Analyzing Real Time Linear Control Systems Using Software Verification

Parasara Sridhar Dugirala  
Department of Computer Science and Engineering  
University of Connecticut  
psd@engr.uconn.edu

Mahesh Viswanathan  
Department of Computer Science  
University of Illinois at Urbana Champaign  
vmahesh@illinois.edu

Abstract—Deployed embedded software interacts with sensors and actuators to control a physical environment. While the evolution of the control system is specified by Ordinary Differential Equations (ODEs), the embedded software periodically senses the state of the system, performs computation over the inputs, and initiates the actuators based on the result of computation. In this paper, we present a bounded time safety verification technique for periodically actuated linear control systems. The model considered in this paper takes into account that the control tasks are executed on a real time operating system and hence the task, in some instances misses the real time deadlines. Using matrix exponentiation, and symbolic evaluation of inputs, we reduce the verification problem of such systems into software verification with computation over reals. We compare different techniques for verifying such software, highlight the merits of each of the approaches, and present our experimental results.

I. INTRODUCTION

Software controlling physical processes often execute on embedded platforms to achieve a number of safety critical tasks, like braking, fuel injection, and stability control in automotive systems. The evolution of the physical process is governed by ordinary differential equations/inclusions, and the software senses the physical state through sensors, and controls the physics by setting the parameters of actuators. The execution platform for such systems is often uni-processor, but it nonetheless runs several applications “concurrently”. The presence of competing claims to processing time, introduces variability in the controller software’s response time due to blocking, and preemption enforced by a scheduling policy. These timing effects, though critical to the functional correctness of the controller, are often ignored during design and analysis, wherein one assumes sensing without jitter, and actuation without delay.

The variability in the execution time of the control software makes design and analysis of such systems extremely challenging. To limit the nondeterminism in the timing behavior, several people have advocated a design and execution model that enforces determinism by combining timing analysis of the software with logical execution time (LET) [20]. In this model [27], [32], the physical plant is sensed periodically at a fixed period, new inputs to the actuator are computed, and physical process is actuated at a fixed $W$ time units after the start of the period; here $W$ is taken to be a bound on the worst-case execution time (WCET) of sensing, computation, and actuation. More recently [17], it has been argued that since WCET analysis can be extremely conservative, it is better to instead to use typical worst-case analysis (TWCA) instead. In such a model, the actuation time of the physical process is taken to be TWCRT (typical worst-case response time) of the controller. If the sensing, computation, and actuation in a certain period takes longer than the computed TWCRT, then the physical process is not actuated in that cycle, and the process evolves according to the inputs computed in the previous cycle; the TWCA ensures that such “missed deadlines” are rare, well understood, and is captured in the formal model of the system that is analyzed.

In this paper, we consider the problem of verifying safety properties of controller software operating in the execution environment outlined above. In [17], the analysis was carried out by modeling the physical process, controller software, sensing and actuating timing model as a single closed loop hybrid automaton [19], and then performing reachability analysis using a model checker like SpaceEx [16]. However, such an analysis can be overly conservative, as we illustrate through an example.

Motivating Example: [Adaptive Cruise Control System] Consider two cars in a leader-follower system where the leading car is moving at a constant speed $v_f$ and the trailing car senses the environment variables, namely the separation between the two cars $s$, its velocity $v$, and its acceleration $a$. The trailing car uses an automatic cruise control mechanism for maintaining safe separation with the leading car. The control program computes the output, namely, the rate of change of acceleration. The differential equations with inputs for such system is given as:

\[
\dot{s} = (v_f - v) \\
\dot{v} = a - k_a \times v \\
\dot{a} = u
\]

In [30], the authors provide a feedback law that requires sensing all the continuous variables and prove that it is
Consider a modification of the control law which requires sensing only the acceleration and velocity, since 
\[ u = -2a - 2(v - v_f) \]  
depends only on the acceleration and velocity. One can verify the safety of the new control law using similar techniques stated in [30], however, in practice, the control mechanism applied is not continuous. The control program that periodically senses values of \( v \), \( a \), \( v_f \) and \( v \) to compute the output \( u \) for every time period \( T \) is given in Figure 1.

For simplicity, let us assume that the actuation time is actually the WCET (and not TWCRT) with instantaneous sensing, actuation, and computing. Hence, the system is actuated for the time period \( (T = 0.01) \) with the values of actuation parameters that are computed at the beginning of the time period. Hybrid automaton that models such periodically actuated systems is given in Figure 2. One technique for analyzing such systems is to compute reachable set using state of the art tools such as SpaceEx. The result of verifying hybrid automaton in Figure 2 using support functions in SpaceEx [16] is given in Figure 3.

The simulations indicate stable behavior of the system but the reachable set computes a coarse overapproximation of the reachable set of states. To understand why such analysis is conservative, let us inspect how it works on this example. The reachable set computation algorithm performs the following operations iteratively. Assuming that the set of states reachable within time \( kT \) (ReachSet) has been computed, the algorithm will compute the set of inputs \( \mathcal{U}_k = \{u = -2a - 2(v - v_f)\} \) \((s, v, a) \in \text{ReachSet}\), for the next interval \([kT, (k+1)T]\), and compute all states reachable in the time interval \([kT, (k+1)T]\) for this set of inputs.

Consider two states \((s_1, v_1, a_1)\) and \((s_2, v_2, a_2)\) in ReachSet and let \( u_1 = -2a_1 - 2(v_1 - v_f) \) and \( u_2 = -2a_2 - 2(v_2 - v_f) \) be their corresponding inputs according to the control law. The reachability computation technique, by computing the set \( \mathcal{U}_k \), applies both the inputs \( u_1 \) and \( u_2 \) to both the states \((s_1, v_1, a_1)\) and \((s_2, v_2, a_2)\) to compute an overapproximation of the reachable set for time interval \([kT, (k+1)T]\), which results in a coarse overapproximation. However, it suffices to apply the input \( u_1 \) to the state \((s_1, v_1, a_1)\) and \( u_2 \) to the state \((s_2, v_2, a_2)\). Current hybrid system verification tools do not preserve this relationship between the inputs and their corresponding states.

In this paper, we propose a technique for analyzing real time linear control systems where the dynamics of the plant are given as linear ODEs. In order to handle the challenges of the reachability approach and improve the efficiency of verification, we reduce the problem to software verification with computation over reals. We make the following observations that aid in the reduction procedure:

First, in order to handle the inputs efficiently, we consider inputs as symbolic functions of the state variables and preserve the relation between inputs and the state of the system. Second, we observe that control software running on a real time operating system can be analyzed as piecewise affine systems. Third, the closed form solution for piecewise affine systems can be obtained by computing fixed matrix exponential, resulting in a linear function that can be efficiently analyzed. Finally, we observe that the timing behaviors of such control tasks can be encoded as a series of if-then-else statements in a program and hence the verification of closed loop linear control system can be reduced to the software verification problem. This reduction has the advantage that we can use traditional software analysis techniques to address the problem. Further we are no longer confined to only analyzing the system for bounded time. Abstract interpretation or loop invariant based verification techniques could potentially be used for checking the safety of the system for unbounded time\(^1\). We conclude this section by observing that though the physical plant dynamics are assumed to be given by linear ODEs, the systems we analyze are not purely linear. This is because the controller software could have nonlinear operations. Thus, certain nonlinear dynamical systems with continuous feedback can be analyzed in our setup.

We have built a prototype tool based on these ideas. We analyze the transformed software that we construct for linear control systems, using abstract interpretation and satisfiability modulo theories (SMT). Our preliminary experimental results show that approach analyzes systems quickly. It also highlights the relative merits of abstract interpretation and SMT in this context.

---

\(^1\)Reachable set computations can sometimes converge to a fix-point, and then be able to prove safety for an unbounded time. However, they typically don’t try to overapproximate, like widening in abstract interpretation and loop invariant synthesis, and therefore, are not engineered to reach a fix-point.
II. Preliminaries

A. Execution Model

The controller software is assumed to run on an embedded uni-processor platform that has other applications running as well. The controller needs to sample the state of the physical process through sensors so that the right actuator parameters can be set to control the physical plant. The competition for resources causes sampling jitter (varying time for the sensor data to be read by the software) and response time jitter (varying time for the software to compute the actuator parameters). It is important to account for these timing effects, but these variations make the design and analysis very complicated.

In order to reduce the nondeterminism in the timing behavior, several authors have proposed to exploit timing analysis with logical execution times (LET). Recently [17], a model combining TWCA and LET has been proposed as an appropriate model for industrial control software. In this model the control software is executed periodically, every \( T \) time units, as shown in Figure 4. Typical worst-case analysis (TWCA) of the software obtains a typical worst-case response time (TWCRT) \( W \), and error bounds on the number of violations of the TWCRT within a given time window. At the start of every period, the state of the physical process is sensed, new actuator parameters are computed based on the physical state sensed, and the new input parameters are actuated at time \( W \), provided the sensing and computation take less than \( W \) time (time periods 1 and 2 in Figure 4). If sensing and computation take longer than \( W \) units, then the current computation is discarded, and the inputs computed in the previous time period continue to control the physical plant (time period 3 in Figure 4); how frequently such misses can happen can be obtained from the TWCA, and this is captured in the formal model that is analyzed.

B. Physical Process Model

The dynamics of the physical process being controlled by the software will be assumed to be governed by a ordinary differential equation (ODE). In this paper, we consider the system to be a linear differential equation as given in Equation 1 where \( x \), an element in \( \mathbb{R}^n \) represents the state of the environment/physical process and \( u \), an element in \( \mathbb{R}^m \), represents its inputs/actuator parameters.

\[
\dot{x} = Ax + Bu. \tag{1}
\]

The solution of ODE in Equation 1 is given as a function \( \xi : \mathbb{R}^n \times \mathbb{R}^m \times \mathbb{R}_{\geq 0} \rightarrow \mathbb{R}^n \) which satisfies the condition: \( \forall x_0 \in \mathbb{R}^n \) and \( \forall u \in \mathbb{R}^m \), \( \forall t > 0 \), \( \frac{d}{dt}\xi(x_0, u, t) = A\xi(x_0, u, t) + Bu \). Such function is called a solution or a trajectory. For linear ODEs, the closed form expression for the trajectory is given as:

\[
\xi(x_0, u, t) = e^{At}x_0 + \int_0^t e^{A(t-\tau)}Bu \, d\tau. \tag{2}
\]

where \( e^{At} \) is the matrix exponential. A trajectory \( \xi \) defined over a finite time horizon \([0, t_f]\), is said to be a finite trajectory where \( t_f \) is denoted as \( \xi.\text{dur} \). For finite trajectories, the state \( \xi(x_0, u, 0) \) is denoted as \( \xi.\text{fstate} \) and \( \xi(x_0, u, t_f) \) is denoted as \( \xi.\text{lstate} \). An execution of the environment is obtained by concatenating two or more finite trajectories. The inductive definition of an execution is given as:

**Base Case:** \( \alpha = \xi \), where the duration of \( \alpha \) is defined as \( \alpha.\text{dur} = \xi.\text{dur} \) and the state at time \( t \) as \( \alpha(t) = \xi(t) \).

**Inductive Case:** \( \alpha = \alpha_1.\xi \) with \( \alpha.\text{dur} = \alpha_1.\text{dur} + \xi.\text{dur} \) and \( \alpha(t) = \text{if} \ (t < \alpha_1.\text{dur}) \ \text{then} \ \alpha_1(t) \ \text{else} \ \xi(t - \alpha_1.\text{dur}) \).

The first and last states of an execution, denoted by \( \alpha.\text{fstate} \) and \( \alpha.\text{lstate} \) are defined similarly. One can intuitively think of a trajectory as the behavior of system 1 with constant input, and an execution as the behavior of system 1 with piecewise constant inputs with possible discontinuities in environment due to autonomous or impulse transitions.

C. Controller Software

A control program has three types of variables given as follows: \( v \) denote its internal variables, \( x_s \) denote the variables that store the sensor recordings of environment, and \( u_{out} \) denote output variables that act as actuation values to the environment. In an execution of the control
control_program ::=  
<initialize v>  
while(true)  
x_in = sense_environment(); // Sensing  
Stmt+ // Computation  
actuate_system(u_out); // Actuation  
endwhile;

Fig. 5: Structure of control programs considered in this paper; sense_environment and actuate_system are special functions for sensing and actuation.

Stmt ::= var <ID>;  
| ID = RealExpr;  
| if(BoolExpr) then Stmt+  
| elseif(BoolExpr) then Stmt+*  
| else Stmt+*  
| endif;  
| while(BoolExpr) do Stmt+  
| endwhile;

Fig. 6: Syntax of statements in control program.

The internal variables v are initialized, then after every T time units (that is the periodicity of the control task), the state of environment is sensed and stored in variables x_s. Then the program is executed, thereby changing the values of v and u_out. In this paper, we assume that the timing behavior of this computation is provided by the Typical Worst Case Analysis (TWCA) with Typical Worst Case Response Time (TWCRT) of this computation given as W. Hence, if the computation terminates within W time units, the values u_out are given as actuation values, i.e. u, and the environment evolves according to ODE in Equation 1 with u as the constant input until the next actuation happens. This sequence of sensing, computation, and actuation repeats. The computation model in this paper makes the following realistic assumptions.

1) The control program is a periodic task with time period T.
2) The Typical Worst Case Response Time (TWCRT) is given as W.
3) The timing behavior, i.e., the deadline misses, of the periodic task is given by the Typical Worst Case Analysis (TWCA) of the control program.

The structure of control program considered in this paper is given in Figure 5. The syntax of statements in program (Stmt) is given in Figure 6.

RealExpr is the set of arithmetic expressions over the inputs, program variables, and real constants. BoolExpr is the set of boolean expressions, obtained by conjunction and disjunctions of comparison (==, !=, >=, >, <=, < denoting the respective operations) over arithmetic expressions. For performing TWCA, it is a common requirement that all the loops in the program should have explicitly known bounds. Hence the while statements are syntactic sugar for finite number of if-then-else statements.

As control program interacts with the environment every T time units, the execution of control program is given as the sequence (x_s^0, v^0, u^0_out), (x_s^1, v^1, u^1_out), ... where v^0 is the initial valuation of the internal variables, x_s^0 is the initial state of the environment and u^0_out = 0. At the j^{th} iteration of the control program, x_s^j denotes the sensed state of the environment, v^j represents the internal variables, and u^j_out is obtained by (v^j, u^j_out) = [P](v^{j-1}, x_s^{j-1}) if the task deadline of TWCRT is met or u^j_out = u^j-1_out if it is not met. Here, [P] represents the result of the computations performed by the control program. We assume that the internal variables v get updated even in case the task misses the TWCRT deadline.

As described earlier in the paper, we reduce the verification of such embedded control systems to the problem of software verification. Verification conditions over programs would involve predicates that are satisfied by the program variables after the execution of a given statement. The syntax of programs with verification conditions would have additional syntax as

Stmt ::= ...  
| Assume(BoolExpr);  
| Assert(BoolExpr);

The statement Assert(BoolExpr) asserts that the current program configuration satisfies the boolean condition specified. The statement Assume(BoolExpr) nondeterministically selects the value of the variables involved in a way that boolean condition is satisfied. A program with Assert and Assume statements is defined to be correct if and only if for all nondeterministic choices made by the program during the assume statements, all the assertions provided by the assert statements hold.

Given an environment that satisfies Equation 1, a control program P, time period T, Typical Worst Case Analysis TWCA, TWCRT W, we denote the closed loop system as A = [P, A, B, T, W, TWCA]. An execution of the closed loop system starting from initial environment state x_0 and initial valuation of program variables v_0, is an execution obtained by concatenating trajectories ξ_0 · ξ_1 · ξ_2 ... such that

1) ξ_0 is a trajectory of the system in Equation 1 with input v_0 = 0, ξ_0.fstate = x_0 and ξ_0.dur = W. That is, the first scheduling of the task will meet the deadline.
2) ∀i ∈ N_+, ξ_i is a trajectory of the system in Equation 1 with input u_i, ξ_i.dur = T, and ξ_i.fstate = ξ_{i+1}.fstate.
3) ∀i ∈ N_+, u_i, the input for trajectory ξ_i is obtained as <v_i, u_i> = [P](v_{i-1}, ξ_{i-1}(t_s)). Where t_s is the time between two successful actuators of the control program, given by TWCA.

We denote such an execution of A as α_A(x_0, v_0) and its state after time t as α_A(x_0, v_0, t). Figure 4 gives an instance of a control task with TWCA. We note that this model not only captures the scheduling aspect of control tasks in real time systems, but also the delay between
sensing and actuation caused due to computation.

Definition 1: Given a system in Equation 1, a controller code $P$ with time period $T$, Typical worst case analysis TWCA, TWCRT $W$, initial set of states $\Theta$, initial set for internal program variables $V$, unsafe set of states $U$, and discrete steps $N$, the closed loop system $A = [P, A, B, TWCA, W]$ is said to be safe if and only if $\forall k \in \mathbb{N}, 0 \leq k \leq N, \forall x_0 \in \Theta, \forall v_0 \in V$, the state of the execution at time $k \times T + W$, $\alpha_A(x_0, v_0, k \times T + W) \cap U = \emptyset$.

Given an execution of $\alpha_A$ starting from $\Theta$ and $V$, where $\alpha_A = \xi_0 \cdot \xi_1 \ldots$ with the duration of $\xi_1$ is $T$. To infer the safety of $\alpha_A$ it suffices to check $\forall i \leq N, \xi_i.fstate \cap U = \emptyset$.

Remark 1 Although we consider bounded number of steps for safety verification, one can trivially extend the definition to unbounded number of steps by checking that $\forall k \geq 0, \alpha_A(x_0, v_0, k \times T + W) \cap U = \emptyset$. Also, note that our definition of safety verification does not check for safety of the intermediate states between time instances $k \times T + W$ and $(k+1) \times T + W$. In Remark 2, we present a sub-class of linear systems for which our technique can verify safety of the intermediate states as well. For more general systems, using Lipschitz bounds and discrepancy functions [10], one can infer safety of the intermediate states from the safety at discrete time instances.

III. Technique

In this section, we present our reduction of closed loop embedded control systems to software verification. For a closed loop system $A$ with linear ODEs representing the dynamics of environment, we construct a program $\text{transform}(A)$ with assume and assert statements, such that if $\text{transform}(A)$ is safe, then the closed loop system is safe. We now present the different steps in constructing $\text{transform}(A)$.

1) Analytical Solutions for Linear ODEs: For linear ODEs of the format given in Equation 1, the closed form expression for trajectory is given in Equation 2. Observe that the closed form expression involves matrix exponentials, which is an infinite sum and cannot be computed exactly for any given arbitrary matrix. However, for periodically actuated systems, we make two observations that makes the verification tractable. First, for a given time $T$, $e^{A T}$ can be computed efficiently (numerical accuracies for matrix exponential will be discussed in Section III-A1). Second, the actuation happens only at the actuation instances and hence the input $u$ is constant for the $T$ time units. Thus, the integral term that accounts for the change of state due to input $u$ can also be efficiently computed. Combining these two observations, we get that the state of the system after $T$ time units is given as $\xi(x_0, u, T) = e^{A T}x_0 + G(A, T)Bu$ where $G(A, T) = \sum_{i=0}^{\infty} \frac{A^{i+1}T}{(i+1)!}$.

The numerical values of $e^{A T}$ and $G(A, T)$ give the discrete step transition for the environment. A positive side effect of this discrete solution is that $e^{A T}$ and $G(A, T)$ are both constant matrices for given values of $A$ and $T$. Hence, using these numerical values, we can compute the relation between $\xi(x_0, u, T)$, $x_0$, and $u$. This relation between the input $u$ and corresponding state $x_0$ helps us improve the accuracy of analysis. Moreover, this relationship is linear, and hence can be efficiently analyzed.

2) Handling Timing Behaviors: In this paper, we consider the timing behaviors to be given by Typical worst case analysis TWCA. The complete description of TWCA is beyond the scope of this paper, so we provide a brief overview of the analysis. TWCA allows the real time tasks to miss the deadlines, however, provides a bound on the number of times such missed happen. Consider the TWCA provided in Table I. The TWCA model guarantees that there would be at most 1 deadline misses in 3 consecutive executions and at most 2 deadline misses in 5 consecutive executions of the given task.

To model such behavior in software we maintain the history of deadline misses and nondeterministically choose whether the current deadline will be missed or not. Example code that tracks all possible deadline misses according to the TWCA given in Table I is provided in Figure 7. The code in Figure 7 is expected to be executed at the beginning of every period. The boolean variables $d_1, d_2, d_3, d_4, d_5$ maintain the history of deadline misses for last 5 periods of the task; $d_1$ is 1 iff the deadline was missed $i$ periods before. The $\text{Assume}$ statement decides whether the deadline in the current period is going to be missed by nondeterministically picking a value for $d_1$. The if-$\text{then-else}$ statements check the required conditions and allow the deadline to be missed only if the nondeterministic choice of $d_1$ is consistent with the TWCA model. We call the above program transformation of TWCA as $\text{Prgm}(TWCA)$. At the end of $\text{Prgm}(TWCA)$, the deadline is met if and only if $\text{deadline\_met} == 1$.

<table>
<thead>
<tr>
<th># deadline misses</th>
<th>consecutive executions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
</tbody>
</table>

TABLE I: Example of a typical worst case analysis model of a task. The table provides the maximum number of misses than can happen in the consecutive executions.
Fig. 8: transform(\(A\)) that performs safety verification of closed loop systems using the matrix exponential and Prgm(TWCA).

A. Transformed Program

We now present transform(\(A\)) by combining the techniques in Sections III-1 and III-2. The transformation essentially generates a program with bounded loop where each iteration of the loop corresponds to the trajectory \(\xi_i\) in the execution of the closed loop system. In the body of the loop, we assign to \(x_{\text{next}}\), the last state of trajectory \(\xi_i\) using the numerical computation of matrix exponential. For checking the safety of the system at these time instances, assert statements with safety predicates on \(x_{\text{next}}\) are added in the program. For handling the timing behaviors and computing the new actuation values according to the missed deadlines, in the transformed program, the actuation parameters \(u_a\) are updated only when the results of computation would be actuated according to the TWCA. The formal definition of this program is given in Definition 2

\textbf{Definition 2}: Given a closed loop system \(A = [P, A, B, T, W, TWCA]\), discrete steps \(N_h\), initial states \(\text{InitSet}\), initial valuations of program variables \(\text{InitVals}\), and unsafe set \(U\), the transform(\(A\)) program for the safety verification is defined in Figure 8.

Intuitively, the program transform(\(A\)) simulates the continuous evolution of the environment and the computations in the control program for \(T\) time units. Assume that at the beginning of the loop, \(x\) represents the state of the environment and \(u\) represents the input to the environment. Updating the variables \(x_{\text{next}}\) according to the numerical values of matrix exponential gives us the state of the environment after \(T\) time units. Prgm(TWCA) tracks the computations of the control program and checks whether a deadline is met or missed. It updates \(u\) only when the deadline is met. Observe that by computing matrix exponentials, the relationship between \(x, x_s, and u\), is maintained, which results in a more accurate analysis. We illustrate the transformation using an example.

\textbf{Example 1} Consider a control program for the adaptive cruise control as given in Figure 9 where \(\text{threshold}_1\) and \(\text{threshold}_2\) are parameters chosen by the programmer for performance tuning.

Consider the initial set defined as \((s = 100 \land v = 60) \land v >= 55 \land v <= 65 \land a == 0\) and the unsafe set defined as \((s <= 60 \lor v >= v_\text{f} + 10 \lor v <= v_\text{f} -10)\), and TWCA in Table I. For \(T = 0.1\), \(e^{AT}\) and \(G(A, T)\) computed using MATLAB are

\[
\begin{pmatrix}
1.0 & -0.0995 & -0.005 \\
0 & 0.99 & 0.0995 \\
0 & 0 & 1.0
\end{pmatrix}
\]

respectively. Also, \(B\) is a column vector given as \([0, 0, 1]^T\). For the sake of presentation, we omitted the first trajectory of the execution and also updating of \(x\). The transformed program for 2 discrete-time steps is given in Figure 10.

\[
\begin{align*}
[s_s, v_s, a_s, v_f_s] &= \text{sense_environment}(); \\
\text{if}(s_s >= \text{threshold}_1 \land v_f_s < v_s - 5) &\text{ then} \\
&\quad u = -3*a_s -3*(v_s - v_f_s) + (s_s - (v_s + 5)); \\
\text{else} &\text{ if}(s_s >= \text{threshold}_2 \land v_f_s < v_s -5) &\text{ then} \\
&\quad u = -2*a_s -2*(v_s - v_f_s); \\
\text{else} &\quad u = 0; \\
\text{endif}; \\
\text{actuate_system}(u);
\end{align*}
\]
\[ \input{fig10.eqns} \]

1) Handling Numerical Approximations: As matrix exponentiation involves an infinite sum, exact value of the exponent \( e^{Ah} \) and \( G(A,h) \) cannot be computed. The value of matrices obtain from MATLAB are numerical approximations of these quantities. However, various algorithms for sound numerical approximations of matrix exponentials are available in literature (refer to [25]). Using these techniques, one can compute estimate and error matrices \( \hat{E} \) and \( \tilde{E} \) for \( e^{Ah} \), and estimate and error matrices \( \hat{G} \) and \( \tilde{G} \) for \( G(A,h) \) such that \( e^{Ah} \in \hat{E} \pm \tilde{E} \), and \( G(A,h) \in \hat{G} \pm \tilde{G} \). Using the estimate and error matrices to compute \( x_{next} = e^{Ah} x + G(A,T)u \) gives us the expression \( x_{next} \in \hat{E}x + \tilde{E}x + \hat{G}u + \tilde{G}u \). To compute sound overapproximation of this operation, add two nondeterministic error variables \( \tilde{x} \) and \( \tilde{u} \) such that \( \tilde{x} \in [-\tilde{E}x, \hat{E}x] \) and \( \tilde{u} \in [-\tilde{G}u, \hat{G}u] \). To compute the range of these nondeterministic expressions, we first guess an invariant for the system, say \( I \), and assign range of \( \tilde{x} \) such that \( \forall \xi \in I, [-\tilde{E}x, \hat{E}x] \subseteq \text{range}(\tilde{x}) \) and similarly assign range for \( \tilde{u} \). Thus, for the invariant \( I \), we compute \( err_x = \{ \max \| \tilde{E}x \| \text{ for } x \in I \} \) and assign \( \tilde{x} \in [-err_x, err_x] \). Similar overapproximation is performed for \( u \). Hence the expression \( x_{next} = \tilde{E}x + \tilde{x} + \tilde{G}u + \hat{u} \) would be sound overapproximation for \( x_{next} = e^{Ah} x + G(A,h)u \).

Note that the invariant \( I \) chosen need not be precise enough to infer safety from the unsafe set \( U \). For example, in the case of adaptive cruise control example, selecting the invariant \( I = s \in [-1000, 1000] \cap v \in [0, 200] \cap a \in [0, 10] \) would bound the error matrices \( err_x \leq 0.01 \). Thus for \( v_f = 60 \), the invariant \( I \) does not prove safety from the unsafe set \( U \). For a given \( I \), we compute \( \tilde{x} \) and \( \tilde{u} \) using a given approximation method for matrix exponential. For using the obtained values of \( \tilde{x} \) and \( \tilde{u} \) for all iterations, one has to add the assertion \( x_{next} \in I \) at the end of every iteration to ensure that the error estimate is correct.

In practice, one can use multi-precision arithmetic to get arbitrarily precise \( \tilde{x} \) for a given invariant \( I \). However, the wrapping effect of these approximations might cause exponential increase in errors. We note that for stable numerical systems, these errors do not accumulate as the eigenvalues of \( e^{Ah} < 1 \). As the computed values are sound numerical approximations, the soundness property from Theorem 1 is still preserved, but the completeness property does not hold. That is, if \( \text{transform}(A) \) is correct, then \( A \) is safe, but if \( \text{transform}(A) \) is unsafe, we can no longer conclude that \( A \) is unsafe. However, if the property violation of \( \text{transform}(A) \) provides a counterexample, one can use guaranteed numerical integration engines like CAPD\(^2\) to validate the counterexamples.

Remark 2 For special forms of linear ODEs (for example, when the matrix \( A \) is nilpotent), the solution of the ODE admits a polynomial closed form solution (say \( (x_0, u, t) = F(x_0, u, t) \)). For such systems, one can not only verify the safety at discrete time instances, but also for the entire duration of trajectory, i.e., \( \forall t < N_h \), \( \forall x_0 \in \Theta \), \( v_0 \in V \), \( \alpha_A(x_0, v_0, t) \cap U = \emptyset \). Checking this would require the following modifications in \( \text{transform}(A) \). First, an additional variable \( t \) is added to \( \text{transform}(A) \) (which represents the time) and its value is nondeterministically chosen from the range \([0, T]\). Second, state variables \( x_{reach} \) representing the state of system at time \( t \) are added along with the statements that assert \( x_{reach} = F(x, u, t) \). Finally, for checking the safety of these intermediate states, the assertion \( x_{reach} \cap U = \emptyset \) is added to the program. The transformed program now not only verifies the safety at discrete time instances, but also at intermediate time values as well. Moreover, for such cases, the matrix exponential is computed exactly and the analysis is not only sound, but also relatively complete.

Having seen the transformation from closed loop embedded control system to software verification, in the next section, we will look at two main techniques commonly used in software verification and apply them for verification of transformed program.

\(^2\)http://capd.ii.uj.edu.pl/index.php
IV. Verifying Transformed Programs

Abstract Interpretation: Abstract Interpretation, originally proposed in [7] is one of the most popular techniques for verifying programs. In this technique, an abstract domain is chosen to overapproximate the set of states and the statements in program are considered to be transformers on these abstract domains. The efficiency and precision of analysis depends on the domain used for analysis. To perform the analysis of loops, the behavior of the program is approximated by computing the fixpoints of these transformations over the domains, typically using a widening operator [8]. One of the drawbacks of this analysis is that a counterexample cannot be generated when the program violates the correctness property. In this paper, we used Interproc abstract interpretation tool, built on top of Aapron Library [21] for verifying the correctness of transformed programs.

Bounded Model Checking Using SMT Solvers: Recent progress in the efficiency of SMT solvers [9], [11] has made them an attractive tool for checking bounded time properties of software and hardware systems. An SMT solver takes as input a formula (boolean combinations of assertions on variables in a theory) and proves whether the formula is satisfiable or unsatisfiable. Operations over variables in a program are encoded as pre and post conditions in Floyd-Hoare logic. For dealing with loops, a loop invariant has to be explicitly provided. Since finding these loop invariants is an undecidable problem, bounded loop unrolling with variable renaming is performed and each of the statements is then encoded in Floyd-Hoare logic. If all the assertions in the program are satisfied, then the program is correct. For a formula that is not satisfiable, SMT solvers return a model that violates the formula which corresponds to a given execution in the program. This ability to generate counterexamples is an advantage of SMT solvers over abstract interpretation techniques. However, without a loop invariant, SMT solvers can only handle bounded loops, which is a drawback compared to the abstract interpretation technique. For checking loops with unknown bounds, loop invariants have to be automatically synthesized before encoding them into SMT formula. In this paper, we used Z3 [9] as the back end SMT solver for bounded model checking (BMC).

V. Experiments

To test the validity of our approach, we consider a set of benchmark examples of linear control systems and apply abstract interpretation techniques and bounded model checking using SMT for checking correctness of transformed programs. In order to compare with the reachable set computation approaches, we also compare the running time with SpaceEx [16]. Given a control program, and the linear dynamics of environment, our prototype tool generates the transformed program as described in Section III-A. For all the experiments in this paper, we only take into account the WCET to present the relative merits and demerits of each of the approaches.

Our experimental evaluation has two parts. In the first part of the evaluation, we compare the results of bounded model checking using Z3 with abstraction interpretation techniques using Interproc and state of the art hybrid systems verification tool SpaceEx. After observing that BMC using Z3 gives the most accurate verification results very quickly, we evaluate a suite of benchmarks using this approach and present the experimental results. All experiments were performed on an Intel i7-Quad core machine with 8GB memory.

A. BMC vs Abstract Interpretation vs SpaceEx

To evaluate the relative merits of analyzing the transformed program using Interproc, Z3, and hybrid systems verification tool SpaceEx, we consider Example 1 and its simplification (referred as kinematic system). For analyzing the precision of analysis, we consider two variants of adaptive cruise control system as ACC1 and ACC2 and two variants of kinematic system as Kinematic1 and Kinematic2 respectively. The unsafe set considered in ACC1 is closer to the reachable set than ACC2 and the unsafe set in Kinematic1 is closer to the reachable set than Kinematic2. Bounded loop unrolling of the transformed program are given as input to SMT solver for verification. For analysis using Interproc, we used the box, octagon and polyhedral domain for analysis. The reachable set computation in SpaceEx used support functions with box, oct and uni32 options. The results are given in Table II.

As expected, the performance and the accuracy of the abstract domain are inversely related. Polyhedral domain, which could prove safety in all 4 cases took couple of orders of magnitude more time than other domains. We observed that in octagon domain, the octagon representing the reachable set had all the variables in the program, which leads to the poor quality of overapproximation. Similarly the performance of SpaceEx depended on the domain chosen for support functions. An interesting observation is that oct could prove safety of Kinematic2 example which uni32 failed. We hypothesize that this is because of linear relations between the inputs and state variables being preserved in the oct constraints during the discrete transitions. Observing the trade off between accuracy and efficiency, we prefer encoding the program correctness as SMT instance over using abstract interpretation. Although box domain could prove the safety of the system in ACC2 and Kinematic2, we prefer encoding the program correctness as SMT formula because it has the advantage of providing a model which violates the property and in this case, it “approximately” corresponds to the execution of the closed loop system.

3Script files for experiments can be downloaded from http://engr.uconn.edu/~psd/closed-loop-verification/
Where $d_s$ is the elongation of spring of restitution $k$ connecting locomotive and car, $v_l$, $m_l$ and $v_c$, $m_c$ represent the velocity and mass of locomotive and car respectively, $b$ represents the damping coefficient, and $c$ represents the aerodynamic friction. In [24], the authors give a controller for $F$ with the control objective that the elongation of the spring converges to 0. We designed a switched controller for the system assuming under the assumption that the system is fully observable. The property verified is a safety property, i.e., the elongation of the spring is less than a given safety threshold for bounded time (30 discrete steps).

**Smart Thermostat:** We consider a variant of the regular thermostat hybrid system with on and off modes. The thermostat model considered is similar to [13] as it can sense the temperatures in the surrounding rooms and also the temperature of weather. The thermostat has 5 modes of operation namely full-cold, medium-cold, off, medium-heat, and full-heat. The decisions to switch among these modes is made based on the sensor readings from the surrounding rooms and environment. We consider the behavior where the weather temperature rises steadily and then steadily falls. The property verified is that the temperature of all the rooms stays in the optimum temperature range for which the thermostat is designed.

**Nonlinear Kinematic Controller:** We consider a variant of the kinematic system. In this example, we consider the controller to be a gear system which gives the acceleration as output to the environment. The acceleration provided is computed as a nonlinear function of the velocity $v$. The goal of this controller is to attain the desired speed of the system (say $v_d$) within a given time bound. The hybrid automata model of the closed loop system with immediate feedback would give nonlinear ODEs. As Z3 can handle nonlinear real arithmetic, we could perform bounded time safety verification of this system.

### C. Discussion

**Unbounded Time Verification:** Although in this paper we presented bounded time verification of closed loop systems, one can using abstract interpretation and loop invariants for verifying the transformed program for unbounded time. Given a program with a loop (bounded or unbounded), Interproc automatically applies the widening

\[ \begin{array}{cccc|cccc}
\text{Benchmark} & \text{Steps} & \text{Z3} & \text{Interproc} & \text{SpaceEx} \\
& & \text{box} & \text{oct} & \text{poly} & \text{box} & \text{oct} & \text{uni32} \\
\hline
\text{ACC}_1 & 25 & P, 25.8 \text{ s} & F, 0.2 \text{ s} & F, 12.2 \text{ s} & P, 18m 50s & F, 0.3 \text{ s} & F, 10.3 \text{ s} & F, 32.8 \text{ s} \\
\text{ACC}_2 & 25 & P, 25.9 \text{ s} & F, 0.2 \text{ s} & F, 12.1 \text{ s} & P, 18m 22s & F, 0.3 \text{ s} & F, 10.3 \text{ s} & F, 32.6 \text{ s} \\
\text{Kinematic}_1 & 25 & P, 5.8 \text{ s} & F, 0.05 \text{ s} & F, 1.8 \text{ s} & P, 4m 18s & F, 0.2 \text{ s} & F, 2.5 \text{ s} & F, 25.9 \text{ s} \\
\text{Kinematic}_2 & 25 & P, 5.8 \text{ s} & P, 0.05 \text{ s} & F, 1.8 \text{ s} & P, 4m 20s & F, 0.2 \text{ s} & P, 2.4 \text{ s} & F, 25.8 \text{ s} \\
\end{array} \]

**TABLE II:** Table showing the running times of safety verification problem using Z3, Interproc (box, octagon, and polyhedral domain), and SpaceEx (box, oct, and uni32 support functions). The table shows whether the tool could prove (P) or failed to prove (F) the property and the time taken.

**B. Experimental Results With BMC Using Z3**

As observed in Table II, considering the trade off between precision and scalability, we evaluate a suite of benchmarks using only Z3. The results of verifying the following benchmarks using Z3 are given in Table III.

**Motor Transmission Shift Controller in Electric Vehicles:** This benchmark, proposed in [4], deals with gear transmission in electric vehicles. This system models the phenomenon where the shaft disengages from one gear and meshes with another. The dynamics of the system during the meshing process is given by the following dynamics.

\[
\begin{align*}
\dot{p}_x &= v_x \\
\dot{v}_x &= F_s/m_s \\
\dot{p}_y &= v_y \\
\dot{v}_y &= R_s \cdot (T_m - T_f)/J
\end{align*}
\]

Where $p_x, p_y, v_x, v_y$ denote the $x$ and $y$ position and velocity of the shaft respectively, $m_s$ is the mass of the shaft, $F_s$ is the force applied during the transmission, i.e., given as an input, $R_s$ is the radius of the shaft, $T_m$ is the motor torque given as an input during the transmission, $T_f$ is the resistive torque, and $J$ denoting the rotational inertia of the gear.

A controller designed for the above system that actuates $F_s$ and $T_m$ is required to smoothen the meshing process during the transmission. In [4], the authors synthesize a piecewise affine, open loop controller of the system. We use a fragment of the open loop controller and verify the property that in bounded time (say within 25 discrete steps), the meshing process is successful.

**Locomotive Spring Damper:** This system obtained from [24] models the spring damping phenomenon when a locomotive pulls the train car. The dynamics of the system are given as:

\[
\begin{align*}
\dot{d}_s &= v_l - v_c \\
\dot{v}_l &= -\frac{k}{m_l}d_s - \frac{b + c}{m_l}v_l + \frac{b}{m_l}v_c + \frac{F}{m_l} \\
\dot{v}_c &= \frac{k}{m_c}d_s + \frac{b}{m_c}v_l - \frac{b}{m_c}v_c
\end{align*}
\]

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Steps</th>
<th>Verif. Res.</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTSC</td>
<td>15</td>
<td>Meshed</td>
<td>12.6 s</td>
</tr>
<tr>
<td>MTSC</td>
<td>20</td>
<td>Meshed</td>
<td>1m 14 s</td>
</tr>
<tr>
<td>MTSC</td>
<td>25</td>
<td>Meshed</td>
<td>5m 55 s</td>
</tr>
<tr>
<td>Locomotive</td>
<td>30</td>
<td>Safe</td>
<td>42.4 s</td>
</tr>
<tr>
<td>Thermostat</td>
<td>35</td>
<td>Optimum</td>
<td>6.9 s</td>
</tr>
<tr>
<td>Thermostat</td>
<td>40</td>
<td>Optimum</td>
<td>15.1 s</td>
</tr>
<tr>
<td>Thermostat</td>
<td>45</td>
<td>Optimum</td>
<td>33.4 s</td>
</tr>
<tr>
<td>Non.Lin.Kin.</td>
<td>20</td>
<td>Safe</td>
<td>2m 25 s</td>
</tr>
</tbody>
</table>

VI. Related Work

Hybrid Systems is the formalism used to model systems that interact with physical environment that are guided by programs. In this formalism, the software state is abstracted as a state transition graph and the continuous interaction with the environment is represented as ODEs. Reachable set computation for discrete models, linear and nonlinear systems [16], [15], [5], bisimulation techniques [29], and abstraction techniques [31] are commonly used techniques for attacking the verification problem. Discretization of linear systems for computing relational abstractions was considered in [33], however, it does not consider periodic inputs from control software.

Abstract interpretation [7] has been successfully applied in static analysis for embedded software to analyze arithmetic overflows, division by zero errors, and buffer overflows. Error analysis of embedded software with computations over float and double have been analyzed in Fluctuat [18]. Similar analysis that uses ellipsoidal domain for stability of digital filters is performed in [14]. In the current paper, the transformed program not only considers the behavior of the software, but also of the environment and hence prove properties of the closed loop system.

In [23], the authors consider a similar system, i.e. an environment and controller program that governs it. They perform symbolic error analysis, which deals with the numerical errors due to the controller implementations. These errors and then used together with the stability proofs of the environment using Lyapunov functions to obtain a region of stability for the control system implementation. Bounded time executions of the same system are considered in [22], however the authors focus on symbolically perform the robustness analysis for getting an overapproximation of the reachable states. This paper also considers a similar setting, however instead of separating the error analysis of software and the stability proofs for environment, we instead generate a transformed program that captures the behavior of the closed loop system.

HybridFluctuat [1], Sahvy [28], and Frehse et. al. [17] consider safety verification of systems where environment is periodically actuated at discrete intervals by a program. While HybridFluctuat uses abstract interpretation to compute the reachable set of software state and uses guaranteed simulation engine GRKLib [2] for continuous trajectories of the environment, Sahvy uses SMT Solver Z3 to generate assertions over the software state and uses Flow* [5] for obtaining the reachable set of the environment. The authors in [17] use SpaceEx for the analysis. The difference between these techniques, and those outlined here, are that we discretize the continuous flow of the environment and construct an equivalent program to be analyzed. One of the advantages of our technique is that one could deduce inductive invariants or fixed point using the abstract interpretation engine to verify the safety of the system for unbounded time, which cannot be achieved in these other approaches.

In [27], the authors consider the setting of environment being actuated periodically. They construct Approximate Quotient Transition System from the model to compute reachable set of states and generate counterexamples for violation of safety property. In [12], the authors perform systematic simulations of the closed loop system and search the state space for counterexamples that violate the
safety property. Although the technique generates sample simulations, the technique is unsound and can only be used for bug-detection.

VII. CONCLUSIONS AND FUTURE WORK

In this paper, we presented a technique for verifying closed loop systems where the environment is governed by linear ODEs and a control program periodically senses the state of environment and sets the actuation parameters. Such a model represents the deployed embedded system more accurately and takes into account the worst case execution time and task scheduling in real time operating systems. We reduce the bounded time safety verification of closed loop system to software verification by presenting a transformation procedure. We use two main techniques namely abstract interpretation and bounded model checking using SMT for verifying correctness of programs after transformation, compare the performance with reachability computation tools and present relative merits and demerits of these approaches.

We believe that the transformation presented in this paper is an important contribution because now one is not just restricted to hybrid systems verification tools, but can also apply software verification techniques for analyzing closed loop systems. This also leads to new avenues for future work such as applying symbolic execution [3], software testing techniques [26], learning loop invariants, etc. for verifying the transformed program. Investigating liveness properties of closed loop systems using program termination techniques [6] and its relation with “closed loop stability” can also be investigated.

Acknowledgements: The authors would like to thank Sayan Mitra for many helpful discussions and feedback on early drafts of this paper. This research was supported by NSF CCF 1422798 grant.

REFERENCES


