Verisig: A tool for verifying safety properties of closed-loop systems with neural network components

Radoslav Ivanov, Taylor J. Carpenter, James Weimer, Rajeev Alur, George J. Pappas, Insup Lee
{rivanov, carptj, weimerj, alur, pappasg, lee}@seas.upenn.edu
University of Pennsylvania

0 Quick Start

This section lists the steps required to install Verisig and run an example. The installation procedure has been tested on **Ubuntu 16.04** and assumes **Java 1.8** has already been installed.

1. Install Flow* libraries

```
$ apt install libgmp3-dev libmpfr-dev libmpfr-doc libmpfr4 libmpfr4-dbg
   ↪ gsl-bin libgsl0-dev bison flex gnuplot-x11 libglpk-dev
```

2. Compile Flow*

```
$ cd flowstar; make; cd ..
```

3. Compile Verisig

```
$ cd verisig-src; ./gradlew installDist; cd ..
```

4. Run Mountain Car example

```
$ ./verisig --flowstar-cmd=./flowstar/flowstar examples/mountain_car/MC.
   ↪ xml examples/mountain_car/sig16x16.yml
```

At this stage, Verisig should compose the SpaceEx model for the Mountain Car (MC.xml) and the deep neural network (sig16x16.yml) and run the resulting model through Flow* with the initial conditions and safety property found in MC.yml. The computation should complete with a **SAFE** result.

1 Introduction

This document describes Verisig,[1] a tool for verifying safety properties of closed-loop hybrid systems with deep neural network (DNN) components. Verisig supports sigmoid- and tanh-based networks and exploits the fact that the sigmoid (and the tanh) is the solution to a quadratic differential equation, which allows us to transform the neural network into an equivalent hybrid system. By composing the network's hybrid system with the plant's, Verisig transforms the problem into a

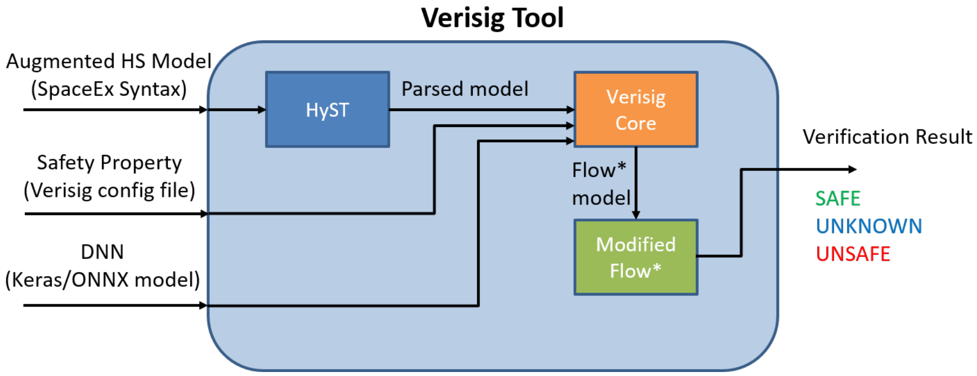

Figure 1: Overview of Verisig. Verisig takes three inputs: 1) a SpaceEx model description of the plant and its internation with the DNN; a 2) safety property to be verified; 3) the DNN description (possibly given as a Keras/ONNX model).

hybrid system verification problem which can be solved using state-of-the-art reachability tools. Specifically, Verisig uses Flow* [2] to solve the resulting hybrid system reachability problem.

The toolchain used in Verisig is illustrated in Figure 1. Verisig takes as input three pieces: 1) a description of the closed-loop system (namely, the plant's hybrid system and its interaction with the DNN) given as a SpaceEx model; 2) the safety property to be verified, given in a configuration file; 3) the DNN description (Verisig currently supports a few options, as described below). We now briefly comment on each of these pieces.

1.1 SpaceEx Model

The SpaceEx EModel ditor is a standard modeling language tool that was developed as part of the SpaceEx tool for verifying safety properties of linear hybrid systems [3]. The editor supports general non-linear hybrid system models and can be used to model networked systems as well, where each component can be described separately. In order to allow for the presence of the DNN in the model, we use a dedicated mode called "DNN" that is an abstraction for the DNN hybrid system. Having a dedicated mode serves two purposes: 1) the user can explicitly specify how the DNN interacts with the plant (e.g., a time-triggered controller) and can also capture DNN-specific operations (e.g., input normalization); 2) the user could change the DNN without modifying the SpaceEx model.

In order to parse the SpaceEx model, we use HyST [1], which is a general hybrid system translation and analysis tool. We added a "Verisig Printer" class to HyST that will be integrated with the main HyST repository. Using HyST not only eliminates the need to write our own SpaceEx XML parser but also means that we could use multiple useful features in HyST such as simple model checking (for debugging the model encoding) as well as model transformation passes (e.g., fill in default values for missing dynamics in a given mode).

1.2 Configuration File

The configuration file specifies the closed-loop safety property to be verified. Specifically, the property consists of initial conditions on plant states as well as an unsafe set (given as a mode and

[1] https://verisig.org

conditions on the continuous variables in that mode). Currently, the configuration file is specified in a YAML format, as shown in the examples.

1.3 DNN Description

Verisig requires the DNN to be described in a YAML format as well. YAML files are easily generated from Java or Python data structures. In the *utils* folder, we provide a few Python scripts that convert popular DNN training formats, such as Keras and ONNX models, into the required YAML format. Note that these scripts assume the DNN is a feedforward DNN – if the user would like to use a different class, such as recurrent DNNs, then they could use the example scripts as a starting point to building a similar script for their desired DNN class.

1.4 Modified Flow*

Note that we have added some modifications to Flow* in order to optimize the neural network verification part. Specifically, we avoid the sigmoid integration described in the original paper [4] but rather obtain Taylor Models analytically. As a result, Verisig users need to also use the modified version of Flow* provided with Verisig.

2 Installation

This section describes the steps required to install Verisig and related components on a computer. The installation procedure primarily consists of installing required libraries and compiling software packages. Note that it is not required for all three software packages (SpaceEx Model Editor, Flow*, and Verisig) to be installed on the same computer; each package is usable in isolation.

2.1 SpaceEx Model Editor (Optional)

The SpaceEx Model Editor is not required for the operation of Verisig, but it does simplify the process of creating and editing the SpaceEx XML model files. If the user chooses to skip the SpaceEx Model Editor, the user may instead modify the SpaceEx XML model files with a text editor.

The SpaceEx Model Editor is distributed as an executable JAR file, so it can be run on any computer that supports Java. The following procedure can be used to install the program:

1. Download and install Java 1.8+ : Instructions can be found at https://java.com/en/download/help/download_options.xml

2. Download the SpaceEx Model Editor JAR : Found on the SpaceEx website under the **SpaceEx Model Editor** section

2.2 Flow*

The modified version of Flow* that is used by Verisig is distributed in the Verisig release package. It is contained in the *flowstar* folder. Installing Flow* consists of installing the required libraries, compiling Flow*, and optionally adding the `flowstar` executable to the PATH environment variable for ease of use.

Flow* requires a MAKE environment and libraries that are only readily available on Linux. As such, Linux is currently the only supported operating system for Flow*. We have verified compatibility with Ubuntu 16.04, but other distributions should work as well. Flow* requires the following packages: **gmplib**, **mpfr**, **gnu gsl**, **bison**, **flex**, **gnuplot**, and **glpk**. The following procedure can be used to install the program:

1. Install required libraries :
   - `$ apt install libgmp3-dev`
   - `$ apt install libmpfr-dev libmpfr-doc libmpfr4 libmpfr4-dbg`
   - `$ apt install gsl-bin libgsl0-dev`
   - `$ apt install bison`
   - `$ apt install flex`
   - `$ apt install gnuplot-x11`
   - `$ apt install libglpk-dev`
2. From the *flowstar* folder, run the make command: `$ cd flowstar; make`
3. (Optional) Add the `flowstar` binary created in the previous step to the PATH environment variable, either through symlinking or by modifying the environment. If this step is omitted. the user must explicitly configure Verisig with the path to `flowstar`.

2.3 Verisig

The main Verisig executable is a Java application, so it can be run on any computer that supports Java. The source for the main Verisig application is included in the *verisig-src* folder. Installing Verisig consists of compiling the Java and creating run scripts using Gradle. The following procedure can be used to install the program:

1. Download and install Java 1.8+ : Instructions can be found at https://java.com/en/download/help/download_options.xml
2. From the *verisig-src* folder, run the `installDist` task of the Gradle wrapper :
   `$ cd verisig-src; ./gradlew installDist`

After installing from the Gradle wrapper, the `verisig` executable in the root folder of the distribution can be used to execute Verisig. Note: under the Windows operating system, the user will have to call `./gradlew.bat`, and will have to manually call the `verisig.bat` file located in *verisig-src/build/install/verisig/bin.*

3 Usage

This section describes how to use Verisig to verify safety properties for a closed-loop hybrid system. For the purposes of this guide, it is assumed that the user has already trained a DNN to use in the verification; the training of such a DNN is outside the scope of this tool.

3.1 Workflow

The general workflow for verifying a system with Verisig consists of three main steps: building a SpaceEx model, composing the hybrid system model with the DNN using Verisig, and running the composed model through Flow*. A description of the workflow steps follows:

3.1.1 Building a SpaceEx Model

Verifying a hybrid system first starts with creating a model of the hybrid system. The model format that is currently accepted by Verisig is the SpaceEx XML format. The method in which the SpaceEx model file is created, whether manually in a text-editor or through the use of the SpaceEx Model Editor, is irrelevant. Details on the format of SpaceEx XML and instructions on how to use the SpaceEx Model Editor can be found on the http://spaceex.imag.fr/. When creating the hybrid system model, the user should create a special mode named **DNN**, which will be used as a placeholder for the DNN. This mode will later be replaced with the hybrid system representation of the DNN, with the incoming transitions leading to the first mode in the DNN and the outgoing transitions coming from the last mode in the DNN. Inputs to the DNN are specified using special '`_f`' states, where '`_f1`' corresponds to the first input to the DNN, '`_f2`' the second input, etc. as specified by the order of inputs in the DNN file. The output of the DNN is retrieved using the '`_f`' states as well, after the DNN has run, where '`_f1`' corresponds to the first output of the DNN. It is common to set the inputs on the incoming reset to the **DNN** mode and retrieve the outputs on the outgoing reset of the **DNN** mode. The special **DNN** cannot be the initial mode of the system, as it will be replaced by multiple modes in the composition process, and thus will not specifically exist in the final system representation. If the user needs the functionality of starting with the DNN, a dummy mode must be created as the initial mode that transitions to **DNN**. ALso note that any dynamics or invariants specified on **DNN** will be ignored.

A SpaceEx XML file must be accompanied by a SpaceEx configuration (`.cfg`) file. SpaceEx is being used as a model format, not for model execution, so this configuration can be empty. In particular, it should be noted that any initial conditions that are specified in this configuration file are ignored.

Verisig currently does not support all of the functionality provided by SpaceEx. In particular, we currently require a flat hybrid system model – i.e. we do not support the networks of components that SpaceEx can represent. In the event that a SpaceEx model contains multiple components, only the first component in the file will be used, unless the `system` property is specified in the SpaceEx configuration file[2].

3.1.2 DNN YAML File Creation

Verisig currently supports DNNs specified in a custom YAML format which specifies the weights, biases, and activation functions of the different layers. We currently provide two Python scripts (h5_to_yaml.py, onnx_to_yaml.py) under *utils* that can be used to convert DNNs from the Keras h5 format or ONNX format to the appropriate YAML form[3]. If the trained network is in a different format, the user will have to create their own conversion script; the provided Python scripts should be a good starting point.

3.1.3 Verisig Configuration YAML File Creation

Verisig uses the YAML file format for its configuration values. The configuration file supports a variety of settings, all but two of which are optional. One of the values that is required to be contained within the configuration is the initial set. This is specified using the `init` key, under which an initial mode and a list of initial states are specified. The second value that is required is the unsafe set. This is specified using the `unsafe` key, under which is a list of modes and related

[2] We found it useful to create a network system with only a single component so that we could utilize constants rather than hard-coding values

[3] PyYaml is required for these scripts to work and can be installed through *pip*

state propositions. All of the state propositions within a list are taken as an intersection. A single mode can be included multiple times with different state propositions to form a union.

The rest of the configuration values are settings that are passed to the resulting flow* model. The settings names match the names used in flow*. Currently Verisig does not support the `fixed step` or `adaptive order` options [4]. For a complete reference as to the configuration format, take a look at the example configuration provided with the distribution.

3.1.4 Running Verisig

Once all of the appropriate files have been created, Verisig can be run. Verisig can be run either with Flow*, in which case Verisig creates the composed hybrid system model and runs Flow*, or without Flow*, in which case Verisig simply creates and outputs the composed Flow* model file.

With Flow* From the top-level folder of the distribution, the following command can be used to run Verisig:

```
$ ./verisig -sc <SpaceEx config> -vc <Verisig config> <SpaceEx XML> <DNN yaml>
```

Where `SpaceEx config` is the path to the SpaceEx cfg configuration file, `Verisig config` is the path to the Verisig YAML configuration file, `SpaceEx XML` is the path to the SpaceEx XML model file, and `DNN yaml` is the path to the DNN YAML file. If the Flow* command was not added to the `PATH` environment variable, the path to the Flow* executable can be specified using the `-flowstar-command` option.

Without Flow* From the top-level folder of the distribution, the following command can be used to run Verisig and output the model file (which can then be run at a different time and/or on a different computer):

```
$ ./verisig -sc <SpaceEx config> -vc <Verisig config> -o -nf <SpaceEx XML> <DNN
    ↪ yaml>
```

Where `SpaceEx config` is the path to the SpaceEx cfg configuration file, `Verisig config` is the path to the Verisig YAML configuration file, `SpaceEx XML` is the path to the SpaceEx XML model file, and `DNN yaml` is the path to the DNN YAML file. The model file will be created using the same name as the SpaceEx XML file only with a `.model` file extension.

A full description of the supported command line arguments can be found in the README file included with the distribution.

3.2 Setting the bounds on time and number of jumps

When verifying a given system, users need to specify bounds on the simulation time and number of (hybrid) jumps. These bounds are specified in the configuration file and are then passed on to Flow*. Since the DNN is part of the hybrid system that is sent to Flow*, the user needs to account for the DNN when computing the bounds. In the current implementation, the DNN does not require the passage of time (note that this is different from the original Verisig paper [4]); the DNN does require one jump per layer for the linear multiplication and one jump per layer for the non-linear activation function. For example, a DNN with two hidden layers and a linear last layer will go through five jumps.

[4] If required, these settings can be directly modified in the resulting Flow* model file

4 Tips & Tricks

There are many tricks that one could use in order to drastically improve the performance of Verisig. Since Verisig is based on Flow*, most of these tricks are related to different aspects of the Flow* implementation.

4.1 Default Values

Verisig provides two forms of value defaulting which can be useful when modeling large systems. First, it is assumed that if a state does not have any dynamics specified in a mode, the intention is for the value to not change. Therefore the user needs only to specify non-zero dynamics. Second, it is assumed that states with unspecified initial conditions are expected to have an initial state of 0.

4.2 Printing

The modified Flow* provides additional functionality for printing of the current system state during the verification process. If the name of a mode begins with `preprint_`, Flow* will output the current interval bounds for the states before the resets in incoming transitions are performed. If the name of a mode instead begins with `postprint_`, Flow* will output the current interval bounds for the states after the resets are performed.

4.3 Non-polynomial resets

The original version of Flow* allows only polynomial functions in resets on mode transitions. However, we encountered many examples where other resets were needed. That is why we added limited support (based on analytic Taylor Model approximations) for other non-linear functions as well, such as sine, cosine, tan, arctan, secant and division. Currently, the above resets can be performed one at a time, i.e., a user cannot perform both a cosine and a sine in the same reset (of course, they could be performed in consecutive resets without allowing time to pass). This functionality is currently implemented through changing a given mode's name. Specifically, if the user would like to perform the reset $y := f(x)$, then they need to specify a mode with the following name:

`_[function_name]_[y]_[x]_[id]`

where `[function_name]` $\in \{sin, cos, tan, arc, sec, div\}$[5] and `[id]` is an integer used to ensure that the mode name is unique. Note that the underscore "_" is used as a delimiter, so please avoid using underscores in variable and mode names. Finally, note that obtaining the Taylor Series for $arctan(x)$ is only possible if $-1 \leq x \leq 1$. If $|x| > 1$, then the user can use the following identity: $arctan(1/x) = \pi/2 - arctan(x)$.

4.4 Neural Network Architecture

Since Flow* makes heavy use of interval analysis, big DNN weights result in large approximation error. Thus, Verisig performs much better on DNNs with relatively small weights. For DNNs with big weights, the user might need to subdivide the initial space into multiple subsets. If users have the option of training the DNN, we recommend training with norm constraints on the weights (which is common practice when training DNNs).

[5] Note that div performs the reset $y := 1/x$.

4.5 Tips for keeping the approximation error small

The following is a short list of observations that might help users keep the approximation error small:

- For most problems, the initial set has to be subdivided into smaller subsets so as to keep the error small. While there is no generally best way of performing this refinement, as a rule of thumb, higher order terms (e.g., acceleration, velocity) are more sensitive and require more fine-grained refinement.
- If a guard contains more than one variable, it helps to precompute the entire guard expression first so as to turn it into a single variable expression. For example, a guard of the form $x+y > 0$ is much more efficiently computed in Flow* if there is a reset $z := x + y$ first, followed by a guard $z > 0$. This behavior is due to the interval analysis, as combined with Taylor Models, used in Flow*.
- Reducing the size of the initial set is the single most effective way of reducing the approximation error.
- Reducing the Flow* step size is generally much more effective than increasing the Taylor model order.

5 Example

There are two examples distributed with the current release:[6]

- Mountain Car – Mountain Car is a benchmark reinforcement learning example, where the task is to train a DNN controller to drive an underpowered car up a hill.
- Quadrotor MPC – in this problem, the task is to train a DNN to approximate a model predictive controller (MPC) so that the DNN can be used instead of the MPC online.

We illustrate Verisig on the Mountain Car reinforcement learning example. There are four files that go with this (and any other) example:

- *MC.xml* – this is the SpaceEx XML model of the plant dynamics and its interaction with the DNN. It contains a “base component” that is the plant dynamics and a “network component” that is used to provide values for the constants in the plant dynamics.
- *MC.cfg* – this is the SpaceEx configuration file. We are currently using this file to specify that there is a network component that contains the constant values.
- *MC.yml* – this is the Verisig configuration file in YAML format. This file specifies the safety property to be verified, plus optional Flow* settings (such as step size and Taylor model order).
- *sig16x16.yml* – this is the DNN encoding in a YAML format.

[6] Both examples are explained in the original Verisig paper [4]

We first describe the SpaceEx model. The car dynamics model is given in discrete time as follows:

$$
\begin{aligned}
p_{k+1} &= p_k + v_k \\
v_{k+1} &= v_k + 0.0015u_k - 0.0025 * cos(3p_k),
\end{aligned}
$$

where $u_k \in [-1, 1]$ is the controller's input, and p_k and v_k are the car's position and velocity, respectively, with $p_0 \in [-0.6, -0.4]$ and $v_0 = 0$. Note that v_k is constrained to be within $[-0.07, 0.07]$ and p_k is constrained to be within $[-1.2, 0.6]$.

The discrete-time nature of the model requires a non-standard hybrid system encoding. Effectively, the model consists only of resets between modes and no dynamics within modes. In order to enforce that no time elapses inside modes, we introduce a special *clock* variable with derivative 1 and invariant $clock \leq 0$.

Since Flow* does not allow cosine resets (but allows cosine in continuous dynamics), we use a dedicated mode to compute the cosine. For example, to compute $y = cos(x)$, one could first perform a reset $y := 0$, and then set the derivative of y to be $\dot{y} = cos(x)$ and simulate that mode for 1s (without changing x). Note that this example could also be encoded with the special encoding described in Section 4.3; however, we keep this workaround in this example in order to be consistent with the encoding used in the original Verisig paper [4].

The neural network in this example is a two-layer (with 16 neurons per layer) sigmoid-based DNN with a two inputs, p_k and v_k, and one output, u_k. The last layer also has a tanh activation in order to bound the control to be between -1 and 1. The network was trained in Keras (using an actor/critic approach) and was converted to YAML format using the *h5_to_yaml.py* script described in Section 3.1.2.

The safety property in this example is in fact a bounded liveness property. We verify that the final reward is at least 90 and that the car goes to the goal mode within 115 steps (the bound on the steps was established by simulating the system). The reward was encoded as a separate state with dynamics[7]

$$reward_{k+1} = reward_k - 0.1 * u_k^2,$$

where a reward of 100 is added upon reaching the goal mode. Note that the initial condition was subdivided into multiple intervals so as to keep the approximation error small enough to verify the property. To replicate the verification results, the user need simply run the *multi_runner.py* script in the *mountain_car* folder. Finally, note that we used a 3rd order Taylor model and an adaptive time step size with a maximum step size of 0.1s (this is the default step size used by Verisig if no step size is specified in the configuration file).

References

[1] S. Bak, S. Bogomolov, and T. T. Johnson. Hyst: a source transformation and translation tool for hybrid automaton models. In *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control*, pages 128–133. ACM, 2015.

[2] X. Chen, E. Ábrahám, and S. Sankaranarayanan. Flow*: An analyzer for non-linear hybrid systems. In *International Conference on Computer Aided Verification*, pages 258–263. Springer, 2013.

[7] In the encoding, we use the special DNN state '`_f1`' to encode the DNN's output u_k.

[3] 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 S. Q. Ganesh Gopalakrishnan, editor, *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, LNCS. Springer, 2011.

[4] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee. Verisig: verifying safety properties of hybrid systems with neural network controllers. *Proceedings of the 22nd International Conference on Hybrid Systems: Computation and Control*, 2019.