C2E2: A Verification Tool For Stateflow Models

Parasara Sridhar Duggirala[1], Sayan Mitra[2], Mahesh Viswanathan[1], and Matthew Potok[2]

[1] Department of Computer Science, University of Illinois at Urbana Champaign,
`duggira3@illinois.edu`, `vmahesh@illinois.edu`

[2] Department of Electrical and Computer Engineering, University of Illinois at Urbana Champaign,
`mitras@illinois.edu`

Abstract. Mathwork's Stateflow is a predominant environment for modeling embedded and cyberphysical systems where control software interact with physical processes. We present Compare-Execute-Check-Engine (C2E2)—a verification tool for continuous and hybrid Stateflow models. It checks bounded time invariant properties of models with nonlinear dynamics, and discrete transitions with guards and resets. C2E2 transforms the model, computing simulations using a validated numerical solver, and then computes reachtube over-approximations with increasing precision. For this last step it uses annotations that have to be added to the model. These annotations are extensions of proof certificates studied in Control Theory and can be automatically obtained for linear dynamics. The C2E2 algorithm is sound and it is guaranteed to terminate if the system is robustly safe (or unsafe) with respect to perturbations to the of guards and invariants of the model. We present the architecture of C2E2, its workflow, and examples illustrating its potential role in model-based design, verification, and validation.

1 Introduction

Cyberphysical systems (CPS) are systems that involve the close interaction between a software controller and a physical plant. The state of the physical plant evolves continuously with time and is often modeled using ordinary differential equations (ODE). The software controller, on the other hand, evolves through discrete steps and these steps influence the evolution of the physical process. This results in a "hybrid" behavior of discrete and continuous steps that makes the formal analysis of these models particularly challenging, so much so, that even models that are mathematically extremely simple are computationally intractable. In addition, many physical plants have complicated continuous dynamics that are described by nonlinear differential equations. Such plants, even without any interaction with a controlling software, are often unamenable to automated analysis.

On the other hand, the widespread deployment of cyberphysical systems in safety critical scenarios like automotives, avionics, and medical devices, have made formal, automated analysis of such systems necessary. This is evident from the extensive activity in the research community [19,18,7]. Given the challenges of formally verifying cyberphysical systems, the sole analysis technique that is commonly used to analyze nonlinear systems is numerical simulation. However, given the large, uncountable space of behaviors, using numerical simulations to discover design flaws is like searching for a needle in the proverbial haystack. In this paper we present a tool C2E2 (Compare-Execute-Check-Engine) that leverages the power of numerical simulations to formally prove or disprove the safety of cyberphysical systems over a bounded time interval.

Systems analyzed by C2E2 are described using Stateflow diagrams. Mathwork's Stateflow is the predominant, even de facto standard, environment for designing and developing embedded and cyberphysical systems both in the industry and in academia. C2E2 interprets Stateflow designs as hybrid automata [13], which is a popular mathematical model, with precise semantics, for describing cyberphysical systems. The models given as input to C2E2 must be *annotated*. The annotations here are similar in spirit to code assertions and contracts used in the software verification domain. Each mode in the Stateflow diagram has to be annotated with what we call a *discrepancy function* by the user. Discrepancy functions are a generalization of several proof certificates used in control theory for analyzing convergence and divergence of trajectories. In[8] we define discrepancy functions and discuss how they can be computed automatically for a reasonably expressive class of models. C2E2 transforms the input model and compiles it with a numerical simulation library to produce a validated simulator for the model. This simulator is then used for computing increasingly precise reach set over-approximations until it proves or disproves the bounded time safety property.

Our simulation based verification approach underlying C2E2, was first presented in [8] and was subsequently used for a significant case study in [9]. The current paper outlines several enhancements to the C2E2 tool that have been made since then. First the verification algorithm presented in [8] only worked for *switched systems* [3] where the time of mode switches is explicitly given in the input. In this paper, we extend the algorithm to analyze "full hybrid automata", i.e., continuous variables can be reset on mode switches, and mode switches take place based on enabling guards rather than explicitly given times. These theoretical improvements enable us to use C2E2 on a new set of examples. We report our experience in using C2E2 on these examples and its performance as experimental results. Next, C2E2 has been engineered to be more robust, moving from an in-house prototype, to something that can be used by the wider academic community. Finally, the tool now has a few additional features that make for a better user experience. First, it has been integrated with Stateflow, to make it useful for a wider community. Second, it can be used through a graphical user interface. And lastly, visualization tools have been added to enable users to plot various aspects of the reachable state space.

[3] Switched system here refers to a design where the state of the physical plant does not change when a discrete transition is taken.

1.1 Related Work

Tools for verifying CPS vary widely based on the complexity of the complexity of the continuous dynamics. Uppaal [14], HyTech [12], and SpaceEx [11] are tools verifying timed automata, rectangular hybrid automata and linear hybrid automata respectively. Current tools available for verifying nonlinear dynamics are d/dt [2], Flow* [5] and Ariadne [3]. Typically these tool use symbolic models for computing reachable set of states (or their overapproximations) from a given initial set for inferring safety of the system. Such tools provide formal soundness guarantees, however, do not typically provide relative completeness. Further, these tools do not analyze Stateflow models and hence a user has to specify model in the a specific input language, which requires additional learning curve for using these tools.

Given the popularity of Simulink-Stateflow framework to model CPS, there are several MATLAB based tools which verify such models. Breach [7] uses sensitivity analysis for analyzing MTL properties of systems using simulations. This analysis is sound and relatively complete for linear systems, but does not provide formal guarantees for nonlinear systems. S-Taliro [18] is a falsification engine that search for counterexamples using Monte-Carlo techniques and hence provides only probabilistic guarantees. STRONG [6] uses robustness analysis for coverage of all executions from the initial set, using Lyapunov functions, however cannot handle nonlinear systems. C2E2, although requires additional annotations, can handle nonlinear systems specified in Stateflow and provide rigorous soundness and completeness guarantees. The simulation based verification algorithm in [8] has been extended for more general properties in [9] and to networked input output systems in [10].

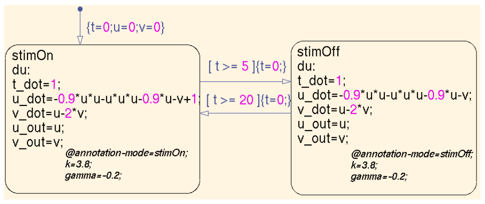
2 Hybrid Models and Safety Verification

Hybrid automata is a convenient and widely used mathematical framework for modeling cyberphysical systems. One of its key features is that it combines two distinct modeling styles, namely, differential equations and automata. For CPS, this enables the physical environment to be modeled by differential equations and software and communication to modeled by discrete state transitions.

Figure 1(a) shows an example of a simplified hybrid model of cardiac cell with a pacemaker created using Mathworks^{TM} Stateflow^{TM}. The pacemaker has two modes or *locations* of operation: in the *stimOn* mode the cell is electrically stimulated by the pacemaker, and in *stimOff* the stimulus is absent. The continuous variables u and v model certain electrical properties of the cell. The stimulus is applied to the cell at regular time intervals as measured by the clock t. The resulting evolution of one of the continuous variables over time is shown in Figure 1(b).

Although there is no published formal semantics of Stateflow^{TM} models, it is standard to consider them as hybrid automata [16,4,7]. Let us denote the set of all the variables (both continuous and discrete) in the model as the set V.

This set includes a special variable *loc* to denote the current location. In this case, *loc* can be either *stimOff* or *stimOn*. The rest of the variables in V are continuous and real-valued. The set of all possible valuations of all variables in V is denoted as $val(V)$—this defines the set of states of the model. The continuous evolution of the variables is modeled by *trajectories*. A single trajectory τ is a function $\tau : [0, t] \rightarrow val(V)$, where $t \geq 0$ is the duration τ. The state of the system at a given time t in τ is $\tau(t)$, and the value of a particular variable $v \in V$ at that state is denoted by $\tau(t).v$. A set of trajectories is specified by differential equations involving the continuous variables (see, for example, Figure 1). A trajectory τ satisfies an ordinary differential equation (ODE) $\dot{v} = f(v)$ if at each time t in the domain of the trajectory, $\frac{d(\tau(t).v)}{dt} = f(\tau(t).v)$. When f is a nice[4] function, the ODE has a unique solution for a given initial state and a duration. With different initial states and time bounds, an ODE defines a set of trajectories.



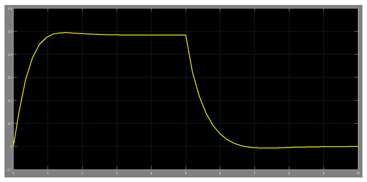
```
{t=0;u=0;v=0}
```

```
stimOn
du:
t_dot=1;
u_dot=-0.9*u*u-u*u*u-0.9*u-v+1;
v_dot=u-2*v;
u_out=u;
v_out=v;
            @annotation-mode=stimOn;
            k=3.8;
            gamma=-0.2;
```

```
[ t >= 5 ]{t=0;}
[ t >= 20 ]{t=0;}
```

```
stimOff
du:
t_dot=1;
u_dot=-0.9*u*u-u*u*u-0.9*u-v;
v_dot=u-2*v;
u_out=u;
v_out=v;
            @annotation-mode=stimOff;
            k=3.8;
            gamma=-0.2;
```

(a) Model.



(b) Behavior.

Fig. 1: (a) Simplified Stateflow$^{\text{TM}}$ model of a cardiac cell and a pacemaker. (b) A simulation of the model from an initial state.

The discrete transitions between the two locations are specified by a set A of actions (see Figure 1). An action $a \in A$ is enabled at a state whenever the state satisfies a special predicate $Guard_a$ that is associates with the action. The discrete transition from the location *stimOn* to *stimOff* is enabled only when $t >= 5$, that is the clock has counted 5 units in *stimOn*. When the system takes a discrete transition, the new state of the system after the transition is defined by a function $Reset_a$ that maps the old state to a new state (and possibly a new location). For example, the reset function for *stimOn* to *stimOff* sets $t = 0$, $loc = stimOff$, and leaves the other continuous variables u and v unchanged. All of these components together defines the behavior of the hybrid automaton in terms of a sequence of alternating trajectories and transitions.

Definition 1. *A Hybrid Automata (HA)* $\mathcal{A}$ *is a tuple* $\langle V, Loc, A, \mathcal{D}, \mathcal{T} \rangle$ *where*

[4] For example, Lipschitz continuous or smooth. A continuous function $f : \mathbb{R}^n \times \mathbb{R} \rightarrow \mathbb{R}$ is *smooth* if all its higher derivatives and partial derivatives exist and are also continuous. It has a Lipschitz constant $K \geq 0$ if for every $x_1, x_2 \in \mathbb{R}^n$, $||f(x_1) - f(x_2)|| \leq K||x_1 - x_2||$.

(a) *$V = X \cup \{loc\}$ is a set of variables. Here loc is a discrete variable of finite type Loc. Valuations of loc are called* locations. *Each $x \in X$ is a continuous variable of type $\mathbb{R}$. Elements of $val(V)$ are called* states.

(b) *A is a finite set of* actions *or transition labels.*

(c) *$\mathcal{D} \subseteq val(V) \times A \times val(V)$ is the set of* discrete transitions. *A transition $(\mathbf{v}, a, \mathbf{v}') \in \mathcal{D}$ is written as $\mathbf{v} \overset{a}{\rightarrow} \mathbf{v}'$. The discrete transitions are specified by finitely many guards and reset maps involving V.*

(d) *$\mathcal{T}$ is a set of trajectories for X which is closed under suffix, prefix and concatenation (see [13] for details). For each $l \in Loc$, a set of trajectories $\mathcal{T}_l$ for location l are specified by differential equations E_l and an invariant $I_l \subseteq val(X)$. Over any trajectory $\tau \in \mathcal{T}_l$, loc remains constant and the variables in X evolve according to E_l such that for all at each time in the domain of τ, $\tau(t)$ satisfies the invariant I_l.*

An execution of a hybrid automaton $\mathcal{A}$ records all the information (about variables) over a particular run. Formally, an *execution* is an alternating sequence of trajectories and actions $\sigma = \tau_0 a_1 \tau_1 \ldots$ where each τ_i is a closed trajectory and $\tau_i(t) \overset{a_{i+1}}{\rightarrow} \tau_{i+1}(0)$, where t is the last time point in τ_i. The *duration* of an execution is the total duration of all the trajectories. The set of all executions is denoted as $execs(\mathcal{A})$. In this paper, we only consider executions with bounded number of switches and with bounded duration. Given a set of initial states $\Theta \subseteq val(V)$, the set of *executions from Θ* are those executions in $execs(\mathcal{A})$ with their first state, $\tau_0(0)$, in Θ. The set of executions starting from Θ of duration at most T and with at most N transitions will be denoted as $exec(\mathcal{A}, \Theta, T, N)$.

Definition 2 (Safe and Unsafe). *Given a hybrid automata $\mathcal{A}$ with an initial set Θ, unsafe set U, time bound T, and transition bound N, it is said to be unsafe, if there exists an execution $\tau_0 a_1 \ldots \tau_k \in exec(\mathcal{A}, \Theta, T, N)$ such that $\tau_k(t) \in U$, where t is the last time point in τ_k. Otherwise, $\mathcal{A}$ is said to be* safe.

Our algorithm for safety verification as well as its analysis rely heavily on the notion of distance between continuous states and trajectories of the automaton. To state our results formally, we need to introduce a we notations first. For a vector $x \in \mathbb{R}^n$, $||x||$ denotes the ℓ^2 norm. For $x_1, x_2 \in \mathbb{R}^n$, $||x_1 - x_2||$ is the Euclidean distance between the points. For $\delta > 0$, $B_\delta(x_1) \subseteq \mathbb{R}^n$ denotes the set of points that is a closed ball of radius δ centered at x_1. For a set $S \subseteq \mathbb{R}^n$, $B_\delta(S) = \cup_{x \in S} B_\delta(x)$. $B_\delta(S)$ expands S by δ. We will find it convenient to also define the notion of shrinking S by δ: For $\delta < 0$, and $S \subseteq \mathbb{R}^n$, $B_\delta(S) = \{x \in S \mid B_{-\delta}(x) \subseteq S\}$. For a bounded set S, a *δ-cover* of S is a finite collection of balls $\mathcal{X} = \{B_\delta(x_i)\}_{i=1}^m$ such that $S \subseteq \bigcup_{i=1}^m B_\delta(x_i)$. Its diameter $dia(S) \triangleq \sup_{x_1, x_2 \in S} ||x_1 - x_2||$.

Definition 3 (Perturbing a Hybrid Automata). *Given a hybrid automata $\mathcal{A} = \langle V, Loc, A, \mathcal{D}, \mathcal{T} \rangle$, we define an ϵ-*perturbation *of $\mathcal{A}$ as a new automaton $\mathcal{A}_\epsilon$ that has identical components as $\mathcal{A}$, except, (a) for each location $\ell \in Loc$, $I_\ell^{\mathcal{A}_\epsilon} = B_\epsilon(I_\ell^{\mathcal{A}})$ and (b) for each action $a \in A$, $Guard_a^{\mathcal{A}_\epsilon} = B_\epsilon(Guard_a^{\mathcal{A}})$.*

Here $I_{loc}^{\mathcal{A}}$ is the invariant of the location of a hybrid automata $\mathcal{A}$ and $Guard_a^{\mathcal{A}}$ denotes the guard set for action a. The definition permits $\epsilon < 0$ for perturbation of a hybrid automaton. Informally, a positive perturbation of a hybrid automata $\mathcal{A}$ bloats the invariants and guard sets and therefore enlarges the set of executions. A negative perturbation on the other hand, shrinks the invariants and the guards and therefore reduces the set of executions.

Definition 4 (Robust safety and unsafety). *Given a hybrid automata $\mathcal{A}$ with an initial set Θ, unsafe set U, time bound T, and bound on discrete transitions N, it is said to be* robustly safe *if and only if $\exists\epsilon > 0$, such that $\mathcal{A}_\epsilon$, with initial set Θ_ϵ, unsafe set U_ϵ, time bound T, and transition bound N is safe. It is said to be* robustly unsafe *if and only if $\exists\epsilon < 0$ such that $\mathcal{A}_\epsilon$, with initial set Θ_ϵ, unsafe set U_ϵ, time bound T, and transition bound N, is unsafe.*

For safety verification C2E2 expects the users to provide annotations for the ODEs defining the trajectories of the hybrid automaton in question. These annotations are called discrepancy function. For a differential equation $\dot{x} = f(x,t)$, the general definition of discrepancy function is given in [8]. In this paper, we consider a special form of discrepancy function given in Definition 5.

Definition 5. *Given a differential equation $\dot{x} = f(x)$, the tuple $\langle K, \gamma\rangle$ is called an exponential discrepancy function for the dynamics, if and only if for any two trajectories τ_1, τ_2 satisfying the differential equation, it holds that:*

$$||\tau_1(t) - \tau_2(t)|| \leq K||\tau_1(0) - \tau_2(0)||e^{\gamma t} \tag{1}$$

We call K as the multiplicity factor and γ as the exponential factor of the annotation.

Along with the input models, we expect the user to specify the discrepancy function by providing values for $\langle K, \gamma\rangle$. Also, if the differential equation is Lipschitz continuous with constant L, then $\langle K, \gamma\rangle = \langle 1, L\rangle$ would be a valid annotation. In Figure 1, the annotations for both the locations are provided as $\langle K, \gamma\rangle = \langle 3.8, -0.2\rangle$.

3 Verifying Hybrid Systems From Simulations

We give an overview of the verification algorithm implemented in C2E2. In brief, the algorithm generates a cover of the initial set and then performs four steps repeatedly until it reaches a safe/unsafe decision for each cover or its refinement. In making this decision, it generates a simulation from the cover and then bloats this simulation by a factor computed from the given annotation to compute an over-approximation of the reachable states. If the over-approximation, decides safe/unsafe then it moves on to another cover, otherwise, it refines the cover. These operations are performed by several subroutines which we describe next.

3.1 Building Blocks

The first building block for the algorithm is a subroutine called $valSim$ that generates validated simulations for the individual dynamical systems or locations of the hybrid automaton. Given a trajectory τ starting from a given state, $valSim$ subroutine computes overapproximation of τ with specific error.

Definition 6 (Validated Simulation). *Given an error bound $\epsilon > 0$, a time step $h > 0$, a time bound $T = (k+1)h$, and an initial state x_0, an (x_0, ϵ, h, T)*-simulation *of the differential equation $\dot{x} = f(x)$ is a sequence set of continuous states $\rho = R_0, R_1, \ldots, R_k$ such that (a) for any i in the sequence $dia(R_i) \leq \epsilon$, and (b) for any time t in the interval $[ih, (i+1)h]$, the solution from x_0 at time t is in R_i, i.e., $\tau(t) \in R_i$.*

The subroutine $valSim(x_0, h, T, f)$ returns a tuple $\langle \rho, \epsilon \rangle$ such that ρ is an (x_0, ϵ, h, T)-simulation. In C2E2, validated simulation engines such as VNODE-LP [17] and CAPD [1] are used for implementing $valSim$. For the completeness of our algorithm, we require that the error ϵ can be made arbitrarily small by decreasing h. In systems with finite precision arithmetic, simulation engines produce accurate simulations up to the order of 10^{-7}, and also there are libraries supporting arbitrary precision integration for some differential equations.

Next, the $computeReachTube$ subroutine: it uses simulations to compute overapproximations of a set of trajectories from a set of initial states.

Definition 7 (Overapproximate Reach Tube). *For a set of initial states S, error bound $\epsilon > 0$, time step $h > 0$, and $T = (k+1)h$, an (S, ϵ, h, T)*-reachtube *of the differential equation $\dot{x} = f(x)$ is a sequence $\psi = R_0, R_1, \ldots, R_k$ such that (a) for any i in the sequence $dia(R_i) \leq \epsilon$, (b) for any trajectory starting τ from S and for each time $t \in [ih, (i+1)h]$, $\tau(t) \in R_i$.*

If S is large and ϵ is small then the strict inclusion may preclude a (S, ϵ, h, T)-reachtube from existing. To compute reachtubes from a compact set S centered at x_0, $computeReachTube$ performs the following three steps:

1. $\langle \rho, \epsilon_1 \rangle \leftarrow valSim(x_0, h, T, f)$, let $\rho = R_0, \ldots, R_k$.
2. $\epsilon_2 = \sup\{K||x_1 - x_2||e^{\gamma t} \mid x_1, x_2 \in S, t \in [0, T]\}$, where $\langle K, \gamma \rangle$ is the annotation for the dynamics given by f.
3. $\psi = B_{\epsilon_2}(R_1), \ldots, B_{\epsilon_2}(R_k)$.

From Definition 5 it follows that ψ is a $(S, \epsilon_1 + \epsilon_2, h, T)$-reachtube. It can be shown [8] that $\epsilon \to 0$ as $\delta \to 0$ and $h \to 0$. In summary, the subroutine call $computeReachTube(S, h, T, f)$ returns $\langle \psi, \epsilon \rangle$ using the above steps. To provide formal guarantees from reachtube over-approximations, we need to distinguish when a given reachtube *must* satisfy a predicate P from when it *may* satisfy it. The next subroutine $tagRegion$ marks or tags each region in a reachtube with respect to a given predicate.

Definition 8 (Tagging). *Given two sets $R, P \subseteq \mathbb{R}^n$ the subroutine $tagRegion(R, P)$ returns $must$, may or $\perp$ such that, (a) if $R \subseteq P$, then return $must$, (b) if $R \cap P \neq \emptyset$ and $R \nsubseteq P$ then return may, and (c) otherwise ($R \cap P = \emptyset$), return $tag = \perp$.*

The above subroutines are used for over-approximating reach sets and deciding safety for individual differential equations of individual locations of a hybrid automaton. In order to reason about, invariants, guards and resets which are essential for capturing the hybrid behavior of mode switches, we have to (a) detect the reachable states that satisfy the respective location invariants and (b) identify the states from which location switches or transitions can occur. The next subroutine $\mathit{invariantPrefix}(\psi, S)$ takes a reachtube and a set S and returns the longest contiguous prefix of ψ that intersects with S. This is later used in solving problem (a).

Definition 9 (Invariant Prefix). *Given a reachtube $\psi = R_0, \ldots, R_k$ and a set S, $\mathit{invariantPrefix}(\psi, S)$ returns the longest sequence $\phi = \langle R_0, \mathit{tag}_0\rangle \ldots \langle R_m, \mathit{tag}_m\rangle$, such that $\forall 0 \leq i \leq m$, $\mathit{tag}_i = \mathit{must}$ when $\forall j \leq i, \mathit{tagRegion}(R_j, S) = \mathit{must}$ and $\mathit{tag}_i = \mathit{may}$ when $\forall j \leq i, \mathit{tagRegion}(R_j, S) \neq \perp$ and $\exists k \leq i, \mathit{tagRegion}(R_k, S) = \mathit{may}$. That is, if $m < k$, then $\mathit{tagRegion}(R_{m+1}, S) = \perp$.*

Intuitively, a region R_i is tagged must in an invariant prefix, if all the regions before R_i (including itself) are contained within the set S. It is tagged may if all the regions before it have nonempty intersection with S, and at least one of them (including itself) is not contained within the set S. Given a reachtube ψ from a set Q, $\psi = \mathit{invariantPrefix}(\psi, \mathit{Inv}_{loc})$ returns the over-approximation of the valid set of trajectories from Q that respect the invariant of the location loc of the hybrid automata. If a region R_i in ϕ is tagged must, then there exists at least one trajectory from Q that can reach R_i. Also, the set of all reachable states from Q that satisfy the invariant are contained in ϕ. Subroutine $\mathit{checkSafety}$ checks whether such an invariant prefix ϕ is safe with respect to an unsafe set U. It defined as:

$$\mathit{checkSafety}(\phi, U) = \begin{cases} \mathit{safe}, \text{ if } \forall \langle R, \cdot\rangle \in \phi, \perp = \mathit{tagRegion}(R, U) \\ \mathit{unsafe}, \text{ if } \exists \langle R, \mathit{must}\rangle \in \phi, \mathit{must} = \mathit{tagRegion}(R, U) \\ \mathit{unknown}, \text{ otherwise} \end{cases}$$

Notice that ϕ is inferred to be safe only when all the regions in the invariant prefix are tagged $\perp$ with respect to U (i.e. empty intersection). It is unsafe only when there is a must region in the invariant prefix is contained within U. This is the core of the soundness argument for the verification algorithm, i.e., if $\mathit{checkSafety}(\phi, U)$ returns safe or unsafe, then indeed that is the correct answer for the set of initial states covered by ϕ.

The final subroutine $\mathit{nextRegions}$ computes over-approximation of the reachable states that serve as the initial states in a new location after a transition to it. A discrete transition a is enabled at the instance trajectory τ satisfies the guard condition Guard_a and the state after the discrete transition is obtained by applying the reset map Reset_a. Given a sequence of tagged regions, the subroutine $\mathit{nextRegions}$ returns the tagged set of reachable regions after a discrete transition.

Definition 10 (Next Regions). *Given $\phi = \langle R_0, \mathit{tag}_0\rangle, \ldots, \langle R_m, \mathit{tag}_m\rangle$, a sequence of tagged regions, the subroutine $\mathit{nextRegions}(\phi)$ returns a set of tagged regions $\mathbf{R}$.*

$\langle R', tag' \rangle \in \mathbf{R}$ if and only if there exists an action a of the automaton and a region R_i in ϕ such that $R' = Reset_a(R_i)$ and one of the following conditions hold:

(a) $R_i \subseteq Guard_a, tag_i = must, tag' = must.$
(b) $R_i \cap Guard_a \neq \emptyset, R_i \nsubseteq Guard_a, tag_i = must, tag' = may.$
(c) $R_i \cap Guard_a \neq \emptyset, tag_i = may, tag' = may.$

A tagged region $\langle R', tag' \rangle \in \mathbf{R}$ is labeled $must$ only when the region R_i is a $must$ region and is contained within the $Guard_a$. In all other cases, the region is tagged may when $R_i \cap Guard_a \neq \emptyset$. This ensures that a regions tagged $must$ are indeed reachable after the discrete transition, and all the regions tagged may contain the reachable states after the discrete transition.

3.2 Verification Algorithm

The C2E2 algorithm for verification is given in Algorithm 1. Lines 4 - 19 implement the main loop that computes a cover the initial set, computes the over-approximation of reachtube, checks safety, and refines the cover if needed. The subroutine $taggedCover$ (line 3), first computes a cover of Θ with sets of δ diameter and tags them (with $tagRegion$) with respect to Θ. The resulting tagged sets are collected in $\mathcal{X}$. We add additional attributes to each tagged region: $Time$ tracks the time of the trajectory leading to a region, Loc tracks its current location, and $Switches$ tracks the number of discrete transitions taken. Although not explicitly mentioned in the algorithm, we update the tags for regions in the reachtube based on the time taken by trajectories and the discrete transitions encountered during verification. The algorithm checks whether all the executions of hybrid automata from the regions in the cover of the initial set are safe or unsafe. If it is safe, then the region is removed from the initial set Θ (line 15); if it is unsafe, then the algorithm returns unsafe (line 11); and if it is neither then the initial cover is refined (line 17).

For each of the regions in the tagged δ-cover of the initial set, the inner loop (lines 6 - 13) computes over-approximations of the reachable states. The subroutine $computeReachTube$ (line 7) computes an overapproximation of all the trajectories starting from the tagged region in the initial set cover. Subroutine $invariantPrefix$ (line 8) computes ϕ, an overapproximation of the valid set of trajectories which respect the invariant of the current location. Safety of ϕ is checked using the subroutine $checkSafety$ (line 9). The regions in ϕ where discrete transitions are enabled and the initial states for the trajectories in the next location are computed in subroutine $computeNext$ (line 10). This loop continues until either the time bound for verification or the bound for number of discrete transitions is satisfied.

The reachtubes computed in lines 7, 8 are an overapproximation of all the reachable states of hybrid automata $\mathcal{A}$, given in Lemma 1. The regions in the reachtubes are $must$ only when they satisfy the invariant of each location and completely contained within the guard set it follows that these regions are reachable by at least one execution of $\mathcal{A}$, given as Lemma 2. For the set of regions $\mathcal{R}$ tagged may, it follows that $\epsilon = max\{dia(R) | R \in \mathcal{R}\}$ bloating of the invariants and guard sets would ensure that these regions are also reachable, given in Lemma 3. As the reachtubes can be made arbitrarily precise by decreasing the time step and the initial partitioning, it follows that given any ϵ, the precision of reachtubes can be bounded by ϵ by using small vales for θ and h. Lemma 1 helps in proving soundness and Lemmas 2 and 3 help in proving relative completeness. This guarantees that if the system is robustly safe, then the algorithm terminates and returns safe, and if the system is robustly unsafe, then the algorithm terminates and returns unsafe, given in Theorem 1.

```
input  : $\mathcal{A}, \Theta, U, T, N$
output: System is $safe$ or $unsafe$
1  $S \leftarrow \Theta; h \leftarrow h_0; \delta \leftarrow \delta_0$;
2  while $S \neq \emptyset$ do
3      $\mathcal{X} \leftarrow taggedCover(\Theta, \delta)$ ;
4      for $elem \in \mathcal{X}$ do
5          $Q_C \leftarrow \{elem\}$;
6          for $e \in Q_C \wedge e.Time < T \wedge e.Switches < N$ do
7              $\langle \psi, \epsilon \rangle \leftarrow computeReachTube(e.R, e.Loc, h, T - e.Time)$ ;
8              $\phi \leftarrow invariantPrefix(\psi, Inv_{e.Loc})$ ;
9              $result \leftarrow checkSafety(\phi, U)$ ;
10             if $result == safe$ then $Q_C \leftarrow Q_C \cup nextRegions(\phi)$;
11             else if $result == unsafe \wedge elem.tag == must$ then return $unsafe$;
12             else break;
13         end
14         if $result == safe$ then
15             $S \leftarrow S \setminus elem.R$ ;
16         else
17             $\delta \leftarrow \delta/2; h \leftarrow h/2;$ ;
18         end
19     end
20 end
21 return $safe$;
```

Algorithm 1: Algorithm for safety verification of hybrid automata using simulations and annotations.

Lemma 1. *All the regions R tagged may or must in ϕ (line 7 of algorithm 1) contains the reachable set of states of hybrid automata $\mathcal{A}$ starting from Θ within T time and N discrete transitions.*

Lemma 2. *If a region R is tagged must in ϕ (line 7 of algorithm 1), then there exists at least one execution of $\mathcal{A}$ from the initial set Θ that reaches R within T time and N discrete transitions.*

Lemma 3. *Let $\mathcal{R}$, be the set of all regions tagged may in ϕ (line 7 of algorithm 1) and $\epsilon = max\{dia(R) | R \in \mathcal{R}\}$. Given any region $R \in \mathcal{R}$, there exists at least one execution of $\mathcal{A}_\epsilon$ from the initial set Θ_ϵ that reaches R within T time and N discrete transitions.*

Theorem 1 (Soundness and Relative Completeness). *Given initial set Θ, unsafe set U, time bound T, bound on discrete transitions N, and hybrid automata $\mathcal{A}$, if the algorithm 1 returns safe or unsafe, then the system $\mathcal{A}$ is safe or unsafe. The algorithm will always terminate whenever the system is either robustly safe or robustly unsafe.*

4 C2E2: Internals and User Experience

4.1 Architecture of C2E2

The architecture for C2E2 is shown in Figure 2. The front end parses the input models, connects to the verification engine, provides a property editor and a plotter. It is developed in Python and vastly extends the Hylink parser [16] for Stateflow models. The verification algorithm (Algorithm 1) is implemented in C++. The frontend parses the input model file (.mdl or .hyxml) into an intermediate format and generates the simulation code. The properties are obtained from the input file or from the user through the front end's GUI. The simulation code is compiled using a validated simulation engine provided by Computer Assisted Proofs in Dynamic Groups (CAPD) library [1]. This compiled code and the property are read by the C2E2 verification algorithm which also uses the GLPK libraries. The verification result and the computed reachable set are read by the frontend for display and visualization. This modular architecture allows us to extend the functionality of the tool to new types of models (such as Simulink, DAEs), different simulation engines (for example, Boost, VNODE-LP), and alternative checkers (such as Z3).

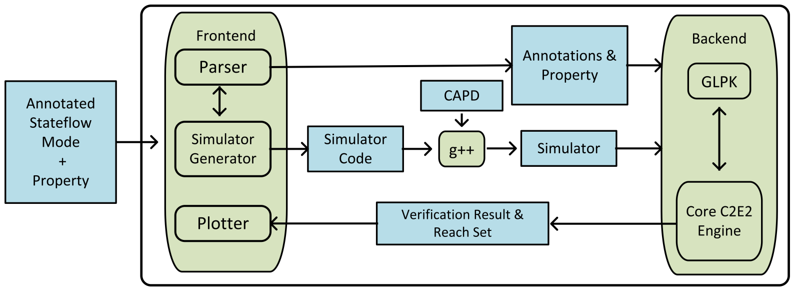


Fig. 2: Architecture of C2E2

4.2 Models, Properties, and Annotations

C2E2[5] takes as input annotated Stateflow models (such as in Figure 1(a)) with ODEs (possibly nonlinear) and discrete transitions defined by guards and resets. The guards have to be conjunctions of polynomial predicates over the state

[5] https://publish.illinois.edu/c2e2-tool/

variables and the reset maps have to be polynomial real-valued functions. The properties can be specified in the .hyxml model files or using the GUI. C2E2 can verify bounded time safety properties specified by a time bound, a polyhedral set of initial states and another set of unsafe states.

Annotations The user has to write annotations for each ODE in the model. This is done by specifying the multiplicity factor K and the exponential factor γ as comments in the Stateflow model, shown in Figure 1(a). For a broad range of nonlinear systems such annotations can be found. We illustrate with some examples.

Example 1. Consider a linear system $\dot{x} = Ax$, where all the eigenvalues of matrix A is nonzero. Let λ_m be the maximum among the real parts of all the eigenvalues of A. Then, for any two trajectories τ_1 and τ_2 of the linear system $||\tau_1(t) - \tau_2(t)|| \leq ||A||e^{\lambda_m t}||\tau_1(0) - \tau_2(0)||$ is an annotation. Here the matrix norm $||A||$ is defined as $\sup\{x^T Ax \mid ||x|| = 1\}$ and it can be computed using semidefinite programming. The input format would be $K = ||A||$ and $\gamma = \lambda_m$.

Example 2. Consider the differential equation in *stimOn* mode of the cardiac cell example: $\dot{u} = (0.1 - u)(u - 1)u - v$ and $\dot{v} = u - 2v$. By computing the maximum eigenvalues of the Jacobian matrix of the differential equation and the maximum norm of the contraction metric [15], we get that $||\tau_1(t) - \tau_2(t)|| \leq 3.8e^{-0.2t}||\tau_1.(0) - \tau_2.(0)||$ as an annotation. The input is specified as $K = 3.8$ and $\gamma = -0.2$.

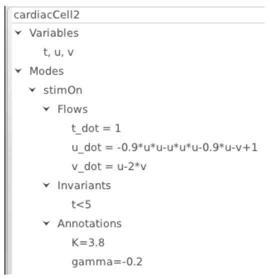
Example 3. Consider the differential equation $\dot{x} = 1+x^2y-2.5x; \dot{y} = 1.5x-x^2y-y$. By analyzing the auxiliary system (x_1, y_1) and (x_2, y_2), using the incremental stability, we have that

$$\frac{d}{dt}[(x_1 - x_2)^2 + 2(x_1 - x_2)(y_1 - y_2) + (y_1 - y_2)^2] = -2(x_1 - x_2 + y_1 - y_2)^2 < 0.$$

Therefore, for trajectories τ_1 and τ_2 of the differential equations it hence follows that $(\tau_1(t).x - \tau_2(t).x)^2 + 2(\tau_1(t).x - \tau_2(t).x)(\tau_1(t).y - \tau_2(t).y) + (\tau_1(t).y - \tau_2(t).y)^2 \leq (x_1 - x_2)^2 + 2(x_1 - x_2)(y_1 - y_2) + (y_1 - y_2)^2$ where $i = 1, 2, x_i = \tau_i.\mathsf{fstate}.x, y_i = \tau_i.\mathsf{fstate}.y$. The function $||\tau_1(t) - \tau_2(t)|| \leq 2||\tau_1(0) - \tau_2(0)||e^{0\times t}$ is an annotation and is specified as $K = 2, \gamma = 0$.

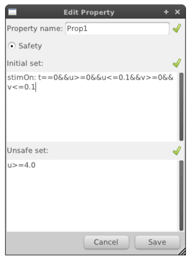
4.3 User Experience

In this section, we discuss the C2E2 interface for handling verification, properties and visualization. The users can add, edit, copy, delete or verify several properties. As each property is edited, the *smart parser* provides real-time feedback about syntax errors and unbounded initial sets (Figure 3(b)). Once properties are edited the verifier can be launched. Visual representation of reachable states and locations can aid debugging process. To this end, we have integrated a visualizer into C2E2 for plotting the projections of the reachable states. Once a property has been verified, the user can plot the valuations of variables against time or valuations of pairs of variables (phase plots). The unsafe set is projected on the set of plotting variables. The property parser and visualizer uses the Parma Polyhedra Library[6] and matplotlib[7]. Example plot for the cardiac cell is shown in Figure 3(c).



```
cardiacCell2
  Variables
     t, u, v
  Modes
     stimOn
        Flows
           t_dot = 1
           u_dot = -0.9*u*u-u*u*u-0.9*u-v+1
           v_dot = u-2*v
        Invariants
           t<5
        Annotations
           K=3.8
           gamma=-0.2
```

(a) Model.



Property name: Prop1

Safety

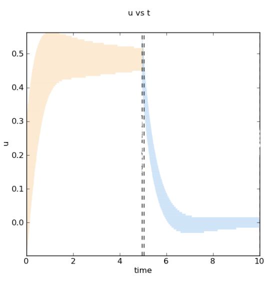
Initial set:

stimOn: t==0&&u>=0&&u<=0.1&&v>=0&&v<=0.1

Unsafe set:

u>=4.0

(b) Property.



(c) Reachable Set.

Fig. 3: Figure showing a snippet of cardiac cell model in (a), property dialog for specifying properties in (b), plot of reachable set for cardiac cell model in (c).

4.4 Stateflow Model Semantics

The annotated Stateflow models when given as input are interpreted as hybrid automaton as C2E2. Nondeterminism, which is allowed in hybrid automata framework is prohibited in Stateflow models. All the discrete transitions in Stateflow models are deterministic and are interpreted as "urgent" i.e. a transition is taken by the system *as soon as* it is enabled. In our fronted, we construct hybrid automaton for Stateflow models respecting the "urgent" semantics. Under such interpretation, the guard sets are only allowed to be hyperplanes. In general, the verification algorithm in Section 3 may not terminate for such guard conditions. We therefore use a heuristic and verify an ϵ perturbed model of the Stateflow model to ensure termination of verification algorithm.

4.5 Experiments

Simulation based verification approach for annotated models has been demonstrated to outperform other verification tool such as Flow* and Ariadne in [8]. In this paper, we present the verification results for some of the nonlinear and linear hybrid automata benchmarks in Table 1. The annotations for each of these benchmarks have been obtained by procedures given in Section 4.2. All the experiments have been performed on Intel i-7 Quad core processor with 8GB ram running Ubuntu 11.10.

[6] http://bugseng.com/products/ppl/
[7] http://matplotlib.org/

Benchmark	Vars.	Num. Loc.	TH	VT (sec)	Result
Cardiac Cell	3	2	15	17.74	safe
Cardiac Cell	3	2	15	1.91	unsafe
Nonlinear Navigation	4	4	2.0	124.10	safe
Nonlinear Navigation	4	4	2.0	4.94	unsafe
Inverted Pendulum	2	1	10	1.27	safe
Inverted Pendulum	2	1	10	1.32	unsafe
Navigation Benchmark	4	4	2.0	94.35	safe
Navigation Benchmark	4	4	2.0	4.74	unsafe

Table 1: Experimental Results for benchmark examples. Vars: Number of Variables, Num. Loc. : Number of discrete locations in hybrid automata, TH: Time Horizon for Verification, VT (sec) : Verification time for C2E2 in seconds, Result: Verification result of C2E2.

This early termination strategy for unsafe behavior of the system (Algorithm 1, line 11) when the system is unsafe is reflected in Table 1. On standard examples C2E2 can successfully verify these systems within the order of minutes and also handle nonlinear differential equations with trigonometric functions of inverted pendulum.

5 Conclusions and Future Work

C2E2 presented in this paper is a tool for verifying a broad class of hybrid and dynamical systems models. It uses validated simulations and model annotations to prove the most commonly encountered type of properties, namely bounded-time invariants. It can handle models created using the Stateflow environment that is the de facto standard in embedded control design and implementation. The improvements presented in this paper beyond the version of [8], include the complete support for hybrid models implemented in a new algorithm and the supporting theory, the new user interface for editing properties, and the reachtube plotting function. The tool is freely available for academic and research use from `https://publish.illinois.edu/c2e2-tool/`.

Our future plans include implementation of features to support temporal precedence properties [9] and compositional reasoning [10,4]. Another avenue of work leverages the "embarrassing parallelism" in the simulation-based approach. We anticipate that the C2E2's architecture and its open interfaces, for example, the .hyxml input format, text-based representation of reachtubes, will support research and eduction in embedded and hybrid systems community by helping explore new ideas in modeling, verification, synthesis, and and testing.

Acknowledgements: The authors were supported by the National Science Foundation research grant CSR 1016791.

References

1. Computer assisted proofs in dynamic groups (capd). http://capd.ii.uj.edu.pl/index.php.
2. E. Asarin, T. Dang, and O. Maler. The d/dt tool for verification of hybrid systems. In *Computer Aided Verification*, pages 365–370. Springer, 2002.
3. A. Balluchi, A. Casagrande, P. Collins, A. Ferrari, T. Villa, and A. Sangiovanni-Vincentelli. Ariadne: a framework for reachability analysis of hybrid automata. In *M.T.N.S.*, 2006.
4. A. Biere and R. Bloem, editors. *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*. Springer, 2014.
5. X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *Computer Aided Verification*, pages 258–263. Springer, 2013.
6. Y. Deng, A. Rajhans, and A. A. Julius. Strong: A trajectory-based verification toolbox for hybrid systems. In *QEST*, pages 165–168, 2013.
7. A. Donzé. Breach, a toolbox for verification and parameter synthesis of hybrid systems. In *C.A.V.* 2010.
8. P. S. Duggirala, S. Mitra, and M. Viswanathan. Verification of annotated models from executions. In *EMSOFT*, 2013.
9. P. S. Duggirala, L. Wang, S. Mitra, M. Viswanathan, and C. Muñoz. Temporal precedence checking for switched models and its application to a parallel landing protocol. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 215–229, 2014.
10. M. Fränzle and J. Lygeros, editors. *17th International Conference on Hybrid Systems: Computation and Control (part of CPS Week), HSCC'14, Berlin, Germany, April 15-17, 2014*. ACM, 2014.
11. G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler. Spaceex: Scalable verification of hybrid systems. In *Computer Aided Verification*, pages 379–395. Springer, 2011.
12. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Hytech: A model checker for hybrid systems. In *Computer aided verification*, pages 460–463. Springer, 1997.
13. D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. *The Theory of Timed I/O Automata*. Synthesis Lectures on Computer Science. Morgan Claypool, November 2005. Also available as Technical Report MIT-LCS-TR-917.
14. K. G. Larsen, P. Pettersson, and W. Yi. Uppaal in a nutshell. *International Journal on Software Tools for Technology Transfer (STTT)*, 1(1):134–152, 1997.
15. W. Lohmiller and J. J. E. Slotine. On contraction analysis for non-linear systems. *Automatica*, 1998.
16. K. Manamcheri, S. Mitra, S. Bak, and M. Caccamo. A step towards verification and synthesis from simulink/stateflow models. In *H.S.C.C*, 2011.
17. N. Nedialkov. Vnode-lp: Validated solutions for initial value problem for odes. Technical report, 2006.
18. T. Nghiem, S. Sankaranarayanan, G. Fainekos, F. Ivancic, A. Gupta, and G. Pappas. Monte-carlo techniques for falsification of temporal properties of non-linear hybrid systems. In *H.S.C.C*, 2010.
19. L. Zou, N. Zhan, S. Wang, M. Franzle, and S. Qin. Verifying simulink diagrams via a hybrid hoare logic prover. In *EMSOFT*, 2013.