Contract-Based Verification of Timing Enforcers

Sagar Chaki, Dionisio de Niz

October 6, 2016
Motivation

STAC = software that accesses the system clock, exchanges clock values, and uses these values to set timers and perform computation

- Key to real-time and cyber-physical systems
- Essential to keep software in sync with the physical world
- Examples = thread schedulers and time budget enforcers, distributed protocols (e.g., plug-and-play medical devices)

Goal: Formally verify STACs at the source code level using deductive (aka auto-active) verification

- Target: ZSRM mixed-criticality scheduler
  - Performs thread CPU allocation and time budget enforcement
  - Available as Linux kernel module implemented in C
  - Currently we focus on ZSRM budget enforcement only
Why Verify Source Code?

Push assurance closer to executable level
• Use verified compilers (e.g., CompCERT) to close the final gap

Don’t need to sacrifice performance
• This is a problem when we verify models
• And is a no-go for low-level system software

Easier to integrate with existing systems
• Linux kernel module means anyone using Linux can use it
• Can be modified to work with other OSes, such as SEL4
• What You Verify Is What You Execute!
Why use Auto-Active Verification?

Soundness

Language expressivity
- Pointers, recursion, loops

Rich specification
- Quantifiers
- Predicates
- Separation

Tool maturity
- Frama-C
  - Multiple backend SMT solvers

Good Balance between human intuition and brute force search
FRAMA-C

Deductive verifier for ANSI C programs

- [https://frama-c.com/](https://frama-c.com/)
- Used to verify software in a number of projects (NASA, avionics, container libraries, hypervisors)

User provides function contracts and loop invariants

- Pre-and-post conditions, which variables are modified, etc.
- Expressed via ACSL language

Frama-C incorporates a number of plugins

- We use the WP (weakest-precondition) plugin
Quick Intro to Frama-C
ZSRM Timing Enforcer Terminology

Threads/tasks
- \( T = \{ \tau_1, \tau_2, \ldots \} \)
- Executes with preemption (i.e., broken up into chunks)
- Each task is a periodic sequence of "jobs"

System calls
- Job arrives: \( job\_arrive(\tau) \)
- Job departs: \( job\_depart(\tau) \)

Timer handlers

Enforcer Functions \( EF = \text{System calls} \cup \text{Timer handlers} \)
- Execute atomically (i.e., without preemption)
Execution/Timeline

Time = Global “Newtonian” clock
• Flows monotonically, dense real-time

Execution \( \pi = S_1 \rightarrow S_2 \rightarrow S_3 \ldots S_{n-1} \rightarrow S_n \)

State \( s_i = (c_i, \text{variable values}) \)
General Verification Strategy

Express property to be verified as an ACSL predicate:

```c
/*@predicate prop = ...;*/
```

For each enforcer function, define an ACSL contract that assumes and ensures “prop”

```c
/*@requires prop;  
@assigns ...;  
@ensures prop;*/
```

For initialization function, we only add “@ensures prop”

Verify the contracts using Frama-C
Implementation Details

In ZSRM, each task is a reservation. Inspired by resource kernels, such as Linux-RK.

The “reserve” struct has pointer fields that enable construction and manipulation of linked lists.

Memory for reservations allocated statically. Simplifies verification considerably since “separation” between the array elements is easy to specify and sufficient for verification.

```c
struct reserve {
    pid_t pid;
    int rid;
    unsigned long long start_ns;
    unsigned long long stop_ns;
    unsigned long long current_execetime_ns;
    unsigned long long execute_time_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];
```
Basic Predicates

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

```c
/*@predicate elem(struct reserve *p) = \exists i \in \{0, \ldots, 10\} \land p == &\text{reserve_table}[i];*/

/*@predicate elemNull(struct reserve *p) = (p == \null) \lor elem(p);*/
```

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

```c
/*@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):

```c
/*@predicate fp14 = \forall i \in \{0, \ldots, 10\} \rightarrow reserve_table[i].rid == i;*/
```
Linked List Predicates

Two pointer fields used to maintain separate linked lists:
next = list of “ready” tasks in order of decreasing priority
rm_next = list of all tasks maintained in order of increasing period

Pointers are either NULL or point to appropriate array elements.

List of ready tasks maintained in order of decreasing priority

List of ready tasks maintained in order of increasing period

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

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

/*@predicate fp32 = \forallall struct reserve *p, *q;
elem(p) => elem(q) => (p->rm_next == q) =>
(p->period_ns <= q->period_ns);*/
Time-Related Predicates

/*@predicate zsr3m2 = \forall int i; 0 <= i < 10 ==> reserve_table[i].current_execetime_ns <=
  reserve_table[i].execetime_ns;*/

/*@predicate zsr3m2 = elem(readyq) ==> (readyq->enforcement_timer.expiration.tv_sec
  * 1000000000L +
  readyq->enforcement_timer.expiration.tv_nsec) <=
  (readyq->execetime_ns - readyq->current_execetime_ns);*/
Using Ghost Variable to track execution time

We add a ghost field `real_execetime_ns` to the `reserve` struct

- This is supposed to compute the actual CPU usage of task
- We update this variable in `job_arrrive()` and `job_depart()`

We prove that the time accounting by ZSRM source code is conservative, i.e., `current_execetime_ns ≥ real_execetime_ns`

```c
/*@predicate zsrn1 = \forall int i; 0 ≤ i < 10 =>
reserve_table[i].real_execetime_ns ≤
reserve_table[i].current_execetime_ns;*/
```

Overall predicate is the conjunction of all the predicates seen so far
Final Verification

Frama-C Aluminium
- 3 backend SMT solvers: Z3, CVC3, CVC4
- 200s timeout

Dell Blade Server
- 40 cores (20 × 2 hyper-threads)
  - Intel(R) Xeon(R) CPU E5-2687W v3 @ 3.10GHz
- 128GB RAM
- Ubuntu 14.04

ZSRM implemented in C as user-space program
- 1484 LOC (14549 LOC after preprocessing)
- 33 functions (enforcer functions and others called by them)

Total verification time: approx 18 minutes (real) 25 minutes (user)
Conclusion and Future Work

Using deductive verification to prove timing and logical correctness of the ZSRM timing enforcer

• Prove that threads are restricted to CPU budgets
• Formalized property as ACSL contracts
• Used Frama-C to discharge them

Preliminary work but promising results

• Working on formalizing soundness of approach
• Completing implementation of kernel module
• Verification of additional timing property (e.g., zero-slack instant)
QUESTIONS?