Auto-Active Verification of Software with Timers and Clocks (STAC)

Sagar Chaki

Dionisio de Niz

Mark Klein

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.

DM-0004125

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

To our knowledge, the first formally verified and performant timing enforcer

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

- Performance 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 OSs (ZSRM in VxWorks), such as SEL4
- What You Verify Is What You Execute!

Why Use Auto-Active Verification?

Soundness

Language expressivity

- Pointers, recursion, loops

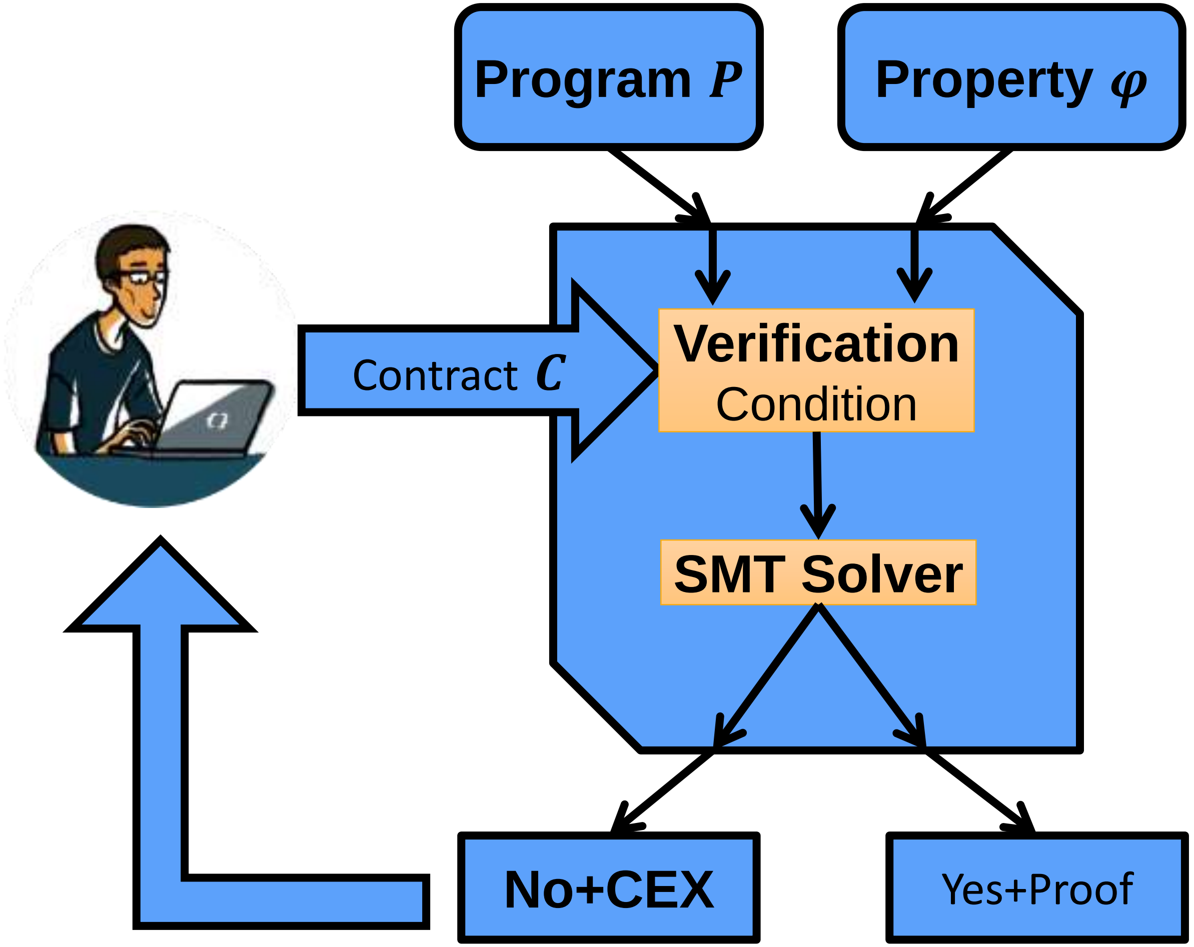
Rich specification

- Quantifiers
- Predicates
- Separation

Tool maturity

- Frama-C
  - https://frama-c.com/
  - Contracts expressed in ACSL

Good Balance between human intuition and brute force search



Auto-Active Verification

Terminology

Threads/tasks

- $T = \{\tau_1, \tau_2, \ldots\}$
- Executes with preemption (i.e., broken up into chunks)
  - Chunks not known at design time
- Initially each task is one continuous computation (e.g., a function)
  - later we will add periods

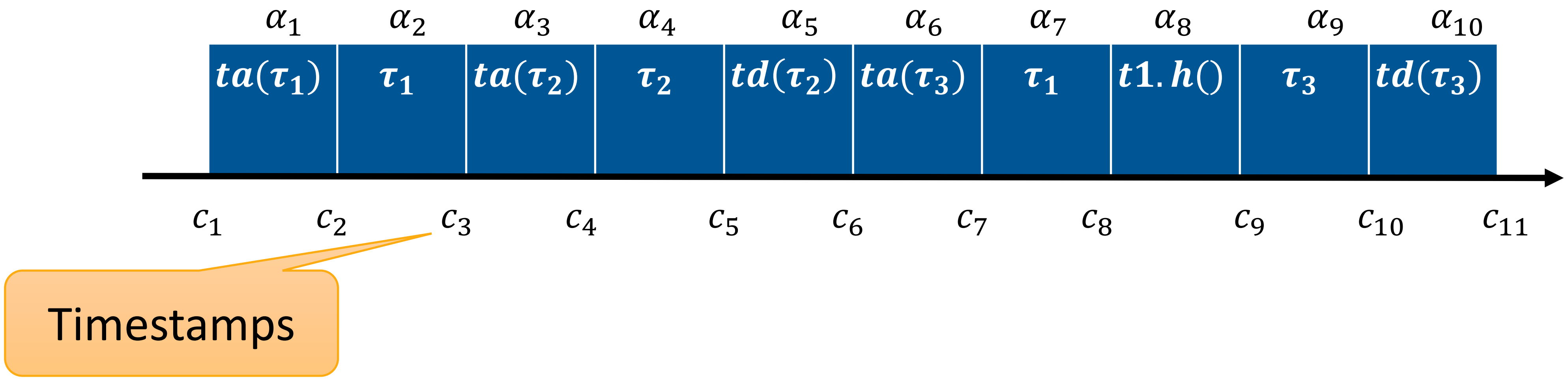
Enforcer Functions EF = System calls ∪ Timer handlers

- Execute atomically (i.e., without preemption)
- System calls
  - Task arrives : $ta(\tau)$
  - Task departs : $td(\tau)$

Execution/Timeline

Time = Global "Newtonian" clock

- Flows monotonically, dense real-time



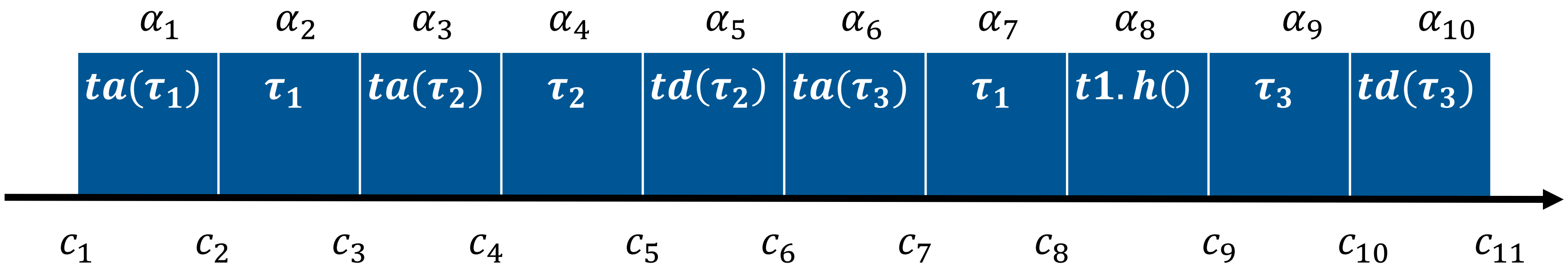
$$\text{Execution } \pi = s_1 \xrightarrow{\alpha_1} s_2 \xrightarrow{\alpha_2} s_3 \ldots s_{n-1} \xrightarrow{\alpha_{n-1}} s_n$$

$$\text{State } s_i = (c_i, r_i, a_i)$$

Thread CPU Usage

$C(\pi, \tau)$ = total cpu usage by thread τ over execution π

- Add up durations of all the transitions labeled by τ



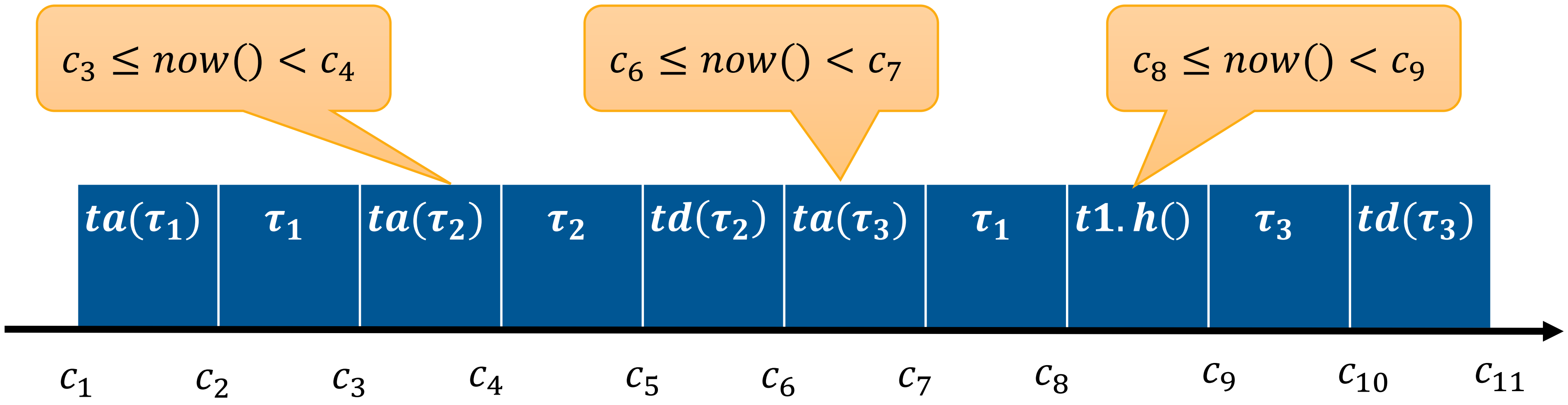
$$C(\pi, \tau_1) = (c_3 - c_2) + (c_8 - c_7)$$
$$C(\pi, \tau_2) = (c_5 - c_4)$$
$$C(\pi, \tau_3) = (c_{10} - c_9)$$

$C(\pi, \tau)$ can never be measured precisely
But can be over-approximated!

Measuring Current Time

System calls and timer handlers use a special function $now()$ to measure current time

We assume that $now()$ returns a value that is within the time boundary of the transition in which it is executed



We assume that multiple calls to $now()$ return strictly increasing values

- Implemented using hardware timestamp counter

Theorem: Over-Approximating CPU Usage

Theorem 1. *For any execution* $\pi = s_1 \xrightarrow{\alpha_1} s_2 \ldots s_{n-1} \xrightarrow{\alpha_{n-1}} s_n$ *and thread* τ, *the following four conditions hold:*

$$\begin{array}{ll} (\mathrm{C1}) & n > 1 \wedge \alpha_{n-1} = \tau \implies \tau.start(\pi) \leq c_{n-1} \\ (\mathrm{C2}) & n = 1 \vee \alpha_{n-1} \neq \tau \implies \tau.start(\pi) \leq c_n \\ (\mathrm{C3}) & n > 1 \wedge \alpha_{n-1} = \tau \implies C(\pi, \tau) \leq \tau.usage(\pi) + c_n - c_{n-1} \\ (\mathrm{C4}) & n = 1 \vee \alpha_{n-1} \neq \tau \implies C(\pi, \tau) \leq \tau.usage(\pi) \end{array}$$

Key Result

Using CPU Estimate to Enforce Budget

Each thread τ has a time budget $B(\tau)$

Definition 3 (Timer). *We say that a budget timer is always properly activated for a thread* τ, *denoted* $Timer(\tau)$, *if at the end of each execution* $\pi = s_1 \xrightarrow{\alpha_1} s_2 \ldots s_{n-1} \xrightarrow{\alpha_{n-1}} s_n$ *such that* $\alpha_{n-1} \in F \wedge sched(s_n, \tau)$ *there exists an active timer* $t \in a_n$ *such that* $t.c \leq c_n + B(\tau) - \tau.usage$.

Theorem 2. *For any thread* τ, *if* $Timer(\tau)$, *then* τ *never exceeds its budget, i.e., at the end of each execution* π, *we have* $C(\pi, \tau) \leq B(\tau)$.

Results extended to periodic threads as well

Verifying $Timer(\tau)$ on Source Code

Started with ZSRM implementation as Linux kernel module

Expressed $Timer(\tau)$ as ACSL annotations and verified with Frama-C

Complete source code with ACSL annotations publicly available

- http://www.andrew.cmu.edu/~schaki/misc/iccps17.tgz
- Compiles on recent Linux distributions
  - Tested to demonstrate good performance
- Verifies with Frama-C Aluminium
- Paper under submission

QUESTIONS?

Please attend the poster session