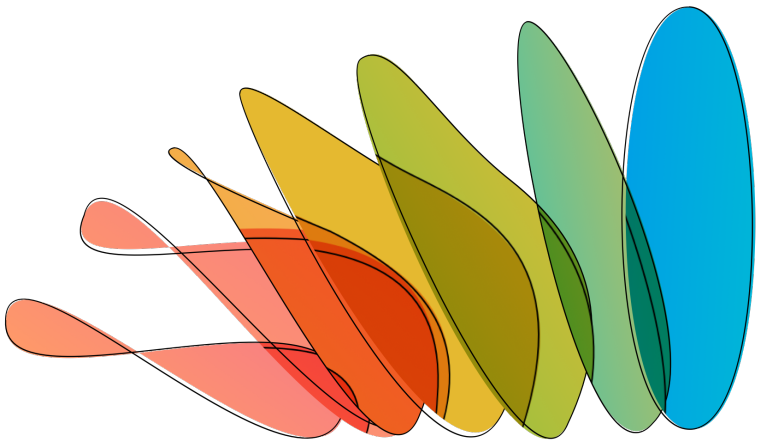
C2E2 TEAM

C2E2HELP@GMAIL.COM

C2E2 USER'S GUIDE

VER 2.0 DECEMBER 2018

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN



c2e2help@gmail.com

Compiled on April 6, 2019

Contents

1
Acknowledgements

Lucas Brown ○ Chuchu Fan ○ Parasara Sridhar Duggirala ○ Suket Karnawat ○ Yu Meng ○ Sayan Mitra ○ Matthew Potok ○ Bolun Qi ○ Mahesh Viswanathan ○ Department of Computer Science and Department of Electrical and Computer Engineering at the University of Illinois at Urbana Champaign ○ Coordinated Science Laboratory ○ The National Science Foundation ○ Air Force's Office of Scientific Research.

2
Introduction

C2*E*2 is a tool for verifying time-bounded invariant properties of hybrid automata models. It supports models with nonlinear dynamics, discrete transitions, and sets of initial states. The invariant properties have to be specified by conjunctions of linear inequalities. Internally, C2*E*2 implements the simulation-based verification algorithms described in the sequence of publications Fan et al. [2016], Fan and Mitra [2015], Duggirala et al. [2013a, 2014], Sukumar and Mitra [2011]. The new version of C2*E*2 uses an on-the-fly discrepancy computation algorithm Fan and Mitra [2015] to automatically generate neighborhoods that conservatively contain all the behaviors of neighboring trajectories. In a nutshell, C2*E*2 parses and transforms the hybrid automata model to a mathematical representation, it generates faithful numerical simulations of this model using a validated numerical simulator, it then bloats these simulations using on-the-fly discrepancy computation to construct over-approximations of the bounded time reachable set, and it iteratively refines these over-approximations to prove the invariant or announce candidate counterexamples.

C2*E*2 has a GUI for loading and editing of hybrid automata models and properties in an internal HyXML format, launching the verifier, and for plotting 2D sections of the reach set computed by the verifier.

Please contact us at c2e2help@gmail.com and let us know about your experiences using C2*E*2. Previous versions of C2*E*2 allowed for the editing of Stateflow™ models and readily supported numerical simulators like VNODE-LP Nedialkov [2006]. Some of these features are getting depreciated because of numerical instability, lack of support, etc. Let us know if you are interested in these features.

3
Installation

To install C2E2, you may either download and run the installation files, or download and virtual machine with the most recent version of C2E2 installed.

Example .hyxml files are included and located in the `examples` folder.

3.1 Download and install latest C2E2

Ubuntu 16.04, 64-bit is currently the only supported operating system. If you are using a different OS, we recommend that you download the virtual machine instead.

- Download the latest distribution of C2*E*2 distribution from: `http://publish.illinois.edu/c2e2-tool/download/`.
- Unzip the files in a local directory, say `~/c2e2/`.
  `tar -xvzf c2e2-2.0.0.tar.gz -C ~/c2e2/`
- Navigate to that directory and run installRequirements.sh with superuser privileges. This should install all the packages needed and may take a while.
  `cd ~/C2E2/`
  `sudo ./installRequirements.sh`
- Launch the GUI for loading, editing, and verifying models.
  `./runc2e2`
- If any of the above steps fail then you can try to install the packaged listed in the file `installRequirements` separately.

3.2 Install the Virtual Machine

The virtual machine can be download from http://publish.illinois.edu/c2e2-tool/download/. Oracle VM VirtualBox was used to create and test the Open Virtual Appliance (.ova).

4
Key Improvements in Version 2.0

One of our main focuses for C2*E*2 version 2.0 was to improve the usability of our software. Previously, editing models had to be done in the `.hyxml` files outside of C2*E*2. To fix this, we made two significant updates to our GUI:

- Models are now editable directly from the GUI on the *Model* tab.
- The new *Editor* tab allows you to edit the HyXML directly without closing C2*E*2

In addition, the *Plot* tab has been updated to improve usability and flexibility:

- The plotter now creates an interactive HTML.
- Any data file generated by C2*E*2 can be loaded into the plotter at any time.

5
Getting Started

In this chapter, we give a quick tour of some features of C2*E*2 using one of the examples that are distributed as part of the package.

5.1 Opening a Model

In the C2*E*2 folder, type the command `./runC2E2` to launch C2*E*2 and you should be able to see the front end of the tool as Figure 5.1. Once C2*E*2 is launched, go ahead and open one of the examples from the `File` menu (or use `Ctrl + O`). All examples are stored in the `examples` folder inside C2*E*2 folder. For this tutorial, we will use the model of an adaptive cruise control system (see the example webpage [1]) which is stored as `TotalMotion40s.hyxml`. For the description of other examples, please refer to the examples webpage [2].

Upon opening the file , the C2*E*2 window should look similar to Figure 5.2. The left hand side of this window is the hybrid automaton *model tree* and the right hand side is the *requirements sidebar*.

[1] https://publish.illinois.edu/C2*E*2-tool/example/adaptive-cruise-control/

[2] https://publish.illinois.edu/C2*E*2-tool/example/

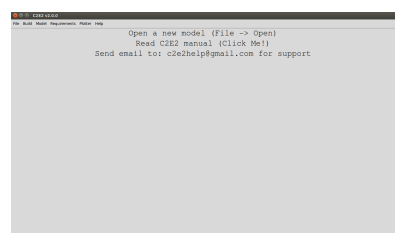


Figure 5.1: Front end of C2*E*2 when launched

5.2 Model Tree

When a model is opened, the HyXML is parsed and the result is displayed in the expandable *model tree*. Automata are at the base, followed by Variables, Modes, and Transitions. Modes can further be expanded to display Flows and Invariants. Every item on the tree can be expanded to view details by clicking on the arrow to the left. As you can see, our example in Figure 5.2 has five variables *sx*, *vx*, *ax*, *sy*, *vy* and *omega*. In addition, we have seven separate modes with six transitions between them.

In version 2.0, you can now also edit the model directly from the *model tree* by right-clicking the item you wish to edit, or by selecting it

and using the options under the *Build* menu. We will go over this in more detail in the next chapter.

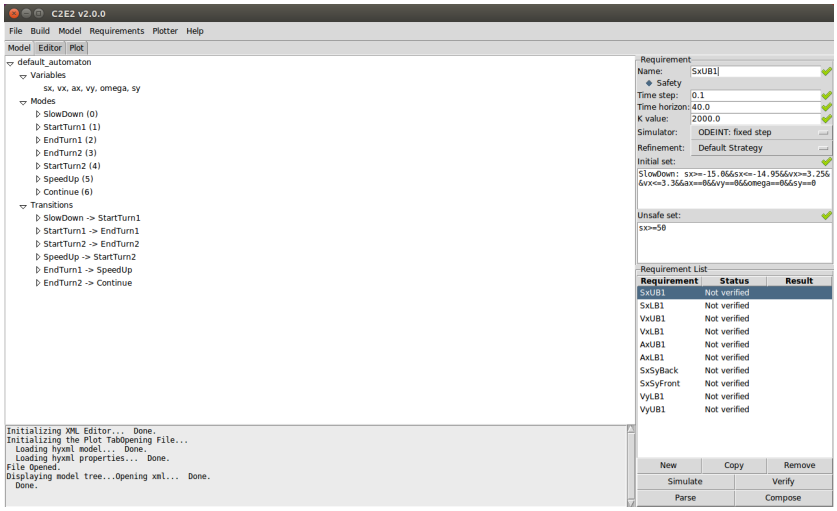


Figure 5.2: *Left:* Model parse tree. *Right:* Verification pane.

5.3 Requirements Sidebar

The lower part of the sidebar shows a list of requirements (properties) for the hybrid automaton that can be verified. The top part of the sidebar shows the detailed parameters of the highlighted requirement. Each requirement is a bounded time safety property.

- *Name*
  Each requirement (property) to be checked is given a name. This name is used to name output files containing the reachtubes.

- *Time Horizon*
  The time bound for simulation and verification.

- *Time Step*
  The time step-size used for simulation and verification. If adaptive step size is used for simulation (see below) then this value is ignored.

- *K Value*
  The coordinate transformation step K for nonlinear models has been set to 2000 by default at the top right corner of the verification pane. This value is important since the inappropriate K value will influence the final result. The user can change the K value to see different outputs.

- *Simulator*
  Current version of C2*E*2 supports Odeint constant time step simulator and Odeint adaptive time step simulator. Previously, CAPD simulator was also supported, but no longer is in the current version. Please contact us if you wish to use the CAPD simulator. The default simulator is Odeint constant time step simulator, and it is been compiled when you opening the model. You can change the simulator by selecting different simulator from the simulator drop down menu as shown in Figure 5.3.

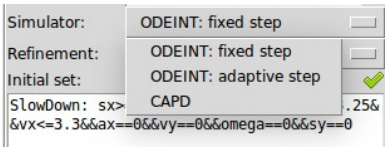


Figure 5.3: Simulator drop down menu

- *Refinement*
  When the initial set needs refinement, C2*E*2 provides different refinement strategies. The default refinement strategy will refine the dimensions within the unsafe set for four times, then iteratively refine the dimension with the largest uncertainty size. C2*E*2 also supports user defined strategy, which can be found at Section 5.5.2. To select the strategy, please use the drop down menu on GUI as shown in Figure 5.4.

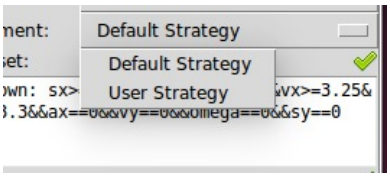


Figure 5.4: Refinement drop down menu

- *Initial set*
  A linear predicate on the variables to specify the *initial set* or *starting states* on the `Initial set` textbox. Currently, the syntax for specifying the initial set is as follows:

- *Unsafe set*

At the bottom of the *requirement sidebar* is the *requirement list*. This is where you can add, edit, copy properties and launch the verifier or simulator. Currently C2*E*2 verifies *bounded time linear invariant properties from linear bounded initial sets*. Such properties are specified by the time bound (T), the initial set and the unsafe set. The *Time horizon* parameter listed at the top of the verification pane is the time bound. Currently C2*E*2 requires both the initial and the unsafe sets to be described by a conjunction of linear inequalities involving the model variables. The models in the `Examples` folder have already has a couple of sample properties.

5.4 Creating a New Property

Here we will walk you through the steps involved in creating a new property like in Figure 5.5.

1. Click `New` in the property pane. This opens an empty new `Property` box on the Right panel in the middle for editing.

2. Enter a name for the property at the top, say `VxLB1`, in the first textbox.

3. Enter a linear predicate on the variables to specify the *initial set* or the *starting states* in the `Initial set` textbox. Currently, the syntax for specifying the initial set is as follows:
   ⟨ `mode-name` ⟩ : ⟨ (`linear-inequality` &&)+ ⟩.
   For example, for the above model:
   `SlowDown`: sx>=-15.0 && sx<=-14.95 && vx>=3.25 && vx<=3.3 && ax==0 && vy==0 && omega==0 && sy==0
   is a valid expression for specifying the set of initial states.

4. Enter the unsafe set in the `Unsafe set` textbox. Currently, the syntax for specifying the unsafe set is a &&-separated sequence of linear inequalities:
   vx<=2.1

5. Press `Add`.

If all the expressions are syntactically acceptable then there will be little green checks next to the textboxes and you will be able to save the property. Otherwise there will be a cross next to the textbox. **Both the unsafe set and the initial set should be described by a collection of linear inequalities and in addition the initial set should be bounded.**

Once the property is added the name of the property appears in the property pane. You may add several properties in the same way. You may also make copies of existing properties to save yourself some typing and edit them. The added properties can be saved with the model. See section **??**.

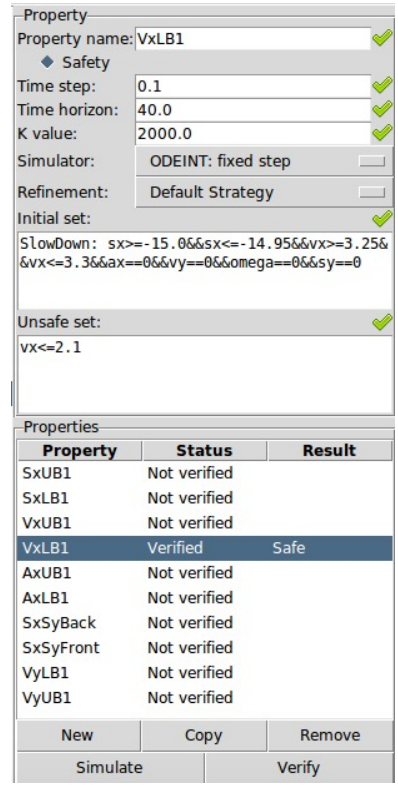


Figure 5.5: Dialog box for adding properties checks the syntax of the initial and unsafe sets.

5.5 Verifying

Once you have created a model and added a property (see Section 5.3) you can launch the verification engine by selecting the property and then clicking the `Verify` button.

C2*E*2 is sound which means that you can trust the Safe/Unsafe answer proclaimed by it. In principle, C2*E*2 is also complete for robust properties Duggirala et al. [2013a]. That is, if the model satisfies the property robustly[3], and if the numerical precision supported by the algorithm is adequate then C2*E*2 should terminate with a Safe/Unsafe proclamation. In practice, the time it takes to verify is sensitive to the time horizon (T), the initial partition. You may want to first run the verification with small values of T and initial set with small size.

[3] Robustness: the requirement that the actual reachable set of the model does not skim the boundary of the unsafe set.

The reachable set over-approximation computed by C2*E*2 is stored in the `/work-dir/<Requirement name>`. You can also check the log file at `/work-dir/log` to check the progress of the verification. Once the verification is done, the result (Safe/Unsafe/Unknown) will show up at the Result column (as in Figure 5.6). Note that if you see verification result as `Unknown`, it is because of the following reason:

1. The system is neither robustly safe nor robustly unsafe Duggirala et al. [2013a].

2. Reachable set computed bloats up and thus the number of refinements needed is too large. Please go back and check the model dynamic and properties, or simulate first to see whether the system trajectories bloats up.

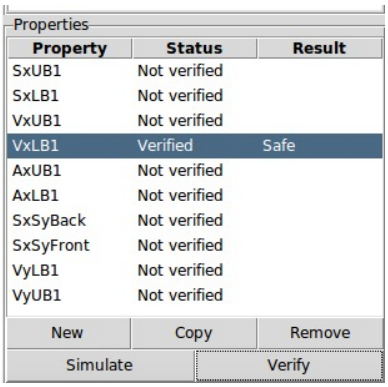


Figure 5.6: One or more properties can be selected by checking the boxes to the left of the property name. The `Verify` button launches the verification engine to verify one property at a time.

5.5.1 *Simulation*

C2*E*2 also allows users to generate pure simulation traces from initial sets. Once you have created a model and added a property (see Section 5.3) you can launch the simulation engine by selecting the property and then clicking the `Simulate` button. C2*E*2 will select several states from initial set and generate simulation traces from those initial states. Note the Safe/Unsafe result shown in this case only stands for the safety of the simulation traces instead of all the reachable states from the initial set.

5.5.2 *User defined refinement strategy*

C2*E*2 supports user defined refinement strategy (see Section 5.1). You can select USER DEFINE STRATEGY from the drop down menu as shown in Figure 5.4. In this case, you need to write down your strategy in a file named `refineorder.txt` and store it in the `/work-dir` folder. The file should be look like Figure 5.7. In each line, you should write down the index of the variables, the order of which is the same as the variables that are shown in the front end GUI. That is, "2" means the second variable shown in the Variables list in the front end. Indexes that are larger than the dimension of system will be ignored automatically. C2*E*2 will refine the initial set according to the order written in `refineorder.txt` iteratively. For example, if use the refinement strategy as in Figure 5.7, the dimension corresponding to the second variable will be refined three times, then the dimension corresponding to the first variable will be refined once, then go back to the first line of `refineorder.txt` if the verification process has not terminated.

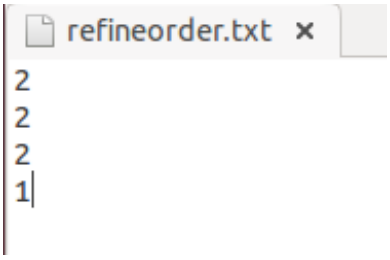


Figure 5.7: The user defined refine strategy file

5.5.3 *Change Verified Properties*

Once a property is verified the status of the property will change to `Verified`, and the result `Safe/Unsafe/Unknown` will appear next to it. C2*E*2 will also create a new plot on the *Plot Tab* for each verified property with the name ⟨`property name`⟩ `plot` (see Section 9). Once the result has shown up, if you change any of the parameters associated with the property, say the initial set or unsafe set, then the status of the property will change to `Verified*`. This (*) indicates that the property and parameters verified is outdated and you can launch the verifier again.

6
Model (HyXML) Creation and Editing

- The new text editor tab, brief overview of xml. refer to details given at the end of the manual. Screen shot of editor tab.
- The graphical editing process. Variables, automata, transitions. Discuss all the dialog boxes. Screenshots. The error checks.
- Properties. New, copy, error checks.

6.1 Editing in the GUI

C2*E*2 version 2.0 allows editing directly in the GUI. To edit any item in the *model tree*, simply right-click it or use the *Model* menu in the menu bar.

6.1.1 *Automata*

Automata are not edited as a whole, but rather they are edited by adding, editing, and removing the variables, modes, and transitions in the automaton. The automaton name can be edited through the *Automaton* dialog. Similarly, when adding a new automaton, the name can be set in the same *Automaton* dialog.

To add an automaton, right-click the *model tree* on an automaton name or in a blank area and select *Add Automaton*. You can also add an automaton through the menu bar, *Model* -> *Automaton* -> *Add Automaton*. To edit an automaton, the same procedures can be followed, however an automaton must be selected.

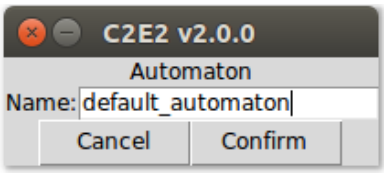


Figure 6.1: Automaton Dialog

6.1.2 *Variables*

Variables are added, edited, and deleted through the *Variable* dialog box. The *Variable* dialog can be accessed through right-clicking the

model tree on the *Variable* heading or on the variables themselves. It can also be accessed through the menu bar, *Model -> Variables -> Edit Variables*. If you have two or more automata in the model, one of them must be selected.

Variables can be set as thin by selecting the check box next to their name. Variable type can be *Real* or *Integer*; variable scope can be *Local*, *Input*, or *Output*. To delete a variable, erase it's name and leave the field blank, then press *Confirm*.

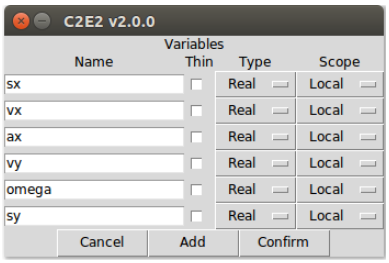


Figure 6.2: Variables Dialog

6.1.3 Modes

Modes are added, edited, and deleted through the *Mode* dialog box. The dialog box is accessed through right-clicking the *model tree* on the *Mode* heading or the modes themselves. It can also be accessed through the menu bar, *Model -> Modes -> Add Mode* or *Edit Mode* or *Delete Mode*.

When adding modes, if you have more than one automaton in your model, one of the automata must be selected - this can be done by right-clicking within the correct automaton, or by having anything in the correct automaton selected when navigating through the menu bar. When editing or deleting modes, the correct mode must be selected.

To set the mode as the initial mode, make sure the corresponding box is selected. add a new Flow or Invariant, click "Add Row" below the appropriate heading. To delete a Flow or Invariant, clear out the equation and leave the filed blank when confirming the dialog. As you can see in Figure 6.3, the ID field is disabled. IDs are used internally and do not need to be edited by the user - they are displayed because they can be set when editing the HyXML directly.

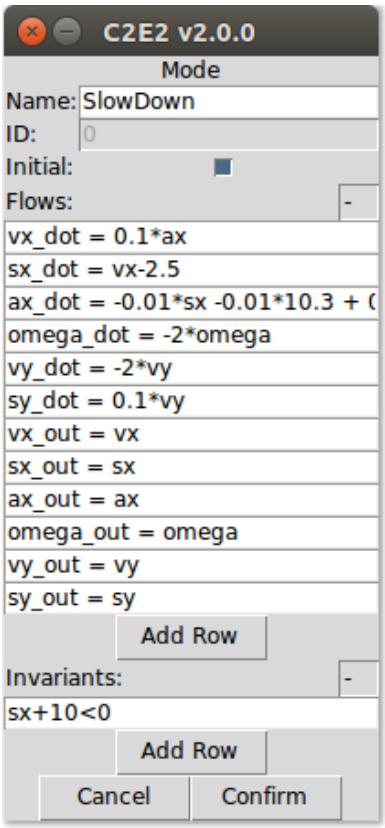


Figure 6.3: Modes Dialog

6.1.4 Transitions

Transitions are added, edited, and deleted through the *Transition* dialog box. The dialog box is accessed through right-clicking the *model tree* on the *Transition* heading or the transitions themselves. It can also be accessed through the menu bar, *Model -> Transition -> Add Transition* or *Edit Transition* or *Delete Transition*.

When adding transition, if you have more than one automaton in your model, one of the automata must be selected - this can be done by right-clicking within the correct automaton, or by having anything in the correct automaton selected when navigating through the menu bar. When editing or deleting transitions, the correct transition must be selected.

Transitions do not have names, but rather are defined by their source and destination modes. The source and destination are set by

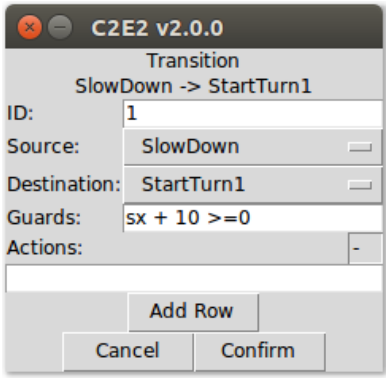


Figure 6.4: Transition Dialog

the drop-down menus, which contain all of the modes currently in the automaton. Currently, C2*E*2 supports a singe guard per transition, so if you to include multiple guards, you need to create multiple transitions. Multiple actions are support, and they can be added by clicking the *Add Row* button.

6.2 HyXML Format

In this section, we'll review the basics of the HyXML file format. HyXMML files can be edited directly in C2*E*2 on the *Editor* tab.

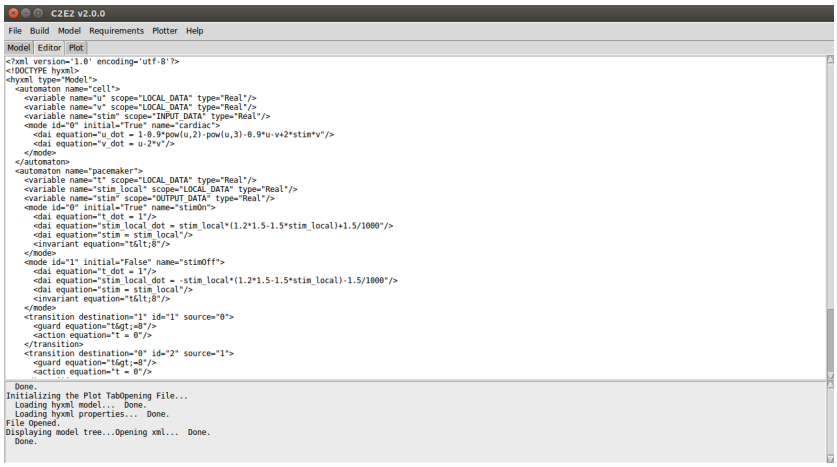


Figure 6.5: Editor Tab.

6.2.1 *Header*

The first few lines of are very familiar to the XML format.

```
<?xml version='1.0' encoding='utf-8'?>
<!DOCTYPE hyxml>
<hyxml type="Model">
```

6.2.2 *Automaton*

The first thing we add to our model is an automaton. To add an automaton, use an `automaton` element, and set it's name with the `name` attribute. All of our Variables, Modes, and Transitions for this automaton will lie between the opening and closing `automaton` tags.

```
<automaton name="default_automaton">
</automaton>
```

6.2.3 *Variables*

Next, we add variables to out automaton. To add a varaible, use a `variable` element, and set it's `name`, `scope`, and `type` attributes.

```
<variable name="sx" scope="LOCAL_DATA" type="Real"/>
```

6.2.4 *Modes*

Adding mode requires three different elements. The `mode` element creates the mode and has attributes `id`, `initial`, and `name`. Within an automaton, only one mode can be the initial mode, and mode IDs must be unique.

After adding the opening `mode` tag, flows and invariants need to be added with the `dai` and `invariant` elements, respectively. The `dai` and `invariant` elements both have the same single attribute, `equation`.

```
<mode id="0" initial="True" name="SlowDown">
 <dai equation="vx_dot" = 0.1*ax"/>
 <invariant equation="sx + 10 &lt; 0"/>
</mode>
```

6.2.5 *Transitions*

Transitions are similar to modes in the HyXML. First, a `transition` element must be added. Guards and actions are then added with the `guard` and `action` elements, respectively. The `transition` element has three attributes, `source`, `destination`, and `id`. The `guard` and `action` element both have the same single attribute, `equation`.

```
<transition source="0" destination="1" id="1"
 <guard equation="sx + 10 &gt; = 0">
</transition>
```

7
Composition, Parsing, and Analysis

In this chapter, we describe in more detail the key operations on hybrid automata that can be performed by C2*E*2. The input a .hyxml file generally contains (i) A list of *component hybrid automata* $\mathcal{A}_1, \ldots, \mathcal{A}_k$, (ii) the definition of (one or more) *composed hybrid automaton* $\mathcal{A}$, (iii) a list of *safety requirements* (See Section **??**). The C2*E*2 workflow has the following steps: (i) The component automata $\mathcal{A}_1, \ldots, \mathcal{A}_k$ are *composed* according to the definition of the composed automaton $\mathcal{A}$ to generate the complete (internal) representation of $\mathcal{A}$. (ii) The automaton model $\mathcal{A}$ is parsed. This generates internal files used for simulation and verification. (iii) Once parsed successfully, $\mathcal{A}$ can be either simulated or verified against the requirements. (iv) Simulations draw random initial states and generate simulation traces and checks for safety of those traces. (v) Verification uses the algorithm presented in **??** to check the safety property of the give model. In the rest of this chapter, we mention some of the key aspects of these four steps. For details on the mathematical model used here and the algorithms, we refer the reader to Duggirala et al. [2014], **?**.

7.1 Composing

It convenient to define a complex model in terms of its components. The composition operation for hybrid automaton defines a bigger (more complex) hybrid automaton $\mathcal{A}$ in terms of a pair of component automata $\mathcal{A}_1$ and $\mathcal{A}_2$. This binary operation can be applied repeatedly to compose a finite set of automata.

A hybrid automaton $\mathcal{A}$ in C2*E*2 is defined by the following following:

(i) L: A finite set of *modes* or *locations*.

(ii) X, U, Y: Finite pair-wise disjoint sets of *state (local), input*, and *output variables*. State variables are called LOCAL in HyXML.

We will denote the set of valuations for these variables by $val(X), val(Y), val(U)$, etc.

(iii) $\mathcal{D} \subseteq L \times L$: A discrete transition graph. For each edge in the graph $(\ell, \ell') \in \mathcal{D}$, two functions are specified. First, $guard(\ell, \ell') : val(X) \rightarrow \mathbb{B}$ is a *transition precondition* , and $reset(\ell, \ell') : val(X) \rightarrow val(X)$ is a (linear) reset function.

(iv) $\{f_\ell\}$: Differential equations describing the evolution of $X \cup Y$ (with U as input) for each mode $\ell \in L$.

When discussing several automata $\mathcal{A}$, $\mathcal{A}_1$, $\mathcal{A}_2$, etc., we will use subscripts $X_\mathcal{A}, X_1, X_2$, and $L_\mathcal{A}, L_1, L_2$, etc. to disambiguate components coming from different automata.

For the composition of a pair of hybrid automata $\mathcal{A}_1$ and $\mathcal{A}_2$ to be well-defined, they must be *compatible*. That is, the following conditions must be met:

- $(X_1 \cup Y_1) \cap (X_2 \cap Y_2) = \emptyset$,
- $U_1 \subseteq Y_2$ and $U_2 \subseteq Y_1$

In the current implementation of C2*E*2, $\mathcal{A}_1$ and $\mathcal{A}_2$ can only interact through shared variables, and not through shared transitions.

For a pair of compatible automata $\mathcal{A}_1$ and $\mathcal{A}_2$ the Compose operation of C2*E*2, constructs the composed automaton $\mathcal{A}$ (as defined below). Once this composed model is saved, a new HyXMLfile with the composed automaton is saved.

(i) $L_\mathcal{A} = L_1 \times L_2$,

(ii) $X_\mathcal{A} = X_1 \cup X_2$, $Y_\mathcal{A} = Y_1 \cup Y_2$, $U_\mathcal{A} = \emptyset$.

(iii) $((\ell_1, \ell_2), (\ell_1', \ell_2')) \in \mathcal{D}_\mathcal{A}$ iff $(\ell_1, \ell_1') \in \mathcal{D}_1$ and $\ell_2 = \ell_2'$, or $(\ell_2, \ell_2') \in \mathcal{D}_2$ and $\ell_1 = \ell_1'$.

(iv) $\{[f_{1,\ell_1}; f_{2,\ell_2}]\}$ for each mode $(\ell_1, \ell_2) \in L_\mathcal{A}$, the differential equations for $X_\mathcal{A} = X_1 \cup X_2 \cup Y_1 \cup Y_2$, are derived by simply concatenating the differential equations for $(X_1 \cup Y_1)$ in mode ℓ_1 with those for $(X_2 \cup Y_2)$ in mode ℓ_2. The differential equations for each mode of $\mathcal{A}$ are derived from those of the component modes.

7.2 Parsing

Before any simulation or verification can happen, the system must first be parsed. When parsing, we perform a variety of checks to

make sure the system is capable of being simulated and verified. In the following sections we enumerate these checks (see Section 6.1 for details about the HyXML format).

7.2.1 *Variables*

- Local variables must all be distinct from all other variables in the model
- Output variables must be distinct from all local variables
- Output variables must be distinct from all other output variables
- Input variables must be distinct from all local variables
- Input variables must be distinct from other input variables within the same automaton
- Every input variable must have a matching output variable in a different automaton

7.2.2 *Mode*

Mode names and mode IDs must be unique within an automaton. Note that mode IDs can only be changed through editing the HyXML directly.

7.2.3 *Flows*

- Expressions must be standard mathematical expressions and use standard mathematical operators
- Expression contains only one symbol on the left-hand side
- Output variables on right-hand side must not end with _dot
- Local variables on left-hand side must end with _dot
- Input variables must not be used on the left-hand side

7.2.4 *Invariants*

- Expressions must be standard mathematical expressions and use standard mathematical operators.

7.2.5 *Transition Guards*

- Expressions must be standard mathematical expressions with standard mathematical operators.
- Expressions must be separated by "&&".

7.2.6 *Transition Actions*

- Expressions must be standard mathematical expressions with standard mathematical operators.
- Expressions must be separated by "&&".

Compatibility checks. This happens here or in compose?
List of error messages and their interpretations.

7.3 Simulation

When the Simulate button is clicked, C2*E*2 transforms the input hybrid automaton model and generates faithful numerical simulations of this model. The type of simulator used can be chosen by the user. The most robust simulator is ODEInt from the Boost library. Other validated numerical simulators like VNODE-LP Nedialkov [2006] and CAPD are also supported but because of numerical instability issues, using these simulators may require some extra work in the latest version of C2*E*2.

Roughly, the simulation procedure works as follows. A sequence of initial states are drawn randomly from the specified starting set. From each of these initial states, a simulation in the current mode is generated using the C++ simulator function. This function is compiled when the HyXML file is parsed. Once the simulation in the current mode is computed, the result is checked to see if any guards are hit. If any of the guards are hit, the simulation is truncated, the initial state in the next mode is computed using the reset function, and the simulation is continued in the new mode for the remaining time. If none of the guards are hit, the simulation is complete. This procedure is repeated for all the initial state. If any of the simulations hit the specified "unsafe set", then the result unsafe is returned, otherwise the result "safe" is returned. Of course, "safe" does not mean that all behaviors from the starting set are safe. The simulation output is stored in a text file in the working directory.

7.4 Verification

There are several papers describing the C2*E*2 verification algorithms Fan and Mitra [2015], Fan et al. [2016], Duggirala et al. [2013b], Fan et al. [2016], Duggirala et al. [2014] and therefore we are not going to repeat them here. For linear systems a global discrepancy function is used, while for nonlinear systems the local discrepancy

computation presented in Fan and Mitra [2015], Fan et al. [2018a] is implemented. The reach tubes computed for verification are stored in a textfile in the working directory.

You may want to look at some of the applications of C2E2 in verification of cyber-physical systems: Toyota's Powertrain control system Duggirala et al. [2015], NASA landing alarm system Duggirala et al. [2014], Complex transistor models Fan et al. [2018b], model of cardiac pacemaker Huang et al. [2014, 2015], and autonomous spacecraft maneuvers Chan and Mitra [2017]. Several other examples are reported in the ARCH competition of 2018 Althoff et al. [2018]. We will be happy to hear about your applications; send email to c2e2help@gmail.com.

8
Plotter

C2*E*2 version 2.0 comes with a completely overhauled plotter. On the new *Plot* tab, you can plot any data set you choose, and the result will be an interactive HTML plot that you can use to focus in on certain parts of your data.

The *Plot* tab has two main components - the *plot pane*, where small images on the plots are displayed, and the *plot sidebar*, where plots can be created and edited.

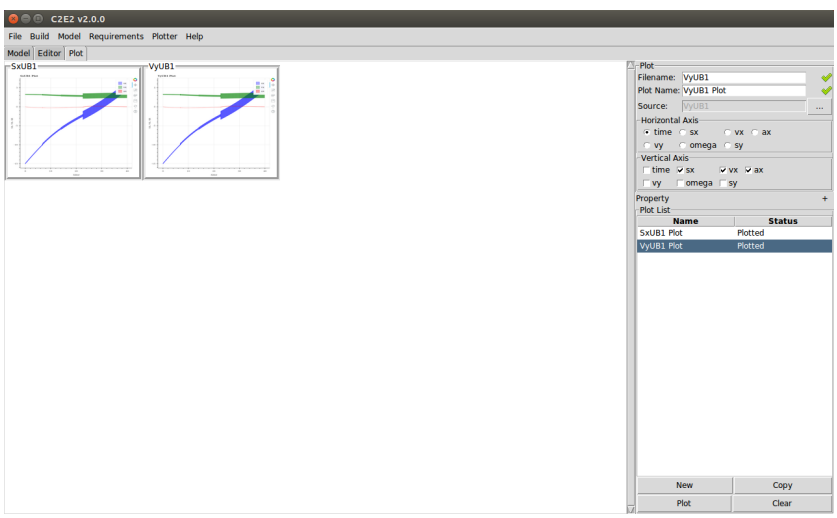


Figure 8.1: Plot Tab.

8.1 Creating and Editing Plots

To create a plot, the first thing you need to do is select the data source. To do this, select on the button next to the *Source* field (labeled "..."), and choose which file to load - the data files generated by C2*E*2 are stored in the `/work-dir/output/` directory. As mentioned, this can be any data source generated by C2*E*2, it does not have to be

related to the model you currently have open in the *Model* and *Editor* tabs. Next, fill out the *Filename* and *Plot Name* fields.

- Filename
  This represents the filename used to store the various plots generated by the plotter. Two files will be generated using this name, *filename*.html and *filename*.png. The PNG file is the preview image displayed done the *plot pane*, and the HTML file in the interactive plot.
- Plot Name
  This the title as it will be displayed on your plot.
- Source
  This is the data source used to create the plot. The data source used must be a properly-formatted file generated by C2*E*2 when running a simulation or verification. These files will be stored in the `/work-dir/output/` directory and be named after the requirement that created them.

If you recently simulated or verified a requirement, you may notice that you have this field, as well as plot name and file name, already filled out. This is because C2*E*2 automatically creates a new plot with the fields defaulted in when a requirement is simulated or verified.

Next, you need to choose which variables will appear on the vertical axis, and which will appear on the horizontal axis. You can only select one variable to be on the horizontal axis, and you must choose at least one variable to be on the vertical axis.

Finally, click on the *Plot* button. Once the plotting is complete, you will see a preview appear on the *plot pane*. You can double-click on this image to open the interactive HTML plot in your default browser.

8.2 Multiple Plots

Multiple plots can be added by using the *New* and *Copy* buttons, and they will be displayed in the *Plot List* on the *Plot Sidebar*.

8.3 Requirements

The requirements (properties) that were used to create the source data file can be displayed by clicking on the + next to the *Properties* label.

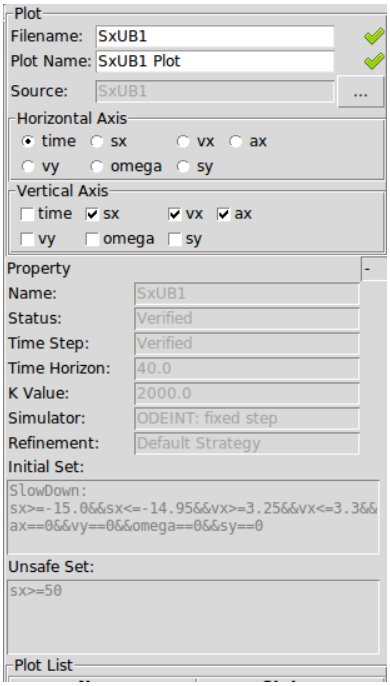


Figure 8.2: Requirements Display

9
Building models for C2E2

In this chapter, we will discuss two example C2*E*2 models.

9.1 Example 1: Laub-Loomis

The first example mentioned is a continuous system with nonlinear dynamics. The example we used is the Laub-Loomis model. It is a model with seven variables. The dynamics of the model is given by.

$$\begin{cases} \dot{x}_1 &= 1.4x_3 - 0.9x_1 \\ \dot{x}_2 &= 2.5x_5 - 1.5x_2 \\ \dot{x}_3 &= 0.6x_7 - 0.8x_2x_3 \\ \dot{x}_4 &= 2 - 1.3x_3x_4 \\ \dot{x}_5 &= 0.7x_1 - x_4x_5 \\ \dot{x}_6 &= 0.3x_1 - 3.1x_6 \\ \dot{x}_7 &= 1.8x_6 - 1.5x_2x_7 \end{cases}$$

In this example, we use initial set given by $x_1(0) \in [1.1, 1.3]$, $x_2(0) \in [0.95, 1.15]$, $x_3(0) \in [1.4, 1.6]$, $x_4(0) \in [2.3, 2.5]$, $x_5(0) \in [0.9, 1.1]$, $x_6(0) \in [0, 0.2]$, $x_7(0) \in [0.35, 0.55]$, and the unsafe set is $x_4 \leq 5$. The model is running in time interval $t \in [0, 20]$. The problem can be described as follows.

```
<hyxml type="Model">
   <automaton name="default_automaton">
      <!-- specify variables -->
      <variable name="x1" scope="LOCAL_DATA" type="Real" />
      <variable name="x2" scope="LOCAL_DATA" type="Real" />
      <variable name="x3" scope="LOCAL_DATA" type="Real" />
      <variable name="x4" scope="LOCAL_DATA" type="Real" />
      <variable name="x5" scope="LOCAL_DATA" type="Real" />
      <variable name="x6" scope="LOCAL_DATA" type="Real" />
      <variable name="x7" scope="LOCAL_DATA" type="Real" />
```

```
      <!-- specify modes -->
      <mode id="0" initial="True" name="Model">
         <!-- specify dynamic equations in mode -->
         <dai equation="x1_dot = 1.4*x3-0.9*x1" />
         <dai equation="x2_dot = 2.5*x5-1.5*x2" />
         <dai equation="x3_dot = 0.6*x7-0.8*x2*x3" />
         <dai equation="x4_dot = 2-1.3*x3*x4" />
         <dai equation="x5_dot = 0.7*x1-x4*x5" />
         <dai equation="x6_dot = 0.3*x1-3.1*x6" />
         <dai equation="x7_dot = 1.8*x6-1.5*x2*x7" />
      </mode>
   </automaton>
   <composition automata="default_automaton"/>

   <!-- setup the initial set and unsafe set of the constraints -->
   <property
      initialSet="Model:
         x1&gt;=1.1 &amp;&amp; x1&lt;=1.3 &amp;&amp;
         x2&gt;=0.95 &amp;&amp; x2&lt;=1.15 &amp;&amp;
         x3&gt;=1.4 &amp;&amp; x3&lt;=1.6 &amp;&amp;
         x4&gt;=2.3 &amp;&amp; x4&lt;=2.5 &amp;&amp;
         x5&gt;=0.9 &amp;&amp; x5&lt;=1.1 &amp;&amp;
         x6&gt;=0 &amp;&amp; x6&lt;=0.2 &amp;&amp;
         x7&gt;=0.35 &amp;&amp; x7&lt;=0.55"
      name="large"
      unsafeSet="x4&gt;=5">
      <!-- specify the parameters for computation -->
      <parameters
         kvalue="50.0"
         timehorizon="20.0"
         timestep="0.05" />
   </property>
</hyxml>
```

C2E2 is able to solve the model with time step 0.05 and k is set to be 50. The computed reachtube for variable x_4 is shown in Figure 9.1.

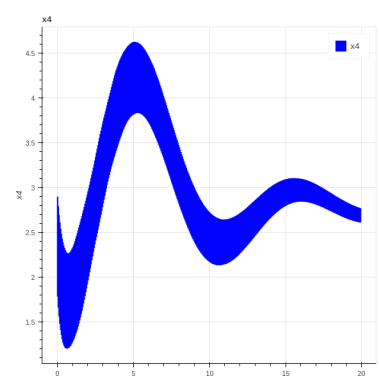
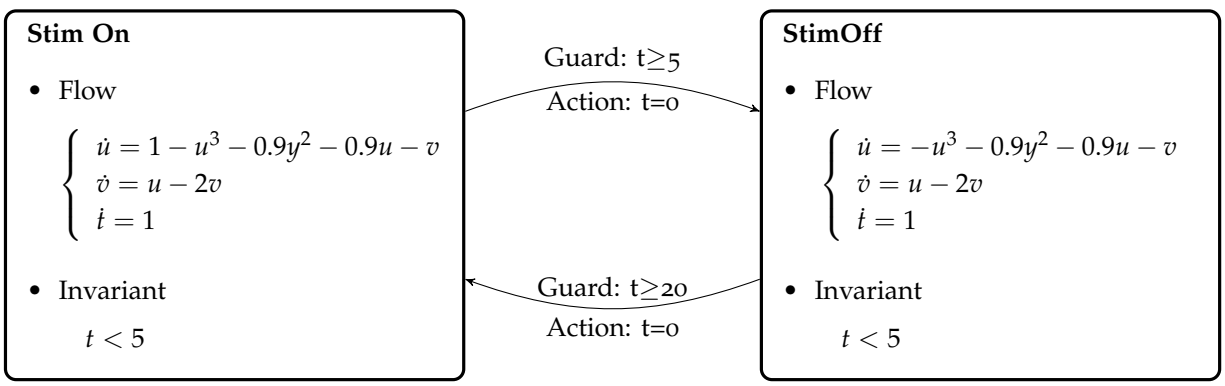


Figure 9.1: Plot for x_4 for example 1.

9.2 Example 2: Cell and Pacemaker

The second example mentioned is a hybrid system. The example we used is the FitzHugh-Nagumo (FHN) model. The model has 2 variables and 2 modes. The dynamic for each mode is give by hybrid automaton.

In this example, we use initial set given by $u(0) = 0$ and $v(0) = 0$ and a variable t is introduced to track time spend in each mode. The model is running in time interval $t \in [0, 40]$. The problem can be described as follows.




```
<hyxml type="Model">
   <automaton name="default_automaton">
      <!-- specify variables -->
      <variable name="t" scope="LOCAL_DATA" type="Real" />
      <variable name="u" scope="LOCAL_DATA" type="Real" />
      <variable name="v" scope="LOCAL_DATA" type="Real" />

      <!-- specify modes -->
      <mode id="0" initial="True" name="stimOn_cardiac">
         <dai equation="t_dot= 1" />
         <dai equation="u_dot= 1 - pow(u,3) - 0.9*pow(u,2) - 0.9*u - v" />
         <dai equation="v_dot= u - 2*v" />
         <!-- specify invariants -->
         <invariant equation="t&lt;5" />
      </mode>
      <mode id="1" initial="False" name="stimOff_cardiac">
         <dai equation="t_dot= 1" />
         <dai equation="u_dot= 0 - pow(u,3) - 0.9*pow(u,2) - 0.9*u - v" />
         <dai equation="v_dot= u - 2*v" />
         <!-- specify invariants -->
```

```
        <invariant equation="t&lt;20" />
      </mode>

      <!-- specify invariants -->
      <transition destination="1" id="0" source="0">
        <guard equation="t&gt;=5" />
        <action equation="t = 0" />
      </transition>
      <transition destination="0" id="1" source="1">
        <guard equation="t&gt;=20" />
        <action equation="t = 0" />
      </transition>
   </automaton>
   <composition automata="default_automaton"/>

   <!-- setup the initial set and unsafe set of the constraints -->
   <property
      initialSet="stimOn_cardiac:
        u==0 &amp;&amp;
        v==0 &amp;&amp;
        t==0" name="P1"
      type="Safety"
      unsafeSet="u&gt;=2.5">
      <!-- specify the parameters for computation -->
      <parameters
        kvalue="2000.0"
        timehorizon="40.0"
        timestep="0.01" />
   </property>
</hyxml>
```

C2E2 is able to solve the model with time step 0.01 and k value 2000. The computed reachtube for variable v is shown in Figure 9.2

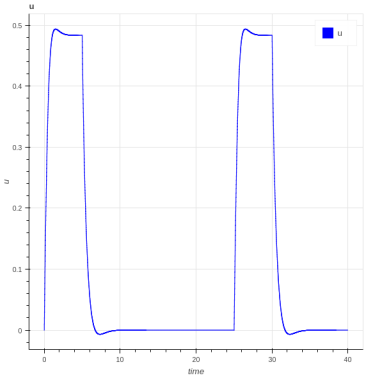


Figure 9.2: Plot for u for example 2

9.3 Intermediate Files

Several files are generated while verifying the model and some of them are worth mentioning.

jacobiannature1.txt The *jacobiannature1.txt* file contains the jacobian for the dynamic of the system. If the system have multiple modes, files with higher index will be generated and each file correspond to one mode of the system. Each file contain the number of variables,

the name of variables, the total number of entries in the jacobian matrix and the matrix itself. These files are used in computing the reach tube.

simulator.cpp The *simulator.cpp* file is auto generated by the frontend. It contains a simulator for the model for simulation engine odeint. The file can be compiled to an independent executable *simu* which is also located inside the work-dir folder. The simulator *simu* and it's corresponding configuration file *Config* (which is also located in work-dir folder), can be executed and providing simulation result by using commnad "./simu < Condig".

SimuOutput The *SimuOutput* file contains the simulated result for the model.

reachtube.dat The *reachtube.dat* file contains the computed reach tube for the model.

A
Required Libraries

The following is a complete list of packages needed for installing C2*E*2.

1. GNU Linear Programming Kit along with Python bindings, GLPK and PyGLPK (http://www.gnu.org/software/glpk/) (http://tfinley.net/software/pyglpk/)
2. GNU parser generator, Bison (http://www.gnu.org/software/bison/)
3. The Fast Lexical Analyzer, Flex (http://flex.sourceforge.net/)
4. Python (http://www.python.org/)
5. Python parsing libraries, Python-PLY (http://code.google.com/p/ply/)
6. GTK libraries for Python (http://www.pygtk.org/)
7. Plotting libraries for Python, Matplotlib (http://matplotlib.org/)
8. Packing configurations library (http://www.freedesktop.org/wiki/Software/pkg-config/)
9. GNU Autoconf (http://www.gnu.org/software/autoconf/)
10. Python xml library, lxml (http://lxml.de/installation.html)
11. Parma Polyhedron Library (http://bugseng.com/products/ppl/)
12. Python Sympy library (http://www.sympy.org/en/index.html)
13. Boost libraries (http://www.boost.org)
14. Python Numpy library (http://www.numpy.org/)
15. Python SciPy library (https://www.scipy.org/)
16. Python Pillow library (https://python-pillow.org/)
17. Python3 GnuPlot library (https://github.com/oblalex/gnuplot.py-py3k)

Bibliography

Matthias Althoff, Stanley Bak, Xin Chen, Chuchu Fan, Marcelo Forets, Goran Frehse, Niklas Kochdumper, Yangge Li, Sayan Mitra, Rajarshi Ray, Christian Schilling, and Stefan Schupp. ARCH-COMP18 category report: Continuous and hybrid systems with linear continuous dynamics. In *ARCH@ADHS*, volume 54 of *EPiC Series in Computing*, pages 23–52. EasyChair, 2018.

Nicole Chan and Sayan Mitra. Verified hybrid LQ control for autonomous spacecraft rendezvous. In *CDC*, pages 1427–1432. IEEE, 2017.

Parasara Sridhar Duggirala, Sayan Mitra, and Mahesh Viswanathan. Verification of annotated models from executions. In *EMSOFT*, pages 1–10, 2013a.

Parasara Sridhar Duggirala, Sayan Mitra, and Mahesh Viswanathan. Verification of annotated models from executions. In *Proceedings of the International Conference on Embedded Software, EMSOFT 2013, Montreal, QC, Canada, September 29 - Oct. 4, 2013*, pages 1–10. IEEE, 2013b.

Parasara Sridhar Duggirala, Le Wang, Sayan Mitra, Mahesh Viswanathan, and César 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*, volume 8442 of *Lecture Notes in Computer Science*, pages 215–229. Springer, 2014. ISBN 978-3-319-06409-3.

Parasara Sridhar Duggirala, Chuchu Fan, Sayan Mitra, and Mahesh Viswanathan. Meeting a powertrain verification challenge. In *CAV (1)*, volume 9206 of *Lecture Notes in Computer Science*, pages 536–543. Springer, 2015.

Chuchu Fan and Sayan Mitra. Bounded verification with on-the-fly discrepancy computation. *13th International Symposium on Automated*

Technology for Verification and Analysis, AVTA'15, Shanghai, China, 2015.

Chuchu Fan, Bolun Qi, Sayan Mitra, Mahesh Viswanathan, and Parasara Sridhar Duggirala. Automatic reachability analysis for nonlinear hybrid models with c2e2. In *International Conference on Computer Aided Verification*, pages 531–538. Springer, 2016.

Chuchu Fan, James Kapinski, Xiaoqing Jin, and Sayan Mitra. Simulation-driven reachability using matrix measures. *ACM Trans. Embedded Comput. Syst.*, 17(1):21:1–21:28, 2018a.

Chuchu Fan, Yu Meng, Jürgen Maier, Ezio Bartocci, Sayan Mitra, and Ulrich Schmid. Verifying nonlinear analog and mixed-signal circuits with inputs. In *ADHS*, volume 51 of *IFAC-PapersOnLine*, pages 241–246. Elsevier, 2018b.

Zhenqi Huang, Chuchu Fan, Alexandru Mereacre, Sayan Mitra, and Marta Z. Kwiatkowska. Invariant verification of nonlinear hybrid automata networks of cardiac cells. In *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*, pages 373–390. Springer, 2014. ISBN 978-3-319-08866-2.

Zhenqi Huang, Chuchu Fan, Alexandru Mereacre, Sayan Mitra, and Marta Z. Kwiatkowska. Simulation-based verification of cardiac pacemakers with guaranteed coverage. *IEEE Design & Test*, 32(5): 27–34, 2015.

Ned Nedialkov. VNODE-LP: Validated solutions for initial value problem for ODEs. Technical report, McMaster University, 2006.

Karthik Manamcheri Sukumar and Sayan Mitra. A step towards verification and synthesis from simulink/stateflow models. In *Tools paper in Hybrid Systems: Computation and Control (HSCC 2011)*, 2011.