Safety Verification for Real-time Event-driven Programs

Pierre Ganty and Rupak Majumdar

CS Department, University of California, Los Angeles, CA, USA
{ `pganty,rupak` }`@cs.ucla.edu`

Abstract. Embeddded real-time systems are typically programmed in low level languages which provide support for event-driven task processing and real-time interrupts. We show that the model checking problem for real-time event-driven Boolean programs for safety properties is undecidable. In contrast, the model checking problem is decidable for languages such as Giotto which statically limit the creation of tasks. This gives a technical reason (static analyzability) to prefer higher-level programming models for real-time programming, in addition to the usual readability and maintainability arguments.

1 Introduction

Real-time event-driven software is the basis of many safety-critical systems, ranging from automobile and avionics control units to medical devices and to large scale supervisory control and data acquisition (SCADA) systems. These systems are often programmed in low level imperative programming languages which offer the programmer an interface for posting and executing tasks based on external or internal events and access to a real-time clock. The basic programming model is as follows. The program is written as a set P of procedures called *handlers* that share a finite global state. In addition to core imperative language constructs, each handler can make *asynchronous calls* $\mathsf{future}\ (p,t)$, where $p \in P$ is a handler, and $t \geq 0$ is an integer time step. Intuitively, $\mathsf{future}\ (p,t)$ schedules the logical task implemented by p to be executed t time steps from now.

Asynchronous calls are stored in a (timed) task buffer for later execution. Each element in the task buffer is a pair (p,t), where p is a handler, and t is the number of time steps in the future when p should be executed. If $t = 0$ for a pair (p,t), we say p is enabled.

Execution of the program is controlled by the ticks of a logical clock. Initially, the task buffer contains a special enabled "main" handler. In each time step, a scheduler picks and removes an enabled handler from the task buffer and executes the code of the handler to completion, in logical zero time. The execution of a handler can cause new handlers to be posted to the task buffer (through the execution of future statements). While there are further enabled handlers, the scheduler non-deterministically picks some enabled handler and runs it to completion (this can lead to further posted tasks). If there are no enabled handlers in the task buffer, time advances by one tick. This causes every (p,t) pair in the task buffer to be replaced by $(p, t-1)$, and the scheduler runs again for this time step.

In the most primitive setting, the programming language is C or assembly, with a timer interrupt and a hand-coded scheduler and event manager. More recently, low level virtual machines such as the E-machine [**?**] have been proposed as a clean logical model for real-time programming. Similar models for logical execution of real-time code are used to implement synchronous languages.

The future construct is a powerful mechanism to express event-driven and time-triggered actions in an embedded system, and this style of programming has been used to implement sophisticated real-time control systems such as autonomous helicopter flight control [**?**]. However, writing correct real-time event-driven programs is hard, as the control flow of the program is obscured by the loose coupling between the handlers provided by future. Therefore, it would be useful to provide algorithmic tools to check for correctness properties of these programs. For non-real time event-driven programs, in which every asynchronous call is of the form future $(p, 0)$ for some $p \in P$, checking safety and liveness properties is indeed decidable [**?**,**?**,**?**], essentially by reduction to Petri nets. In fact, the safety verification problem is decidable for more general models, such as event-driven programs with priorities [**?**]. The decidability results are non-trivial, as the programs are not finite-state: the task buffer can grow unboundedly large in the course of the execution.

We show in this paper that checking safety properties for real-time event-driven programs, on the other hand, is *undecidable*. We work in the simplified setting where each future statement is either future $(p, 0)$, signifying the handler p should be executed in this time step, or future $(p, 1)$, signifying the handler p should be executed one time step from now. Then, the execution state of the program contains two task buffers: buffer b_0 containing tasks that are enabled "now" and buffer b_1 containing tasks to be executed in the next time step. When buffer b_0 is empty, execution moves to tasks in b_1 (and puts future tasks in the "next" buffer b_2. Conceptually, this can be modeled by assigning priorities to posted tasks, with the tasks in b_i having priority over b_{i+1}. However, the decidability results from [**?**] for event-driven programs with priorities do not apply: there are an infinite number of priorities. Using the observation that only two buffers are "active" at any time, and the techniques of [**?**], we can reduce checking safety properties of real-time event-driven programs to checking coverability for Petri nets with two inhibitor arcs. While in general the latter problem is undecidable, our reduction produces inhibitor arcs with a specific structure (to encode the shift from one task buffer to the next and back again), for which coverability might well be decidable.

Instead, we show undecidability of the problem *ab initio*. Our proof is a careful encoding of the execution of a 2-counter machine as a real-time event-driven program. Intuitively, there is a handler h_i for each counter c_i $(i = 0, 1)$, and the value of counter c_i is maintained by the number of posted calls to handler h_i. Increment and decrement of counters can be simulated by posting or executing the corresponding handler. The problem is in simulating zero tests.

This is not possible in the non-real time case, and not possible for Petri nets. The technical part of our proof is to use the ability to "postpone" tasks to the next time step to simulate zero tests. In order to simulate a zero test for c_i, we nondeterministically assume the zero test succeeds, but set a variable remembering that zero test has been performed. We then copy the state of the machine (its control location as well as the value of the other counter) to the next time step. If in this process, an outstanding instance of c_i is found, then the non-deterministic guess is incorrect, and the current branch of the simulation "dies" by setting an error bit. Additional bookkeeping is performed to separate machine simulation steps from checking steps. Overall, the effect is that each run of the 2-counter machine can be simulated by a run of the real-time event-driven program (where in each time step, the program simulates machine instructions up to the next conditional), and conversely, any run of the real-time event-driven program which does not set the error bit corresponds to a run of the 2-counter machine.

While we focus on the undecidability of control location reachability, our proof also shows that related analysis problems, such as whether the task buffer is bounded, or if time always eventually advances, are all undecidable. Moreover, while we focus on real-time programs, even in non-real time settings, APIs implementing event-driven programming, such as libevent [**?**], additionally have a "timeout" call, where certain handlers run when the timer expires. These calls are ignored or abstracted in decidability proofs for event-driven systems [**?**,**?**]. Our results show that safety verification is undecidable if these calls are modeled.

While our result is negative, there is a different interpretation for it. The E-machine was proposed by its authors as a target language for a real-time compiler, and direct programming at the E-machine level was discouraged. Instead, they proposed the use of higher-level languages such as Giotto [**?**] or xGiotto [**?**] to write code at the programmer level. (More recently, languages like Virgil [**?**] has been proposed with similar intent.) By restricting the ability to post tasks arbitrarily, these higher-level languages ensure that for any Giotto or xGiotto program, at any point of the execution, there is at most a bounded, statically determined, number of posted tasks. In this case, just by finiteness of the state space, all verification problems are decidable. Our result can be interpreted as an argument for using higher-level programming languages: programs written in the higher-level languages can come with tool support for precise model checking, programs written in lower-level languages do not.

Acknowledgments. We thank Tom Henzinger for suggesting this problem.

2 The Computational Models

2.1 Programming Model

We start with some preliminary definitions. Let Σ be a finite and non-empty set. A *multiset* $M\colon \Sigma \mapsto \mathbb{N}$ over Σ is a function that maps each symbol of to a natural value ($\mathbb{N}$ denotes the set of all natural numbers). Let us denote by

$\mathbb{M}[\Sigma]$ the set of all multiset over Σ. Given two multisets $M, M' \in \mathbb{M}[\Sigma]$ we define $M \oplus M' \in \mathbb{M}[\Sigma]$ to be multiset such that $\forall a \in \Sigma\colon (M \oplus M')(a) = M(a) + M'(a)$. We sometimes use the following notation for multisets $M = [\![q_1, (q_2)^c, q_3]\!]$ (where $c \in \mathbb{N}$) for the multiset $M \in \mathbb{M}[\{q_1, q_2, q_3, q_4\}]$ such that $M(q_1) = 1$, $M(q_2) = c$, $M(q_3) = 1$, and $M(q_4) = 0$. Also as for sets we use the symbol $\emptyset$ to denote an empty multiset.

We now define a formal model for real-time event-driven programs. We represent imperative programs using a generalization of control flow graphs [**?**], that include special edges corresponding to asynchronous procedure calls. Let P be a finite set of *procedure names* and X a set of Boolean variables. An *asynchronous control flow graph* (ACFG) G_p for a procedure $p \in P$ is a pair (V_p, E_p) where V_p is the set of *control nodes* of the procedure p, including a unique *start node* v_p^s and a unique *exit node* v_p^e, and E_p is a set of directed edges between the control nodes V_p. The edges in E_p are partitioned into edges $E^{(o)}$, $E^{(now)}$, and $e^{(next)}$ corresponding to one of the following:

- *an operation edge* in $E^{(o)}$ corresponding to an assignment to a variable in X, or a conditional predicate over X;
- *a current post edge* in $E^{(now)}$ to a procedure $q \in P$, or
- *a next post edge* in $E^{(next)}$ to a procedure $q \in P$.

Intuitively, a current post edge corresponds posting an asynchronous task q that should be executed in the current time step (i.e., $\mathsf{future}\ (q, 0)$), and a next post edge corresponds posting an asynchronous task q that should be executed in the next time step (i.e., $\mathsf{future}\ (q, 1)$).

A *program* $G^{\bowtie}$ comprises a set of pairwise disjoint ACFGs G_p for each procedure in $p \in P$ (we also say *handler*). The control nodes of $G^{\bowtie}$ are given by $V^{\bowtie} = \bigcup_{p \in P} V_p$: the union of the control nodes of the individual procedures. The edges of $G^{\bowtie}$ are given by $E^{\bowtie} = \bigcup_{p \in P} E_p$, the union of the edges of the individual procedures. A *timed asynchronous program*, or TAP for short, $A = (P, X, G^{\bowtie}, \mathtt{main})$ consists of a set of procedure names P, a set of variables X, a program $G^{\bowtie}$, and an initial procedure $\mathtt{main} \in P$ such that no asynchronous call edge calls $\mathtt{main}$.

Semantics. Fix a TAP $A = (P, X, G^{\bowtie}, \mathtt{main})$. A *valuation* is a mapping that associates a value to each variable in X. For each $(v, v') \in E^{(o)}$, we assume a binary *update relation* $\mathrm{Up}_{(v,v')}$ on valuations such that $(d, d') \in \mathrm{Up}_{(v,v')}$ if d' is the valuation obtained by executing the operation on edge (v, v'). This is defined in the usual way for assignments (which updates the valuation to the assigned variable) and conditionals (which ensures the conditional is true at d and $d' = d$).

We now define the *abstract semantics* of A. The abstract semantics of A is given by a transition system where each state $((v, d), M_1, M_2)$ consists in: the *abstract state* (v, d) given by a control node $v \in V^{\bowtie}$ and a valuation d of X; and two multisets $M_1, M_2 \in \mathbb{M}[P]$ called respectively the current and next multisets of pending calls.

The *initial state* is $((v^s_{\mathtt{main}}, d_0), \emptyset, \emptyset)$ in which the multisets are empty and the abstract state $(v^s_{\mathtt{main}}, d_0)$ consists in the starting node of the $\mathtt{main}$ procedure together with an initial valuation of the program's global variables. The transitions of the program are defined as follows.

Internal operation. There is a transition from a state $((v, d), M_1, M_2)$ to the state $((v', d', M_1, M_2)$ if there is an edge $(v, v') \in E^{(o)}$ and $(d, d') \in \mathrm{Up}_{(v,v')}$.

Asynchronous call. There is a transition from $((v, d), M_1, M_2)$ to $((v', d), M_1 \oplus [\![q]\!], M_2)$ if there is an edge $(v, v') \in E^{(now)}$ which calls procedure q (*asynchronous post current*). There is a transition from $((v, d), M_1, M_2)$ to $((v', d), M_1, M_2 \oplus [\![q]\!])$ if there is an edge $(v, v') \in E^{(next)}$ which calls procedure q (*asynchronous post next*). There is a transition from $((v^e_{\mathtt{main}}, d), M_1 \oplus [\![q]\!], M_2)$ to $((v^s_q, d), M_1, M_2)$ if $q \neq \mathtt{main}$ (*asynchronous call dispatch*). Also, there is a transition from $((v^e_q, d), M_1, M_2)$ to $((v^e_{\mathtt{main}}, d), M_1, M_2)$ (*asynchronous return*).

Time transition. There is a transition from $((v^e_{\mathtt{main}}, d), \emptyset, M_2)$ to $((v^e_{\mathtt{main}}, d), M_2, \emptyset)$ ♣ WHY DO YOU NEED THE FOLLOWING:♣ provided $M_2 \neq \emptyset$.

We now give some intuition about the control node $v^e_{\mathtt{main}}$ which plays a special role in the above semantics. If the current state is such that the control node is $v^e_{\mathtt{main}}$ and (i.e., $((v^e_{\mathtt{main}}, d), M_1, M_2)$ for some multisets M_1, M_2 and dataflow fact d), then a procedure call from the multiset of current pending calls (i.e. M_1), if any, is dispatched. Otherwise, if M_1 is empty, we go to the next time frame (following a time transition) provided there are pending calls to dispatch (i.e. $M_2 \neq \emptyset$). After firing the time transition the multiset of current pending calls is now given by M_2. The program terminates when both multisets are empty. Thus $v^e_{\mathtt{main}}$ models a special "dispatch loop." Our programming model and semantics is a generalization of asynchronous programs studied in [**?**,**?**].

A *run* in the transition system of a TAP A is a finite path that starts with the initial state. A state s is reachable in a TAP A if there exists a run whose last state is s.

Abstract state reachability. Given a TAP $A = (P, G^{\bowtie}, \mathtt{main})$ and an abstract state (n, d) of A, the abstract state reachability problem asks if there exists two multisets $M_1, M_2 \in \mathbb{M}[P]$ such that the state $((n, d), M_1, M_2)$ is reachable in A.

In this paper, we will show that abstract state reachability is undecidable. Our proof shows that if we can solve the above problem then we can solve the reachability problem for two counters machine, a turing powerful model. Naturally our next section recalls the definition of two counters machine and associated reachability problem.

2.2 Two Counter Machines

A 2-counter machine C (2CM for short), is a tuple $\langle \{c_1, c_2\}, L, \mathsf{Instr}\rangle$ where:

- c_1, c_2 take their values in $\mathbb{N}$;
- $L = \{l_1, \ldots, l_u\}$ is a finite non-empty set of u locations;
- Instr is a function that labels each location $l \in L$ with an instruction that has one of the following forms:
  - l: $c_j := c_j + 1$; `goto` l' where $j \in \{1, 2\}$ and $l' \in L$, this is called an increment, and we define $\mathsf{TypeInst}(l) = \mathsf{inc}_j$;

- l: $c_j := c_j - 1$; `goto` l' where $j \in \{1,2\}$ and $l' \in L$, this is called a decrement, and we define $\mathsf{TypeInst}(l) = \mathsf{dec}_j$;
- l: `if` $c_j = 0$ `then goto` l' `else goto` l'' where $j \in \{1,2\}$ and $l', l'' \in L$, this is called a zero-test, and we define $\mathsf{TypeInst}(l) = \mathsf{zerotest}_j$;

Semantics. Those instructions have their usual obvious semantics, in particular, decrement can only be done if the value of the counter is strictly greater than zero.

A *configuration* of a 2CM $\langle\{c_1, c_2\}, L, \mathsf{Instr}\rangle$ is a tuple $\langle loc, v^1, v^2\rangle$ where $loc \in L$ is the value of the program counter and, v_1 and v_2 are positive integers that gives the values of counters c_1 and c_2, respectively.

A *computation* γ of a 2CM $\langle\{c_1, c_2\}, L, \mathsf{Instr}\rangle$ is a finite non-empty sequence of configurations $\langle loc_1, v_1^1, v_1^2\rangle, \langle loc_2, v_2^1, v_2^2\rangle, \ldots, \langle loc_r, v_r^1, v_r^2\rangle$ whose length, denoted by $|\gamma|$, equals $r-1$ and such that (i) "initialization": $loc_1 = l_1$, $v_1^1 = 0$, and $v_1^2 = 0$, i.e. a computation starts in l_1 and the two counters valued to 0; (ii) "consecution": for each $i \in \mathbb{N}$ such that $1 \leq i \leq |\gamma|$ we have that $\langle loc_{i+1}, v_{i+1}^1, v_{i+1}^2\rangle$ is the configuration obtained from $\langle loc_i, v_i^1, v_i^2\rangle$ by applying instruction $\mathsf{Instr}(loc_i)$.

Control location reachability. Given a 2CM $C = \langle\{c_1, c_2\}, L, \mathsf{Instr}\rangle$ and a control location $l \in L$, the *control location reachability problem* asks if there exists a computation γ whose last configuration is $\langle l, v_1, v_2\rangle$ for some $v_1, v_2 \in \mathbb{N}$. If so we say that control location l is reachable in C.

Theorem 1. *The control location reachability for* 2CM *is undecidable.*

3 The Reduction

We are given an instance of the control location reachability problem: a 2CM $C = \langle\{c_1, c_2\}, L, \mathsf{Instr}\rangle$ and a control location $l_x \in L$. We are asked if l_x is reachable in C. We will show the abstract state reachability for timed asynchronous programs is undecidable by encoding a 2CM as a timed asynchronous program. In fact, we reduce the 2CM control location reachability to the following abstract state reachability on timed asynchronous program. Given the TAP of the Fig. 1, 2, 3, 4 is there an abstract state $(v^e_{\mathtt{main}}, d)$ where d maps `cloc` to l_x, `error` to false that is reachable? Also in the above reachable state, d maps `Oc1`, `Oc2` `c1_eq_0` and `c2_eq_0` to false, and `timer` to `on`.

The procedures. Besides `main` the program has 5 procedures: `c_1`, `c_2`, `machine`, `timeron`, `timeroff` whose details are given below.

`c_1`, `c_2`: implements some operations on counters `c_1` and `c_2`, respectively. At every point in time, the number of pending calls to each of those procedure gives the corresponding counter's value;

`machine`: simulate the counter machine;

`timeron`: opens a time frame;

`timeroff`: closes a time frame and spawns the next one by posting `timeron`().

The variables.

`Oc1`, `Oc2`: read `Oc1` as "next `c_1`()" (like in the LTL notation). This variable is

```
global error, timer, Oc1, Oc2, c_1_eq_0, c_2_eq_0, cloc, dest;

main() {
    error = false;
    timer = off;
    Oc1=Oc2=false;
    c_1_eq_0=c_2_eq_0=false;
    cloc=dest=l_1;
    post timeron();
}
```

Fig. 1. `main`(), and global variables declaration.

```
timeron () {
    if error == true || timer == on || (Oc1||Oc2) == true {
        error = true;
        return;
    }
    timer = on;
    post timeroff();
    post machine();
}

timeroff () {
    if error == true || timer == off || (Oc1||Oc2) == true {
        error = true;
        return;
    }
    nextpost timeron();
    timer=off;
    c_1_eq_0=c_2_eq_0=false;
}
```

Fig. 2. `timeron`() and `timeroff`()

```
c_i () {
    if error == true || timer == off || Ocj == true (j!=i) || c_i_eq_0 == true {
        error=true;
        return;
    }
    if Oci == true {
        Oci = false;
        if typeinst(cloc) != dec_i
            post c_i();
        cloc=dest;
        post machine();
    }
    else if c_j_eq_0 == true (j!=i)
        nextpost c_i();
    else
        post c_i();
}
```

Fig. 3. `c_1`() and `c_2`()

```
machine() {
    if error == true || timer == off || (Oc1||Oc2) == true {
        error=true;
        return;
    }
    switch(typeinst(cloc)) {
        case inc_i: // of the form c_i:=c_i+1 goto l'
            post c_i();
            cloc=l';
            post machine();
            break;
        case zerotest_i: // of the form if c_i=0 then l' else l''
            if (*) {  // non deterministic choice
                c_i_eq_0=true;
                cloc=l';
            } else {
                Oci=true;
                dest=l'';
            }
            break;
        case dec_i: // of the form c_i:=c_i-1 goto l'
            Oci=true;
            dest=l';
            break;
    }
}
```

Fig. 4. `machine`()

used to enforce that when set to true, the next dispatch to occur is `c_1`() for otherwise the program sets `error` to true;

c1_eq_0, c2_eq_0: c1_eq_0 is set to true whenever a zerotest_1 has been simulated and the if branch has been followed (that should happen whenever there are no pending call to `c_1`());

`error`: is set to true whenever the simulation is unfaithfull. This forces every subsequent reachable state be such that `error` valuates to true;

`timer`: it is switched from `off` to `on` at the beginning of a time frame (first dispatch just after a time transition, if any, or just after executing the `main`) and from `on` to `off` at the end of a time frame (last dispatch just before the time transition);

`cloc`: indicates what the current instruction is;

`dest`: it is used in some cases to indicate what the next instruction is.

Let us now get more insights on the behavior of the TAP by giving a possible execution given at Fig. 5.

The diagram gives, for the first time frame, the sequence of procedure that runs (the double arrows above the dashed and dotted line) and for each of those the calls that are posted (the dots underneath each running procedure).

First runs the `main` procedure which will initializes the global variables and post a call to `timeron`. So the multiset of pending call is $[\![\texttt{timeron}]\!]$. Now `timeron` is dispatched and posts a call to `machine` and `timeroff` (yielding $[\![\texttt{machine}, \texttt{timeroff}]\!]$). Then comes the dispatch of `machine` which will perform the actual simulation of the 2CM. First instruction is an increment of counter 1. The dispatch of `machine` posts a call to `c_1` (to simulate the actual increment) and repost itself to continue the simulation ($[\![\texttt{machine}, \texttt{timeroff}, \texttt{c\_1}]\!]$). Second instruction is an increment to `c_2` which is simulated by the dispatch of `machine` as given above ($[\![\texttt{machine}, \texttt{timeroff}, \texttt{c\_1}, \texttt{c\_2}]\!]$).

Now since we have pending call to `c_1` and `c_2` they can be dispatched. The dispatch of `c_2` does not modify the state of the TAP (and nor does the dispatch of `c_1`). Note that to do so the dispatch of `c_2` posts one call to `c_2`.

The third instruction to simulation is a decrement of counter 1. The dispatch of `machine` will set `0c1` to true ($[\![\texttt{timeroff}, \texttt{c\_1}, \texttt{c\_2}]\!]$). This enforces the next dispatch has to be `c_1` for otherwise the variable `error` is set. So the dispatch of `c_1` simulates the actual decrement. It also posts `machine` to resume the simulation ($[\![\texttt{timeroff}, \texttt{c\_2}, \texttt{machine}]\!]$).

Now follows a dispatch to `c_2` that does not modify the state of the TAP as described above.

The fourth instruction is a $\mathsf{zerotest}_1$. Since counter one equals 0 (we incremented and decremented it starting from value 0) the zero test should follow the if branch. Doing so in the TAP, the dispatch of `machine` will set the variable c1_eq_0 to true. ($[\![\texttt{timeroff}, \texttt{c\_2}]\!]$)

Hence, the dispatch of `c_2` will post a call to `c_2` in the next time frame. So we have $[\![\texttt{timeroff}]\!]$ for the current time frame and $[\![\texttt{c\_2}]\!]$ for the next time frame.

The dispatch of `timeroff` will post `timeron` in the next time frame. Now a time transition takes place. In the new time frame the bag of pending calls is given by ⟦`c_2`, `timeron`⟧.

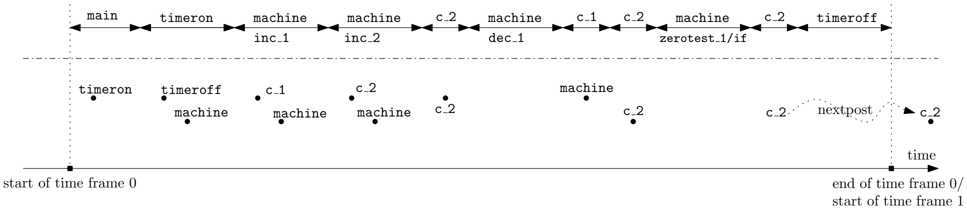

Fig. 5. An execution of the TAP

4 The Proof of Correctness

First we start with a series of facts about the program given at Fig. 1, 2, 3, 4.

1. `error` is initialized to false by `main`(), if it is switched to true its value eventually never change. Whenever `error` is set to true, the dispatch of `c_1`(), `c_2`(), `machine`(), `timeron`(), `timeoff`() does not modify the current datafact and does not add any call to the multiset of pending calls.
2. Every pending call in the current time frame will be dispatched before moving to the next time frame (i.e. before taking a timer transition). This fact holds by semantics of timed asynchronous programs.
3. `timer` is modified by `timeron`() and `timeroff`() only and is initialized by `main`().
4. no procedure but `timeroff`() can switch `c1_eq_0` or `c2_eq_0` from true to false and no procedure but `machine`() can switch `c1_eq_0` or `c2_eq_0` from false to true.
5. In a time frame there is at most one post to `timeron`() and `timeroff`().

   Proof. `main`(), which is executed only once, posts one call to `timeron`() in the same time frame. When `timeron`() is executed, it posts at most one call to `timeroff`() in same time frame, which whenever executed, posts at most one call to `timeron`() in the next time frame. Also we have that only `main`() and `timeroff`() post `timeron`(), only `timeron`() posts `timeroff`().∎

6. In frame i if the first dispatch is not `timeron`() or the last dispatch is not `timeroff`() then `error` is set to true in i.

   Proof. (1) In every time frame i, if the first dispatch is different from `timeron`() then `error` is set to true. This is so because the value of `timer` is `off` by Fact 3, `main`() and induction hypothesis (the last dispatch of frame

$i-1$ is `timeroff`()) and the first line of `c_1`(), `c_2`(), `machine`(), `timeroff`() which set `error` to true when `timer` is `off`.
(2) In every time frame i, if the last dispatch (before the time transition) is different from `timeroff`() then `error` is set to true. This is so because, after executing `timeroff`(), the value of `timer` is `off` and by the first line of `c_1`(), `c_2`(), `machine`(), `timeroff`() we find that `error` is set to true. For the case of `timeron`() we find that it cannot run after `timeroff`() because we have shown above in (1) that the first dispatch of every frame is `timeron`() for otherwise `error` is set to true.■

7. the number of pending calls to `machine`() at any point in time is bounded by one.

   Proof. `machine`() is posted once by `timeron`(), by itself, `c_1`() or `c_2`(). Fact 5 shows that `timeron`() posts at most one call to `machine`(). `c_1`() (resp. `c_2`()) posts `machine`() whenever `Oc1` (resp. `Oc2`) is true. Whenever `Oc1` and `Oc2` are set to true by the dispatch of `machine`(), it also posts no call to `machine`().■

8. if `Oc1` (resp. `Oc2`) is true, the next dispatch yields `error` is set to true unless this dispatch is `c_1`() (resp. `c_2`()).

   Proof. it follows from the condition expression of the if statement of the procedure `timeron`(), `timeroff`(), `c_2`() (resp. `c_1`()) and `machine`().■

4.1 Proof

The 2CM reaches the state (l_x, v_1, v_2) iff the associated TAP A reaches a state $((v^e_{\mathtt{main}}, d), M_1, M_2)$ where d maps `cloc` to l_x, `error` to false, and M_1, M_2 are such that

- $M_1(\mathtt{machine}) = 1$, we are "between" the simulation of two instructions of 2CM,
- $M_1(\mathtt{c\_1}) = v_1, M_1(\mathtt{c\_2}) = v_2$, we want counters to coincide with v_1, v_2,

In our proof, we will consider each instruction in turn and show how the TAP simulates it. We will also show that if the TAP does not faithfully simulate the 2CM then it will set `error` to true.

BC: $\langle l_1, 0, 0\rangle$ and — after the execution of `main`() followed by `timeron`() — $((v^e_{\mathtt{main}}, d), M_1, M_2)$ where $M_1 = [\![\mathtt{machine}, \mathtt{timeroff}]\!]$, $M_2 = \emptyset$ and d maps `error`, `timer`, `Oc1`, `Oc2`, `c1_eq_0`, `c2_eq_0`, `cloc` to false, on, false, false, false, false and l_1, respectively. Fact 6 shows that if the first dispatch to take place after executing `main`() is different from `timeron`() then `error` is set to true.

IC: let $\langle l_x, v_1, v_2\rangle$ be a state of the 2CM and $((v^e_{\mathtt{main}}, d), M_1, M_2)$ a state of the TAP where $M_1 = [\![\mathtt{machine}, \mathtt{timeroff}, (\mathtt{c\_1})^{v_1}, (\mathtt{c\_2})^{v_2}]\!]$ $M_2 = \emptyset$ and d maps `error`, `timer`, `Oc1`, `Oc2`, `c1_eq_0`, `c2_eq_0`, `cloc` to false, off, false, false, false, false and l_x, respectively.

Fact 6 says that `machine`(), `c_1`() or `c_2`(), if any, cannot be dispatched after `timeroff`() for otherwise it yields `error` set to be true. Since `error`, `timer`, `Oc1`, `Oc2`, `c1_eq_0`, `c2_eq_0` valuate to false, off, false, false, false, false, respectively, we find that the dispatch of `c_1`() or `c_2`() leaves the state unchanged. As we will see below, the update of the current state is given by the dispatch of `machine`(). So in the explainations below, `machine`() is assumed to be the dispatch to take place.

The rest of the proof naturally falls into three parts according to the instruction at l_x:

•TypeInst(l_x) = inc$_1$ and is of the form $l_x\colon c_1 := c_1 + 1;$ `goto` l'. In that case the state of the 2CM is updated to $\langle l_x, v_1 + 1, v_2\rangle$. In the TAP, the execution of `machine`() goes as follows: the conditional of the if statement fails and the block of code for the inc$_1$ case is executed. The state is updated to $((v^e_{\texttt{main}}, d), M_1, M_2)$ where $M_1 = [\![\texttt{machine}, \texttt{timeroff}, (\texttt{c\_1})^{v_1+1}, (\texttt{c\_2})^{v_2}]\!]$ (`machine`() has been posted by `c_1`() which posted itself as well); $M_2 = \emptyset$ and d maps `error`, `timer`, `Oc1`, `Oc2`, `c1_eq_0`, `c2_eq_0`, `cloc` to false, off, false, false, false, false and l' (because `cloc` is updated), respectively.

•TypeInst(l_x) = dec$_1$ and is of the form $l_x\colon c_1 := c_1 - 1;$ `goto` l'. First, we assume that $v_1 > 0$. In that case the state of the 2CM will be updated to $\langle l_x, v_1 - 1, v_2\rangle$. In the TAP, the execution of `machine`() goes as follows: the conditional of the if statement fails and the block of code for the dec$_1$ case is executed. The datafact is updated such that `Oc1` is set to true and `dest` is set to l'. A dispatch now takes place. Fact 8 shows that any dispatch but `c_1`() yields `error` to be set to true. We conclude from $v_1 > 0$, that $M_1(\texttt{c\_1}) > 0$, hence that there is a pending call to `c_1`(). So after the dispatch of `c_1`() the state is updated to $((v^e_{\texttt{main}}, d), M_1, M_2)$ where $M_1 = [\![\texttt{machine}, \texttt{timeroff}, (\texttt{c\_1})^{v_1-1}, (\texttt{c\_2})^{v_2}]\!]$ (`machine`() has been posted during the dispatch of `c_1`()); $M_2 = \emptyset$ and d maps `error`, `timer`, `Oc1`, `Oc2`, `c1_eq_0`, `c2_eq_0`, `cloc` to false, off, false, false, false, false and l' (because `cloc` has been assigned to `dest` that has been updated to l' during the dispatch of `machine`()), respectively.

Let us now assume that $v_1 = 0$. In that case the instruction is not enabled and the 2CM is "stuck" in the state $\langle l_x, v_1, v_2\rangle$. In the TAP, the execution of `machine`() will set `Oc1` to true. Fact 8 shows that any dispatch but `c_1`() yields `error` to be set to true which will happen since $v_1 = 0$, hence $M_1(\texttt{c\_1}) = 0$ (there is no pending call to `c_1`()).

•TypeInst(l_x) = zerotest$_1$ and is of the form $l_x\colon c_1 = 0$ `then goto` l' `else goto` l''. Our case study is as follows: $v_1 = 0$ and $v_1 \neq 0$.

Fact: after `c1_eq_0` or `c2_eq_0` is set to true then every post is added to the next time frame.

If $v_1 = 0$ then the 2CM updates its state to $\langle l', v_1, v_2\rangle$.

In the TAP, the execution of `machine`() goes as follows: the conditional of the if statement fails and the block of code for the zerotest$_1$ case is executed.

- if branch is taken. (this is a faithfull simulation). The dispatch of `machine`() sets `c1_eq_0` to true and sets `cloc` to l'. This, as we will see, leads a time transition to eventually take place provided the multiset of pending calls does not contain `c_1`(). Otherwise, `error` will be set to true.

We conclude from $v_1 = 0$, that $M_1(\texttt{c\_1}) = 0$, hence that there is no pending call to `c_1`(). By fact 6 (`timeroff`() occurs whenever the multiset of pending call is $[\![\texttt{timeroff}()]\!]$) we find that each pending call to `c_2`(), if any, will be dispatched before `timeroff`() and consequently be "moved" to the next time frame by the statement `nextpost`. Whenever the multiset of pending calls is $[\![\texttt{timeroff}()]\!]$ then the dispatch of `timeroff`() occurs and it resets the `c1_eq_0` to false. Then the time transition takes place since the multiset of pending calls is empty. As seen in Fact 6 the first dispatch to take place after the time transition is `timeron`() which post `machine`() so that the updated state is now $((v^e_{\texttt{main}}, d), M_1, M_2)$ where $M_1 = [\![\texttt{machine}, \texttt{timeroff}, (\texttt{c\_2})^{v_2}]\!]$ (`machine`() is reposted by `timeron`() and $M_1(\texttt{c\_1}) = v_1 = 0$ because no `c_1`() has been posted in the new time frame, $M_1(\texttt{c\_2}) = v_2$ because each call has been copied from the previous frame); $M_2 = \emptyset$ (because of the time transition) and d maps `error`, `timer`, `Oc1`, `Oc2`, `c1_eq_0`, `c2_eq_0`, `cloc` to false, off, false, false, false, false and l' (because `cloc` has been assigned to l'), respectively.

- else branch is taken. (this is an unfaithfull simulation) We conclude from $v_1 = 0$, that $M_1(\texttt{c\_1}) = 0$, hence that there is no pending call to `c_1`(). The dispatch of `machine`() sets `Oc1` to true and sets `dest` to l''. The next dispatch to occur cannot be `c_1`() (because there is none to dispatch) and so `error` is set to true by Fact 8.

If $v_1 \neq 0$ then the 2CM updates its state to $\langle l'', v_1, v_2 \rangle$.

In the TAP, the execution of `machine`() goes as follows: the conditional of the if statement fails and the block of code for the $\mathsf{zerotest}_1$ case is executed.

- the if branch is taken. (this is an unfaithfull simulation) The dispatch of `machine`() sets `c1_eq_0` to true and sets `cloc` to l'. Fact 4 shows that the only procedure that can change the value of `c1_eq_0` is `timeroff`() and Fact 6 shows it yields an error if `timeroff`() is not dispatched last in the current time frame. We conclude from $v_1 \neq 0$, that $M_1(\texttt{c\_1}) \neq 0$, hence that there is a pending call to `c_1`(). Its dispatch yields `error` to be set to true because the datafact at the time of dispatch is such that `c1_eq_0` is true.
- the else branch is taken. (this is a faithfull simulation). We conclude from $v_1 \neq 0$, that $M_1(\texttt{c\_1}) \neq 0$, hence that there is a pending call to `c_1`(). The dispatch of `machine`() sets `Oc1` to true and sets `dest` to l''. By Fact 8, the next dispatch to occur has to be `c_1`() for otherwise `error` is set to true. The dispatch of `c_1`() yields the following updated state $((v^e_{\texttt{main}}, d), M_1, M_2)$ where $M_1 = [\![\texttt{machine}, \texttt{timeroff}, (\texttt{c\_1})^{v_1}, (\texttt{c\_2})^{v_2}]\!]$ (`machine`() and `c_1`() are posted in `c_1`()); $M_2 = \emptyset$ and d maps `error`, `timer`, `Oc1`, `Oc2`, `c1_eq_0`, `c2_eq_0`, `cloc` to false, off, false, false, false, false and l'' (because `cloc` has been assigned to `dest` that has been updated to l'' during the dispatch of `machine`()), respectively.

Theorem 2. *The abstract state reachability for* TAP *is undecidable.*