Contract-Based Verification of Timing Enforcers
[Extended Abstract]

Sagar Chaki
chaki@sei.cmu.edu

Dionisio de Niz
dionisio@sei.cmu.edu

ABSTRACT

A timing enforcer not only allocates CPU cycles to threads but also uses timers to enforce time budgets. An approach for verifying safety properties of timing enforcers at the source code level is presented. We assume that the enforcer is implemented as a set of entry functions that are executed atomically on critical system-level events, such as arrival and departure of periodic jobs. The key idea is to express the safety property as an invariant, and prove that it is inductive across all the entry functions. The approach is validated by proving correctness of the enforcement of CPU cycle budgets for tasks by a mixed-criticality scheduler called ZSRM that is implemented in C. The inductiveness of the necessary ZSRM invariants is proved by expressing them as function contracts using the ACSL specification language, and verifying the contracts using the FRAMA-C tool.

Keywords

Contracts; Deductive Verification; Real-Time Scheduling

1. INTRODUCTION

We focus on the verification of a class of software that rely on the use of timers and clocks to implement their functionality. We call them Software with Timers and Clocks (STACs). STACs are at the heart of many real-time embedded software that perform safety-critical functions in systems we use in our daily lives (e.g., automobiles). A prime example is a real-time thread scheduler. Verifying their correct behavior is therefore very important.

This challenge is complicated by several factors. First, the semantics of STACs involve time. Timers and clocks are implemented by an OS sub-system that keeps track of time using special hardware (e.g., High Precision Event Timer and the time stamp counters). This must be modeled soundly. Second, STACs can behave incorrectly due to numeric overflows, which can violate monotonicity of timestamps maintained in variables with finite program types (such as `int`). Many software verification tools treat finite types as unbounded ones (e.g., `int`s as integers) which is unsound for detecting numeric overflows. Finally, STACs use pointers and memory operations substantially since they are often high-performance system software. For example, a scheduler typically maintains active threads as a linked list which is dynamically updated as threads are created and destroyed. Verifying correctness of pointer-manipulating programs is known to be challenging for software verification tools.

In this paper, we propose an approach to verify STACs at the source code level. In particular, we focus on verifying the correctness of timing enforcers that not only allocate CPU cycles to threads, but also enforce timing budgets. Our target is the ZSRM [5] mixed-criticality scheduler. ZSRM schedules threads (a.k.a. tasks) that execute a procedure periodically (such a periodic execution is called a *job*). For instance, a video processing tasks executes a frame processing procedure (job) every 50ms for a video running at 20 frames per second. Our approach works as follows:

– We model the enforcer as a set of three functions: (i) `init` is invoked once at the very beginning to initialize the enforcer to a proper state; (ii) `job_arrive` is invoked when a job arrives, i.e., when the time reached the next period of the corresponding job; and (iii) `job_depart` is invoked when a job departs, i.e., the corresponding job function completes execution. These functions execute atomically, i.e., all interrupts are disabled during their execution.

– An execution of the enforcer consists of an initial execution of `init` followed by arbitrary many executions of `job_arrive` and `job_depart`. The correctness of the enforcer is expressed as an invariant *Inv* that it must maintain. To this end, we need to prove that: (a) `init` ensures *Inv*; and (b) `job_arrive` and `job_depart` preserve *Inv*, i.e., assuming they are invoked in a state where *Inv* holds, they terminate in a state where *Inv* also holds. We can express this using the following three Hoare triples: $\{\top\}$`init`$\{Inv\}$, $\{Inv\}$`job_arrive`$\{Inv\}$, and $\{Inv\}$`job_depart`$\{Inv\}$.

– Given an implementation of the enforcer in C, we use the ACSL contract language to express these Hoare triples as function contracts. We then use the FRAMA-C [6] tool to discharge the contracts. FRAMA-C essentially constructs a verification condition from the C code and the ACSL [2] annotations and then proves it using a backend SMT solver.

Our approach overcomes the challenges mentioned earlier as follows. First, ACSL and FRAMA-C enable us to be sound w.r.t. numeric overflows, since the semantics of ACSL handles these correctly. Second, ACSL and FRAMA-C also enable us to verify enforcer implementations in the presence of pointers and memory operations. In particular, the expressiveness of ACSL allows us to specify loop invariants and supporting assertions involving pointers and structures. Finally, we handle time by encoding the semantics of timers and clocks as part of our invariant. In this rest of this paper, we present our approach and some preliminary results. We conclude with a survey of related work, and thoughts on ongoing and future work.

2. ZSRM SCHEDULING

As introduced before, ZSRM schedules threads (aka tasks) that execute a specific procedure periodically. Each such

HILT '16 Pittsburgh, PA, USA
© 2016 ACM. ISBN 123-4567-24-567/08/06. . . $15.00
DOI: 10.475/123_4

execution is called a job. Formally, a task τ_i is a tuple (T_i, C_i, W_i^0, W_i^1), where T_i is its period, C_i is its criticality, W_i^0 is the nominal worst-case execution time of a job, and W_i^1 is the overload worst-case execution of a job. Tasks are scheduled under preemptive fixed-priority scheduling policy where the job of a task is allowed to run until it completes or a job of a higher-priority task arrives and preempts it. This ensures that the job executing is the one with the highest priority ready to execute. Priorities are assigned to tasks at creation time and never changed. These priorities are assigned in a rate-monotonic [9] fashion, i.e., tasks with shorter periods are assigned higher priorities.

ZSRM *schedulability.* Consider any job J_i of task τ_i (T_i, C_i, W_i^0, W_i^1) ZSRM provides the following guarantee: J_i is guaranteed to receive W_i^1 units of CPU time by its deadline if no job J_j of any other task τ_j with a criticality higher than τ_i ($C_j > C_i$) executes for more than W_j^0.

ZSRM works in two stages. First, an offline analysis is performed to either declare the system to be unschedulable, or compute for each task τ_i a zero-slack instance Z_i. Informally, this is the last time at which jobs J_k of tasks τ_k with lower criticality than τ_i ($C_k < C_i$) must be stopped to guarantee that τ_i can execute for W_i^1 before its deadline. Next, at runtime the ZSRM scheduler uses timers to enforce the zero-slack instants (stopped lower-criticality tasks) as well as enforce that no job of any task τ_i exceeds its W_i^1 execution time.

In this paper we focus on proving the correctness of the budget enforcement that guarantees that no task τ_i executes beyond its W_i^1.

3. ZSRM SCHEDULER

The ZSRM scheduler is implemented in C as a Linux kernel module. The main global data structure used is an array of `reserve` structures (cf. Fig. 1), where each element represents a CPU budget (or CPU reservation) associated with a specific task. The scheduler maintains two linked lists of `reserve` structures: (i) the list of all tasks ordered by priority: this is maintained via the `rm_next` field; and (ii) the list of all reserves (and hence tasks) with active jobs (i.e., ready to execute) sorted by priority: this is maintained via the `next` field. Pointers to the heads of the two lists are stored in global variables `rm_head` and `readyq` respectively. Note that while the lists are updated dynamically by modifying their heads and the `next` and `rm_next` fields, the memory for the list elements are allocated statically via the array.

The other fields of `struct reserve` are: (i) `pid` stores the task's process id; (ii) `rid` stores the element's index, i.e., for any `i`, we have `reserve_table[i].rid == i`; (iii) since ZSRM schedules preemptively, a job's execution can be split up into segments by jobs with higher priority; `start_ns` and `stop_ns` record the starting and stopping times (in nanoseconds) of the most recently completed or ongoing job segment; these are used to compute a running count of the actual time allocated to the job, which is recorded in `current_exectime_ns`; (iv) `exectime_ns` is the maximum time that can be allocated to the job; (v) `period` and `priority` are self-explanatory; (vi) `period_timer` is used to enforce periodic job arrivals; and (vii) `enforcement_timer` is used to limit the job's execution time.

Even though the `reserve_table` array has a fixed size, not all its elements are used at all times. Unused elements are identified by distinguished field values, usually 0 or `NULL`. Initially, all elements are unused. As new tasks are added dynamically at runtime, via an entry function `create_reserve`, they are assigned to unused elements.

```
struct reserve {
  pid_t  pid;
  int rid;
  unsigned long long start_ns;
  unsigned long long stop_ns;
  unsigned long long current_exectime_ns;
  unsigned long long exectime_ns;
  struct timespec period;
  unsigned long long period_ns;
  struct timespec execution_time;
  int priority;
  struct zs_timer period_timer;
  struct zs_timer enforcement_timer;
  struct reserve *next;
  struct reserve *rm_next;
} reserve_table[MAX_RESERVES];
```

Figure 1: The global `reserve` structure array. Each element represents a task.

The scheduler has three other entry functions besides `create_reserve`: (i) `init` is called once when the kernel module is loaded; (ii) `job_arrive` is called every time a new job is activated periodically with the index of the corresponding task in `reserve_table` as argument; and (iii) `job_depart` is called when the currently executing job completes, again with the index of the corresponding task as argument. All entry functions execute without preemption, and the call to `job_arrive` preempts the currently executing job if its priority is lower than the arriving job. Once an entry function terminates, the job corresponding to `*readyq` is executed.

4. ZSRM SCHEDULER INVARIANTS

Semantically, the ZSRM scheduler can be modeled as a sequential C program that first calls `init`, and then the other three entry functions in arbitrary order. It behaves correctly if it maintains a specific set of invariants across invocations of its entry functions. In this section, we describe these invariants and show how they are expressed via predicates and contracts in ACSL [2]. The predicates presented in this paper assume that the size of the `reserve_table` array (i.e., `MAX_RESERVES`) is 10. This can be changed to any positive integer, and our approach would still be applicable.

Basic Predicates. The following predicates represent basic useful concepts:

– A pointer is either the address of an element of `reserve_table` or `NULL`.

```
/*@predicate elem(struct reserve *p) = \exists int i;
(0 <= i < 10 && p == &(reserve_table[i]));*/
/*@predicate elemNull(struct reserve *p) =
(p == \null) || elem(p);*/
```

– Variables `readyq` and `rm_head` are either the address of an element of `reserve_table` or `NULL`.

```
/*@predicate fp11 = elemNull(readyq);*/
/*@predicate fp12 = elemNull(rm_head);*/
```

– The `rid` fields of `reserve_table` elements are correct (note that `==>` denotes logical implication in ACSL):

```
/*@predicate fp14 = \forall int i; 0 <= i < 10 ==>
  reserve_table[i].rid == i;*/
```

Linked List Predicates. Recall that the scheduler maintains two linked lists of tasks via the `next` and `rm_next` fields of the `reserve` structure. The following two predicates state that these fields are either `NULL` or point to some element of `reserve_table`:

```
/*@predicate fp21 = \forall struct reserve *p;
elem(p) ==> elemNull(p->next);*/
/*@predicate fp22 = \forall struct reserve *p;
elem(p) ==> elemNull(p->rm_next);*/
```

The next predicate states that if there are two elements `p` and `q` of `reserve_table` such that `p->next` points to `q` then `p->priority >= q->priority`.

```
/*@predicate fp31 = \forall struct reserve *p, *q;
elem(p) ==> elem(q) ==> (p->next == q) ==>
(p->priority >= q->priority);*/
```

Note that predicates `fp11` and `fp31` together imply that the linked list obtained by starting with `readyq` and following the `next` field has monotonically non-increasing `priority` values. This shows that the scheduler correctly implements the preemptive fixed priority policy since the task corresponding to `*readyq` is the one scheduled at the end of each entry function. In fact, `fp31` is a stronger condition than we need since it holds even for elements of `reserve_table` that are unreachable from `readyq`. However, it is easier to verify. Note also that since the memory for the linked list is statically allocated, we do not have to define and prove list-reachability predicates, which are substantially harder, and require reasoning about separation among heap fragments.

The next predicate states that if there are two elements `p` and `q` of `reserve_table` such that `p->rm_next` points to `q` then `p->period_ns <= q->period_ns`.

```
/*@predicate fp32 = \forall struct reserve *p, *q;
elem(p) ==> elem(q) ==> (p->rm_next == q) ==>
(p->period_ns <= q->period_ns);*/
```

Thus, the list of tasks starting with `rm_head` and following the `rm_next` field is sorted in increasing order of periods. We now turn our attention to predicates that involve time.

Time-Related Predicates. In this paper, we focus on verifying that the scheduler correctly enforces the maximum time budget W_i^1 of each job. This time budget is precomputed and stored in the `exectime_ns` field of the `reserve` structure corresponding to the job's task. In addition, a running count of the actual time used by the job is maintained in the `current_exectime_ns` field, which is updated every time the job is preempted. The enforcement of the time budget is then expressed by the following predicate:

```
/*@predicate zsrm3 = \forall int i; 0 <= i < 10 ==>
  reserve_table[i].current_exectime_ns <=
  reserve_table[i].exectime_ns;*/
```

The actual enforcement of the budget is achieved by ensuring that whenever a job is scheduled, its `enforcement_timer` is set to go off at the job's time budget expiry time, and that the handler function for this timer terminates the job by sending an appropriate signal. Recall that the job to be scheduled is always pointed to by `readyq`. Hence, this condition is expressed by the following predicate:

```
/*@predicate zsrm2 = elem(readyq) ==>
  (readyq->enforcement_timer.expiration.tv_sec
    * 1000000000L +
  readyq->enforcement_timer.expiration.tv_nsec) <=
  (readyq->exectime_ns - readyq->current_exectime_ns);*/
```

We verify that `current_exectime_ns` records the "actual" time allocated to a job correctly as follows: (i) we introduce a global "ghost" variable `now` to represent physical time, following Abadi and Lamport [1]; (ii) we introduce a function `stac_time` and ensure that it is the only function used by the scheduler to obtain the current clock value; (iii) we provide the following ACSL contract for `stac_time`:

```
/*@requires \true; @assigns now;
  @ensures now > \old(now) && \result == now;*/
```

In essence, this contract states that `stac_time` returns the "actual time" and time increases strictly across multiple invocations. Next, we add another ghost field `real_exectime_ns` to the `reserve` structure to represent the "actual" time allocated to a job. We also add ghost code to update `real_exectime_ns` of the active job, whenever it is preempted, in a trivially correct way using the `now` variable directly. The following predicate states that `current_exectime_ns` always over-approximates `real_exectime_ns`, which is sufficient to prove that the scheduler enforces W_i^1 correctly.

```
/*@predicate zsrm1 = \forall int i; 0 <= i < 10 ==>
  reserve_table[i].real_exectime_ns <=
  reserve_table[i].current_exectime_ns;*/
```

Verifying Predicate Inductiveness. The overall predicate that prove inductive across each entry function is the conjunction of all the predicates presented so far. We refer to this predicate as `zsrm`. To prove that `zsrm` is inductive across a function `foo` we add the following ACSL contract to `foo`.

```
/*@requires zsrm; @assigns assigns-list;
  @ensures zsrm; */
```

Informally, this contract states that if `foo` is executed in a state that satisfies `zsrm` then the state after its execution also satisfies `zsrm`, in effect that `zsrm` is inductive across `foo`'s execution. We then verify that the implementation of `foo` satisfies this contract using FRAMA-C. Note that `assigns-list` in the contract denotes a set of locations potentially modified by `foo` and is specific to the body of `foo` itself.

5. VERIFICATION

We verified the inductiveness of the predicate `zsrm` presented in Sec. 4 across all four entry functions of the ZSRM scheduler. We used FRAMA-C, a deductive verifier for ANSI C programs annotated with function contracts and loop invariants using ACSL. FRAMA-C works by generating Floyd-Hoare [7] style verification conditions (VCs) from the annotated C program, and proving that the VCs are valid using backend theorem provers, such as Coq, Z3, or CVC. FRAMA-C uses WHY3 programs as an intermediate format to represent and prove the VCs. We used FRAMA-C Aluminium, and WHY3 version 0.87.1, the latest available at the time of our experiments. FRAMA-C can use multiple SMT solver in parallel to discharge each VC. We used Z3 v4.4.2, CVC3 v2.4.1 and CVC4 v1.4, with a timeout of 200 seconds. Since FRAMA-C runs multiple solvers in parallel, it benefits from multi-core processors. All our experiments were done on a Intel Xeon E5-2687W v3 machine with 20 cores (40 hyperthreads) running at 3.1 GHz, and 120 GB RAM.

The implementation of ZSRM we verified consisted of about 20 functions (4 entry functions and 16 supporting ones) spread over 1100 lines of pure C (i.e., no assembly) and 340 lines of ACSL annotations. In the end, FRAMA-C could verify all contracts in about 17 minutes. We now present some salient aspects of this verification effort.

– We annotated all functions in the scheduler with contracts derived from the `zsrm` predicate. Each function's contract was verified separately, but the verification was compositional. Thus, if function `foo` calls `bar`, then when verifying the contract of `foo` FRAMA-C uses the contract of `bar` at its callsite. This makes our verification more tractable.

– To successfully prove the inductiveness of ZSRM, we needed to strengthen it with the following extra predicate:

```
/*@predicate zsrm4 = \forall int i; 0 <= i < 10 ==>
(reserve_table[i].enforcement_timer.expiration.tv_sec
  * 1000000000L +
reserve_table[i].enforcement_timer.expiration.tv_nsec)
<= reserve_table[i].exectime_ns;*/
```

This is necessary to rule out numeric overflows and underflows when timestamp counters are updated. Note that sound C semantics used by FRAMA-C enables us to explicitly detect and fix such issues.

– We detected three bugs in the scheduler implementation. One was an "off-by-one" error where we had a $>$ comparison instead of a $\geq$. Another was a potential `NULL` pointer dereference. The last was a missing `return` statement. Unlike model checking, we do not get counterexamples during deductive verification. Instead, a bug results in a failed proof for a VC, and is discovered by inspecting the VC manually.

– In addition to function contracts, we added 10 loop invariants to enable FRAMA-C to complete verification successfully. In most cases, the loop invariants were simple, and required modest effort to construct. One particularly complicated case involved multiple nested loops.

– In some cases, FRAMA-C required "supporting assertions" to complete successful verification. In essence, this allows the use of Hoare's chaining rule. For example, consider a program $c_1; c_2$ where ; denotes sequential composition. Suppose we have to prove $\{P\}c_1; c_2\{Q\}$. The VC generated for this may be too difficult for an SMT solver. However, suppose we add an assertion α after c_1. Then proving $\{P\}c_1; \alpha; c_2\{Q\}$ is reduced by FRAMA-C to proving $\{P\}c_1\{\alpha\}$ and $\{\alpha\}c_2\{Q\}$ separately. The VCs for these two sub-proofs are simpler and in practice easier to discharge.

We believe that both the annotation level and the verification effort are comparable to other recent deductive source code verification efforts. Moreover, the effort is justified since it targets a critical, continuously used, and rarely-updated component of safety-critical systems and the effort is amortized over the long deployment of such systems.

Related Work. There has been a recent resurgence of deductive software verification [10] driven by well-designed contract specification languages such as ACSL and Ada 2012 [3], SMT solvers such as Z3 and CVC, and auto-active verifiers such as FRAMA-C and Boogie. This technique has been used to verify complex software ranging from container libraries, numeric programs, to full-fledged operating systems [4, 8]. Our work aims to build on this work to verify timing enforcers. Timing enforcers, such as ZSRM [5], are typically verified at the protocol/algorithmic level. Our goal is to push verification closer to its actual implementation. Finally, the use of deductive verification for real-time systems has been performed [1] but the focus has been on models. In contrast, we focus on source code.

Ongoing and Future Work. We are currently finishing up the verification of ZSRM. Currently, we assume that `stac_time` returns the precise value of physical time. Of course, this is impossible since all computation takes time, and the value returned by `stac_time` must be less that physical time at the point where control returns to its caller. We must model this appropriately using contracts. This in turn requires defining a formal semantics of STACs, which is one of our goals. Finally, we aim to validate our approach on other timing enforcers besides ZSRM.

Acknowledgment. We thank Mark Klein for many insightful comments and suggestions. Copyright 2016 ACM[1].

6. REFERENCES

[1] M. Abadi and L. Lamport. An Old-Fashioned Recipe for Real-Time. *TOPLAS*, 16(5), 1994.

[2] ACSL website. http://frama-c.com/acsl.html.

[3] AdaCore. Ada 2012: Contracts and Aspects, 2012. http://www.adacore.com/uploads/technical-papers/Ada2012_Rationale_Chp1_contracts_and_aspects.pdf.

[4] E. Cohen, M. Dahlweid, M. A. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *Proc. of TPHOLs*, 2009.

[5] D. de Niz, K. Lakshmanan, and R. Rajkumar. On the Scheduling of Mixed-Criticality Real-Time Task Sets. In *Proc. of RTSS*, 2009.

[6] Frama-C website. http://frama-c.com.

[7] C. Hoare. An axiomatic basis for computer programming. *CACM*, 12(10), 1969.

[8] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood. seL4: formal verification of an OS kernel. In *Proc. of SOSP*, 2009.

[9] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *JACM*, 20(1), 1973.

[10] J. Tschannen, C. A. Furia, M. Nordio, and N. Polikarpova. AutoProof: Auto-Active Functional Verification of Object-Oriented Programs. In *Proc. of TACAS*, 2015.

[1]This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. [Distribution Statement A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM-0003828