Analyzing Real-Time Event-Driven Programs[⋆]

Pierre Ganty[1,2] and Rupak Majumdar[2]

[1] CS Department, University of California, Los Angeles, CA, USA
{pganty,rupak}@cs.ucla.edu
[2] IMDEA Software, Spain

Abstract. Embedded 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 automotive and avionics control units 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 access to a real-time clock and an interface for posting and executing tasks based on external or internal events. 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, there is a statement future. The future statement takes a pair (p, t) (called an *asynchronous call*) in argument, where $p \in P$ is a handler, and $t \geq 0$ is an integer time step. Intuitively, future (p, t) schedules, or *posts*, the task implemented by the handler p to be executed t time steps from now. The posted 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 remaining number of time steps in the future when p should be executed. Each such call in the buffer is called a *pending call*.

Execution of the program is controlled by the ticks of a logical clock. Each task is assumed to execute in logical zero time. Initially, the task buffer contains exactly one pending call: $(\texttt{main}, 0)$. In each time step, a *scheduler* picks and removes an arbitrary pending call $(p, 0)$ (if one exists) from the task buffer and executes the code p to completion, in logical zero time. Below we call this

[⋆] This research was sponsored by the NSF grants CCF-0546170, CCF-0702743, and CNS-0720881.

operation a *dispatch*. The execution of a handler can cause new asynchronous calls to be posted to the task buffer (through the execution of future statements in the code). If there is no pending call of the form $(p, 0)$ 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.

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 [7]. 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 [12,8,3], essentially by reduction to Petri nets. In fact, the safety verification problem is decidable for more general models, such as event-driven programs where handlers can be recursive and in which handlers have priorities [2]. The decidability results are non-trivial as the programs are not finite-state: the task buffer as well as the call stack 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 is *undecidable*. We work in the simplified setting where (1) each future statement is either future $(p, 0)$, scheduling the handler p to be executed in this time step, or future $(p, 1)$, scheduling the handler p to be executed one time step from now and (2) we do not allow recursion. This simplified setting is powerful enough to show undecidability.

Our undecidability proof for the safety checking problem uses a careful encoding of the execution of a 2-counter machine [10] as a real-time event-driven program. Intuitively, there are two handlers h_1 and h_2 for the two counters c_1 and c_2, and the value of counter c_1 (resp. c_2) is maintained by the number of pending calls $(h_1, 0)$ (resp. $(h_2, 0)$). Increment and decrement of counter c_i can be respectively simulated by posting and dispatching $(h_i, 0)$ for $i = 1, 2$. The problem is in simulating zero tests. This is not possible in the non real-time case.

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 guess the outcome of the test. If c_i is guessed to be zero, then the state of the machine (its control location as well as the value of the other counter) is copied to the next time step (i.e., dispatching the pending call $(h_i, 0)$ posts $(h_i, 1)$). However, if during this process, a pending call $(h_i, 0)$ is found, then the guess is incorrect, and the current branch of the simulation "dies" by setting an error bit.

Additional book-keeping 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 zero test guessed positive), 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 our result is negative, we consider a more positive interpretation. The E-machine was proposed in [6] as a low-level virtual machine with a clean logical model for real-time programming. It is intended 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 [5] or xGiotto [4] to write code at the programmer level. (More recently, languages like Virgil [13] 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 pending calls. 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 can not.

2 The Computational Models

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 Σ to a natural value ($\mathbb{N}$ denotes the set of all natural numbers). Let us denote by $\mathbb{M}[\Sigma]$ the set of all multisets 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.

2.1 Programming Model

We represent imperative programs using a generalization of control flow graphs [1], that includes special edges corresponding to asynchronous calls. Let P be a finite set of *procedure names* (or *handler*) and X a finite 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 as follows:

- *the operation edges* corresponding to an assignment of variables in X, or a conditional predicate over X;
- *the post edges* to a procedure $q \in P$ labelled by a statement $\mathsf{future}\ (q, 0)$ or $\mathsf{future}\ (q, 1)$.

A *program* $G^{\bowtie}$ comprises a set of pairwise disjoint ACFGs G_p for each procedure in $p \in P$. 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 *real-time asynchronous program*, or RTAP for short, $A = (P, X, G^{\bowtie}, \texttt{main})$ consists of a set of procedure names P, a set of variables X, a program $G^{\bowtie}$, and an initial procedure $\texttt{main} \in P$ that is not referenced by any post edge.

Semantics. Fix a RTAP $A = (P, X, G^{\bowtie}, \texttt{main})$. A *valuation* is a mapping that associates a Boolean value to each variable in X. For each operation edge (v, v'), 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 (see, e.g., [11]) 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 $M_1, M_2 \in \mathbb{M}[P \setminus \{\texttt{main}\}]$ called the multisets of *current* and *next pending calls* that stores pending calls of the form $(p, 0)$ and $(p, 1)$, respectively.

The *initial state* is $((v^s_{\texttt{main}}, d_0), \emptyset, \emptyset)$ in which the multisets of pending calls are empty and the abstract state $(v^s_{\texttt{main}}, d_0)$ consists in the starting node of the main procedure together with an initial valuation of the program's Boolean variables. The transitions 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 operation edge (v, v') and $(d, d') \in \mathrm{Up}_{(v,v')}$.

Asynchronous call. There is a transition from $((v, d), M_1, M_2)$ to $((v', d), M_1 \oplus [\![q]\!], M_2)$ (resp. $((v', d), M_1, M_2 \oplus [\![q]\!])$) if there is a post edge (v, v') labeled future $(q, 0)$ (resp. labeled future $(q, 1)$). There is a transition from $((v^e_{\texttt{main}}, d), M_1 \oplus [\![q]\!], M_2)$ to $((v^s_q, d), M_1, M_2)$ which we refer to as a *dispatch*. Also, there is a transition from $((v^e_q, d), M_1, M_2)$ to $((v^e_{\texttt{main}}, d), M_1, M_2)$ provided $q \in P \setminus \{\texttt{main}\}$.

Time transition. There is a transition from $((v^e_{\texttt{main}}, d), \emptyset, M_2)$ to $((v^e_{\texttt{main}}, d), M_2, \emptyset)$.

We now give some intuition about the control node $v^e_{\texttt{main}}$ which plays a special role in the above semantics. If the current state is such that the control node is $v^e_{\texttt{main}}$ (i.e., $((v^e_{\texttt{main}}, d), M_1, M_2)$ for some multisets M_1, M_2 and valuation d), then a pending call from M_1, if any, is *dispatched*. Otherwise, if M_1 is empty, we go to the next time step (following a time transition). After firing the time transition the multiset of current pending calls is now given by M_2. Thus $v^e_{\texttt{main}}$ models a special "dispatch loop". Our programming model and semantics is a generalization (with timed asynchronous calls) of the asynchronous programs studied in [12,8].

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

Abstract state reachability. Given a RTAP $A = (P, X, G^{\bowtie}, \mathtt{main})$ and an abstract state (v, d) of A, the abstract state reachability problem asks if there exists two multisets $M_1, M_2 \in \mathbb{M}[P \setminus \{\mathtt{main}\}]$ such that the state $((v, d), M_1, M_2)$ is reachable in A. If so we say the abstract state (v, d) 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 counter machines, a Turing powerful model. The next section recalls the definition of two counter machines and the 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 are counters taking 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: $c_j := c_j + 1$; $\mathtt{goto}$ 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: $c_j := c_j - 1$; $\mathtt{goto}$ 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: $\mathtt{if}$ $c_j = 0$ $\mathtt{then\ goto}$ l' $\mathtt{else\ goto}$ 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. The instructions have their usual obvious semantics, in particular, decrement can only be done if the value of the counter is positive.

A *configuration* of a 2CM $\langle\{c_1, c_2\}, L, \mathsf{Instr}\rangle$ is a tuple $\langle loc, n_1, n_2\rangle$ where $loc \in L$ is the value of the program counter and $n_1, n_2 \in \mathbb{N}$ give 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 $\langle loc_1, n_1^1, n_1^2\rangle, \langle loc_2, n_2^1, n_2^2\rangle, \ldots, \langle loc_r, n_r^1, n_r^2\rangle$ such that (i) "initialization": $loc_1 = l_1$, $n_1^1 = 0$, and $n_1^2 = 0$, i.e., a computation starts in l_1 and the two counters are set initially to 0; (ii) "consecution": for each $i \in \{1, \ldots, r-1\}$ we have that $\langle loc_{i+1}, n_{i+1}^1, n_{i+1}^2\rangle$ is the configuration obtained from $\langle loc_i, n_i^1, n_i^2\rangle$ 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 $\langle l, n_1, n_2\rangle$ for some $n_1, n_2 \in \mathbb{N}$. If so we say that control location l is reachable in C.

Theorem 1 ([10]). *The control location reachability problem for* 2CM *is undecidable.*

```
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;
    future (timeron,0);
}
```

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

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 will show the abstract state reachability for real-time asynchronous programs is undecidable by encoding a 2CM as a real-time asynchronous program.

In what follows, we use a C-like notation to represent RTAP for readability, and we also use variables that range over a finite set of values instead of only Booleans. Each piece of handler code can be converted to our formal model using standard compiler algorithms [1].

Precisely, we reduce the 2CM control location reachability to the following abstract state reachability on real-time asynchronous program. Let the RTAP given by the code of Fig. 1–6, 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`, which starts the simulation, the program has 5 procedures: `c_1`, `c_2`, `machine`, `timeron`, `timeroff` whose details are given below.
`c_1`, `c_2`: implements operations on counters `c_1` and `c_2`, respectively, as well as book-keeping to ensure the simulation is valid. The number of pending calls to each of these procedures reflects the value of the corresponding counter;
`machine`: simulates the instructions of the counter machine.

The following procedures implement book-keeping operations that ensure the simulation is valid.
`timeron`: starts a time step for simulation;
`timeroff`: terminates a time step, resetting all the "book-keeping" state, and spawns the next one by posting `timeron` for the next time step.

The variables
`Oc1`, `Oc2`: read `Oc1` as "next `c_1`" (like in the LTL notation). This variable is such that if `Oc1` is true, the next dispatch yields `error` is set to true unless this dispatch is `c_1`(); (same for `Oc2`)

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()) (same for c2_eq_0);
error: is set to true whenever the simulation is unfaithful. Once set, error remains set forever. This forces every subsequent reachable state to be such that error valuates to true;
timer: it is supposed to be switched from off to on at the beginning of a time step and from on to off at the end of a time step. A faithful simulation of the machine occurs while timer is on, and any attempt to simulate the machine while timer is off results in error being set;
cloc: points to the current location of the 2CM;
dest: is used in some cases to store a location of the 2CM such that if the 2CM is faithfully simulated cloc will be assigned to that location.

Let us now get more intuition on the behavior of the RTAP given at Fig. 1–6 by studying a possible execution that is graphically depicted at Fig. 2 and discussed below.

The diagram gives, for the first time step, the sequence of procedures that are executed in a valid simulation (the double arrows above the dashed and dotted line) and for each of those the calls it posts (the dots underneath each running procedure).

The program starts with the execution of main. It will initialize the variables and post a call to timeron. So the multiset of current pending calls is $[\![$timeron$]\!]$. Now timeron gets dispatched and posts a call to machine and timeroff (yielding $[\![$machine, timeoff$]\!]$). Then comes the dispatch of machine which performs the actual simulation of the 2CM. First instruction increments counter 1. The dispatch of machine posts a call to c_1 (to simulate the actual increment) and repost itself to continue the simulation ($[\![$machine, timeroff, c_1$]\!]$). Second instruction is an increment to c_2 which is simulated by the dispatch of machine as given above ($[\![$machine, timeroff, c_1, c_2$]\!]$).

The dispatch of c_2 does not modify the state of the RTAP. To do so the dispatch of c_2 posts one call to c_2.

The next instruction is a decrement of counter 1. The dispatch of machine will set 0c1 to true ($[\![$timeroff, c_1, c_2$]\!]$). If the next dispatch is not c_1 the variable error is set; otherwise the dispatch of c_1 simulates the actual decrement. It also posts machine to resume the simulation ($[\![$timeroff, c_2, machine$]\!]$).

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

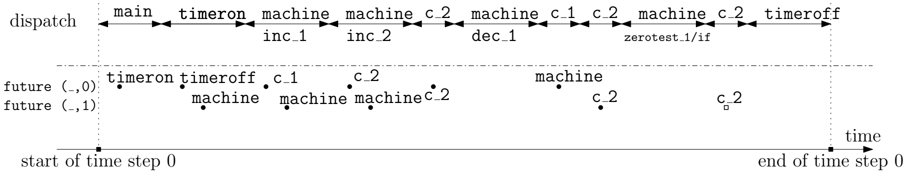

Fig. 2. An execution of the RTAP

```
timeron () {
    if error == true || timer == on || (Oc1||Oc2) == true {
        error = true;
        return;
    }
    timer = on;
    future (timeroff,0);
    future (machine,0);
}
```

Fig. 3. `timeron()`

```
timeroff () {
    if error == true || timer == off || (Oc1||Oc2) == true {
        error = true;
        return;
    }
    future (timeron,1);
    timer=off;
    c_1_eq_0=c_2_eq_0=false;
}
```

Fig. 4. `timeroff()`

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 *then* branch. Doing so in the RTAP, the dispatch of `machine` will set the variable `c1_eq_0` to true ($[\![$`timeroff`, `c_2`$]\!]$).

Hence, the dispatch of `c_2` will post a call to `c_2` in the next time step. So we have $[\![$`timeroff`$]\!]$ and $[\![$`c_2`$]\!]$ for the current and next pending calls, respectively.

The dispatch of `timeroff` will post `timeron` in the next time step. Now a time transition takes place. The pending calls in the new time step are now given by $[\![$`c_2`, `timeron`$]\!]$.

4 The Proof of Correctness

First, we start with a series of facts about the program given at Fig. 1–6.

1. `error` is initialized to false by `main`(), if it is ever set to true, its value never changes back to false. Whenever `error` is set to true, the dispatch of `c_1`(), `c_2`(), `machine`(), `timeron`(), `timeroff`() does not modify the current valuation and does not post any further call.
2. every current pending call will be dispatched before moving to the next time step (i.e., before taking a time transition). This fact holds by semantics of real-time asynchronous programs.

```
c_i () {
    if error == true || timer == off || Ocj == true || c_i_eq_0 == true {
        error=true;
        return;
    }
    if Oci == true {
        Oci = false;
        if typeinst(cloc) != dec_i
            future (c_i,0);
        cloc=dest;
        future (machine,0);
    }
    else if c_j_eq_0 == true
        future (c_i,1);
    else
        future (c_i,0);
}
```

Fig. 5. c_1() and c_2() where $\mathtt{i} \in \{1,2\}$ and $\mathtt{j} = (3 - \mathtt{i})$

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

Fig. 6. machine()

3. `timer` is modified by `timeron`() and `timeroff`() only and is initialized by `main`() to `off`.
4. only `timeroff`() can switch `c1_eq_0` or `c2_eq_0` from true to false and only `machine`() can switch `c1_eq_0` or `c2_eq_0` from false to true.
5. in a time step 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 step. When `timeron`() is executed, it posts at most one call to `timeroff`() in same time step, which whenever executed, posts at most one call to `timeron`() in the next time step. Also we have that only `main`() and `timeroff`() post `timeron`(), only `timeron`() posts `timeroff`(). □

6. in step $i > 0$ if the first dispatch is not `timeron`() or the last dispatch is not `timeroff`() then `error` is set to true in i. The exception is time step 0, in which `main`() is followed by `timeron` at the beginning.

   Proof. (1) In every time step $i > 0$, 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 step $i-1$ 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 step 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`() in the same time step because we have shown above in (1) that the first dispatch of every step is `timeron`() for otherwise `error` is set to true. □

7. the number of pending calls to `machine`(), at any point in time, is at most 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 sets `error` to true unless this dispatch is `c_1`() (resp. `c_2`()).

   Proof. it follows from the conditional of the first line of `timeron`(), `timeroff`(), `c_2`() (resp. `c_1`()) and `machine`(). □

4.1 Proof

The 2CM reaches the state (l_x, n_1, n_2) iff the associated RTAP A reaches a state $((v^e_{\texttt{main}}, d), M_1, M_2)$ where d maps `cloc` to l_x, `error` to false, and M_1, M_2 are such that:

– $M_1(\texttt{machine}) = 1$, we are “between” the simulation of two instructions of 2CM,
– $M_1(\texttt{c\_1}) = n_1, M_1(\texttt{c\_2}) = n_2$, we want counters to coincide with n_1, n_2.

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

Initialization: let $\langle l_1, 0, 0\rangle$ be the initial state of the 2CM and let $((v^e_{\texttt{main}}, d), M_1, M_2)$ be the state of the RTAP after the execution of `main`() followed by `timeron`(). Fact 6 shows that if the first dispatch immediately after executing `main`() is different from `timeron`() then `error` is set to true. So the above state is such that $M_1 = [\![\texttt{machine}, \texttt{timeroff}]\!]$, $M_2 = \emptyset$ and d maps `error`, `timer`, `0c1`, `0c2`, `c1_eq_0`, `c2_eq_0`, `cloc` to false, on, false, false, false, false and l_1, respectively.

Consecution: let $\langle l_x, n_1, n_2\rangle$ be a state of the 2CM and $((v^e_{\texttt{main}}, d), M_1, M_2)$ a state of the RTAP where $M_1 = [\![\texttt{machine}, \texttt{timeroff}, (\texttt{c\_1})^{n_1}, (\texttt{c\_2})^{n_2}]\!]$, $M_2 = \emptyset$ and d maps `error`, `timer`, `0c1`, `0c2`, `c1_eq_0`, `c2_eq_0`, `cloc` to false, on, false, false, false, false and l_x, respectively. This relationship will serve as our induction hypothesis.

Fact 6 says that if the dispatch of `machine`() or `c_1`() or `c_2`() occurs after the dispatch of `timeroff`() then `error` is set to true. Since `error`, `timer`, `0c1`, `0c2`, `c1_eq_0`, `c2_eq_0` valuate to false, on, 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 explanations 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 $\langle l', n_1 + 1, n_2\rangle$. In the RTAP, the execution of `machine`() goes as follows: the conditional of first line fails and the piece of code for the inc_1 case is executed. The state is updated to $((v^e_{\texttt{main}}, d), M_1, M_2)$ where $M_1 = [\![\texttt{machine}, \texttt{timeroff}, (\texttt{c\_1})^{n_1+1}, (\texttt{c\_2})^{n_2}]\!]$ (`machine`() posted `c_1`() and itself), $M_2 = \emptyset$ and d maps `error`, `timer`, `0c1`, `0c2`, `c1_eq_0`, `c2_eq_0`, `cloc` to false, on, false, false, false, false and l' (because `cloc` is updated), respectively. (the same holds for inc_2)

•TypeInst(l_x) = dec_1 and is of the form $l_x\colon c_1 := c_1 - 1;$ `goto` l'. First, we assume that $n_1 > 0$. In that case, the state of the 2CM will be updated to $\langle l', n_1 - 1, n_2\rangle$. In the RTAP, the execution of `machine`() goes as follows: the conditional of the first line fails and the piece of code for the dec_1 case is executed. The valuation is updated such that `0c1` 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 $n_1 > 0$, that $M_1(\texttt{c\_1}) > 0$, hence that there is a pending call to `c_1`(). So the dispatch of `c_1`() updates the state to $((v^e_{\texttt{main}}, d), M_1, M_2)$ where $M_1 = [\![\texttt{machine}, \texttt{timeroff}, (\texttt{c\_1})^{n_1-1}, (\texttt{c\_2})^{n_2}]\!]$ (`machine`() has been posted during the dispatch of `c_1`()), $M_2 = \emptyset$ and d maps `error`, `timer`, `0c1`, `0c2`, `c1_eq_0`, `c2_eq_0`, `cloc` to false, on, 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 $n_1 = 0$. In that case the instruction is not enabled and the 2CM is "stuck" in the state $\langle l_x, n_1, n_2\rangle$. In the RTAP, 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 $n_1 = 0$, hence $M_1(\texttt{c\_1}) = 0$ (there is no pending call to `c_1`()). (the same holds for dec_2)

•$\mathsf{TypeInst}(l_x) = \mathsf{zerotest}_1$ and is of the form $l_x\colon c_1 = 0$ **then goto** l' **else goto** l''. We consider two cases: $n_1 = 0$ and $n_1 \neq 0$.

If $n_1 = 0$ then the 2CM updates its state to $\langle l', n_1, n_2\rangle$.

In the RTAP, the execution of `machine`() goes as follows: the conditional of the first line fails and the piece of code for the $\mathsf{zerotest}_1$ case is executed.

- the *then* branch is taken. (this is a faithful simulation). The dispatch of `machine`() sets `c1_eq_0` to true and sets `cloc` to l'. We show that a time transition will eventually take place.

  We conclude from $n_1 = 0$, that $M_1(\texttt{c\_1}) = 0$, hence, at this point, the state of the RTAP is of the form $((v^e_{\texttt{main}}, d), M_1, M_2)$ where $M_1 = [\![\texttt{timeroff}, (\texttt{c\_2})^{n_2}]\!]$ and d maps `error`, `timer`, `Oc1`, `Oc2`, `c1_eq_0`, `c2_eq_0`, `cloc` to false, on, false, true, false, false and l' (because `cloc` has been assigned to l' and `c1_eq_0` has been set to true during the dispatch of `machine`()). By Fact 6 we find that each pending call to `c_2`(), if any, should be dispatched before `timeroff`() for otherwise `error` is set to true. The valuation d given above shows the dispatch of a pending call to `c_2` yields the statement $\mathsf{future}(\texttt{c\_2}, 1)$ to be executed. Eventually, whenever the multiset of current pending calls is $[\![\texttt{timeroff}]\!]$ then the dispatch of `timeroff`() occurs and it resets the `c1_eq_0` to false. A time transition now takes place since the multiset of current pending calls is empty. As seen in Fact 6, the first dispatch immediately after the time transition should be `timeron`() which post `machine`() and updates the state to $((v^e_{\texttt{main}}, d), M_1, M_2)$ where $M_1 = [\![\texttt{machine}, \texttt{timeroff}, (\texttt{c\_2})^{n_2}]\!]$ (`machine`() is reposted by `timeron`() and $M_1(\texttt{c\_1}) = n_1 = 0$ because no `c_1`() has been copied to the new time step, $M_1(\texttt{c\_2}) = n_2$ because each call has been copied from the previous step); $M_2 = \emptyset$ (because of the time transition) and d maps `error`, `timer`, `Oc1`, `Oc2`, `c1_eq_0`, `c2_eq_0`, `cloc` to false, on, false, false, false, false and l' (because `cloc` has been assigned to l'), respectively.
- the *else* branch is taken. (this is an unfaithful simulation) We conclude from $n_1 = 0$, 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 $n_1 \neq 0$ then the 2CM updates its state to $\langle l'', n_1, n_2\rangle$.

In the RTAP, the execution of `machine`() goes as follows: the conditional of the first line fails and the piece of code for the $\mathsf{zerotest}_1$ case is executed.

– the *then* branch is taken. (this is an unfaithful 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 step. We conclude from $n_1 \neq 0$, 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 valuation at the time of dispatch is such that c1_eq_0 is true.
– the else branch is taken. (this is a faithful simulation). We conclude from $n_1 \neq 0$, 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''. Fact 8 shows that if the next dispatch to occur is not c_1() then error is set to true. The dispatch of c_1() updates the state to $((v^e_{\texttt{main}}, d), M_1, M_2)$ where $M_1 = [\![\texttt{machine}, \texttt{timeroff}, (\texttt{c\_1})^{n_1}, (\texttt{c\_2})^{n_2}]\!]$ (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, on, 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. Our simulation is based on a guess the outcome of the test. Above, the dispatch of c_1() is required to validate the guess was correct. Otherwise error is set to true.

(the same holds for zerotest$_2$)

Notice that the "temporary" location dest is required to hold the next location in cases that require validation through running of c_1 or c_2. This concludes the simulation of the 2CM by the RTAP.

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

As an immediate consequence of the above encoding we also find that the boundedness checking problem that asks, given a RTAP, if there exists a finite value that bounds the size of the multisets of pending calls at every point in time is undecidable. It also naturally follows that liveness properties are undecidable for this model.

Corollary 1. *The boundedness and the liveness checking problem for* RTAP *are undecidable.*

5 Discussion

In the standard "untimed" model for event-driven systems [12,8], timers are abstracted away. This can lead to false alarms in the analysis, as we demonstrate through the example at Fig. 7. The procedure timeout (present in event-driven programming APIs such as libevent [9]) has the following intended semantics: if a particular event occurs before the timer reaches the timeout value (given by the last parameter) then the handler given by the first argument is executed, otherwise if the event does not occur and the timer reaches the timeout value, the handler given by the second argument is posted. The procedures untimed_timeout and

```
global b;

main() {
    b=0;
    timeout(h1,h2,1);
    timeout(h1,h3,2);
}

h1() {}

h2() { assert(b==0); }

h3() { b=1; }
```

```
untimed_timeout(task h, task h', int ts) {
    if(*)
        future(h,0);
    else
        future(h',0);
}

timed_timeout(task h, task h', int ts) {
    if (*)
        choose i in {0,...,ts-1}
        future(h,i);
    else
        future(h',ts);
}
```

Fig. 7. The assertion does not fail if `timed_timeout` implements `timeout` but can fail if `untimed_timeout` implements `timeout`

`timed_timeout` give implementations of `timeout` in the untimed and timed settings, respectively. We abstract the occurence of the event by a non-deterministic choice (`*` in the conditional). For this program, there is no assertion violation when `timed_timeout` implements `timeout`, because there is no execution in which `h2` is executed after `h3`. However, this timing behavior is lost in the implementation `untimed_timeout`, where the scheduler could dispatch `h2` before `h3` and the assertion can fail. Unfortunately, Theorem 2 shows that safety verification is undecidable if we assign timing constraints to posted calls.

Acknowledgments. We thank Tom Henzinger for suggesting this problem.

References

1. Aho, A., Sethi, R., Ullman, J.: Compilers: Principles, Techniques, and Tools. Addison-Wesley, Reading (1986)
2. Atig, M.F., Bouajjani, A., Touili, T.: Analyzing asynchronous programs with preemption. In: FSTTCS 2008: Proc. 28th Int. Conf. on Fondation of Software Technology and Theoretical Computer Science (2008)
3. Ganty, P., Majumdar, R., Rybalchenko, A.: Verifying liveness for asynchronous programs. In: POPL 2009: Proc. 36th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages, pp. 102–113. ACM Press, New York (2009)
4. Ghosal, A., Henzinger, T.A., Kirsch, C.M., Sanvido, M.A.A.: Event-driven programming with logical execution times. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 357–371. Springer, Heidelberg (2004)
5. Henzinger, T.A., Horowitz, B., Kirsch, C.M.: Giotto: A time-triggered language for embedded programming. In: Henzinger, T.A., Kirsch, C.M. (eds.) EMSOFT 2001. LNCS, vol. 2211, pp. 166–184. Springer, Heidelberg (2001)
6. Henzinger, T.A., Kirsch, C.M.: The embedded machine: predictable, portable real-time code. In: PLDI 2002: Proc. 23rd Conf. on Programming Language Design and Implementation, pp. 315–326. ACM Press, New York (2002)

7. Henzinger, T.A., Kirsch, C.M., Majumdar, R., Matic, S.: Time safety checking for embedded programs. In: Sangiovanni-Vincentelli, A.L., Sifakis, J. (eds.) EMSOFT 2002. LNCS, vol. 2491, pp. 76–92. Springer, Heidelberg (2002)
8. Jhala, R., Majumdar, R.: Interprocedural analysis of asynchronous programs. In: POPL 2007: Proc. 34th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages, pp. 339–350. ACM Press, New York (2007)
9. Libevent, `http://www.monkey.org/~provos/libevent/`
10. Minsky, M.: Finite and Infinite Machines. Prentice-Hall, Englewood Cliffs (1967)
11. Mitchell, J.: Foundations for Programming Languages. MIT Press, Cambridge (1996)
12. Sen, K., Viswanathan, M.: Model checking multithreaded programs with asynchronous atomic methods. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 300–314. Springer, Heidelberg (2006)
13. Titzer, B.L.: Virgil: objects on the head of a pin. In: OOPSLA 2006: Proc. 21st ACM-SIGPLAN conference on Object-oriented programming systems, languages, and applications, pp. 191–208. ACM Press, New York (2006)