.: Software verification for TinyOS. In: Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN 2010, pp. 400–401, ACM (2010)
3. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)

4. Clarke, E., Kroening, D., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: Proceedings of the 40th Annual Design Automation Conference, DAC 2003, pp. 368–371. ACM, New York, NY, USA (2003)
5. Cooprider, N., Archer, W., Eide, E., Gay, D., Regehr, J.: Efficient memory safety for TinyOS. In: Proceedings of the 5th International Conference on Embedded Networked Sensor Systems, SenSys 2007, pp. 205–218. ACM, New York, NY, USA (2007)
6. Gay, D., Levis, P., von Behren, R., Welsh, M., Brewer, E., Culler, D.: The nesC language: a holistic approach to networked embedded systems. In: Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, PLDI 2003, pp. 1–11. ACM, New York, NY, USA (2003)
7. Hill, J., Szewczyk, R., Woo, A., Hollar, S., Culler, D., Pister, K.: System architecture directions for networked sensors. In: Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS IX, pp. 93–104. ACM, New York, NY, USA (2000)
8. Li, P., Regehr, J.: T-check: bug finding for sensor networks. In: Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks, IPSN 2010, pp. 174–185. ACM, New York, NY, USA (2010)
9. Safe TinyOS. http://docs.tinyos.net/index.php/Safe_TinyOS