CAISAR Documentation

Release 0.1.1

The CAISAR Development Team

Jan 05, 2023

CONTENTS

CAISAR

Authors Michele Alberti, François Bobot, Julien Girard

Version 0.1, December 2022

This work has been partly supported by the Confiance.ai program.

CHAPTER

ONE

FOREWORD

CAISAR is a platform for the Characterization of Artificial Intelligence Safety And Robustness. It supports an expressive modelling language that allows the specification of a wide variety of properties to enforce on a machine learning system, and relies on external provers to discharge verification conditions.

1.1 Overall Design

CAISAR uses the Why3 platform, as a library, to provide both the specification language, called WhyML, and the infrastructure needed to interoperate with external provers. These are generally called *provers*, although some of them do not directly provide a logical proof.

The supported provers are the following:

- PyRAT, a neural network verifier based on abstract interpretation,
- Marabou, a Satisfiability Modulo Theory (SMT) solver specialized in neural network verification,
- SAVer, a support vector machine verifier based on abstract interpretation,
- nnenum a neural network verifier that combines abstract interpretation and linear programming techniques,
- *Classic* SAT/SMT solvers that support the SMT-LIBv2 input language.

CAISAR aims to be component-agnostic: it can model and assess the trustworthiness of artificial intelligence system that potentially mixes both symbolic- and data-driven approaches. It supports the verification of properties for multiple machine learning models. Neural Network (NN) are handled through the ONNX exchange format. Support Vector Machines (SVM) are supported through an ad-hoc CSV format.

The motivations and design ideas behind CAISAR have been presented at the workshop AISafety 2022 [GABCL2022].

1.2 Availability

The CAISAR platform page is https://git.frama-c.com/pub/caisar. The development version is available there, in source format, together with this documentation and several examples.

CAISAR is also distributed under the form of an opam package or a Docker image.

CAISAR is distributed as open source and freely available under the terms of the GNU LGPL v2.1.

See the file `README.md` for quick installation instructions, and section *Installation* of this document for more detailed instructions.

1.3 Contact

Report any bug to the CAISAR Bug Tracking System: https://git.frama-c.com/pub/caisar/issues.

1.4 Acknowledgements

We gratefully thank the people who contributed to CAISAR, directly or indirectly: Zakaria Chihani, Serge Durand, Tristan Le Gall, Augustin Lemesle, Aymeric Varasse.

CHAPTER

TWO

INSTALLATION

The latest release of CAISAR is available as an opam package or a Docker image.

The development version of CAISAR is available only by compiling the source code.

2.1 Install through Opam

Please note: CAISAR requires the OCaml package manager (Opam) v2.1 or higher, which is typically avaible in all major GNU/Linux distributions.

To install CAISAR via opam, do the following:

```
$ opam install caisar
```

2.2 Install through Docker

This method requires Docker to be installed in your system. A ready-to-use Docker image of CAISAR is available on Docker Hub. To retrieve it, do the following:

```
$ docker pull laiser/caisar:pub
```

Alternatively, a Docker image for CAISAR can be created locally by proceeding as follows:

```
$ git clone https://git.frama-c.com/pub/caisar
$ cd caisar
$ make docker
```

To run the CAISAR Docker image, do the following:

```
$ docker run -it laiser/caisar:pub sh
```

2.3 Compile from source

To build and install CAISAR, do the following:

```
$ git clone https://git.frama-c.com/pub/caisar
$ cd caisar
$ opam switch create --yes --no-install . 4.13.1
$ opam install . --deps-only --with-test --yes
$ make
$ make install
```

To run the tests:

```
$ make test
```

CHAPTER

THREE

USING CAISAR

3.1 Prover registration

CAISAR relies on external provers to work. You must install them first, then point CAISAR to their location. Please refer to each prover documentation to install them.

Provers tend to be complex programs with multiple options. Some combination of options may be suited for one verification problem, while inefficient for another. As they also have different interface, it is also necessary to register their call in a way CAISAR can understand. To do so, you must register provers inside the `config/caisar-detection-data.conf` file. Each supported prover is registered by specifying the following fields:

- `name`: the name under which CAISAR will know the prover
- `exec`: the prover's executable name
- `alternative` (optional): an alternative configuration for the prover. To use it with CAISAR, use the option `--prover-altern`. Useful when you want to use the same prover on different problems.
- `version_switch`: the command used to output the prover's version
- `version_regexp`: a regular expression parsing the output of `version_switch`
- `version_ok`: CAISAR supported version of the prover. Provers should only be used with their supported version.
- `command`: the actual command CAISAR will send to the prover. There are a few builtin expressions provided by CAISAR:
  - `%f`: the generated property file sent to the prover
  - `%t`: the timelimit value
  - `%{nnet-onnx}`: the name of the neural network file
- `driver`: location of the CAISAR driver for the prover, if any

Assuming you have installed a prover and you filled the `caisar-detection.conf` file, you can register the prover to CAISAR using the following command: `PATH=$PATH:/path/to/solver/executable caisar config -d`.

3.2 Property verification

A prominent use case of CAISAR is to model a specification for an artificial intelligence system, and to verify its validity for a given system.

The modelling uses WhyML, a typed first-order logic language. Example of WhyML are in the source code. You may also read the *CAISAR by Examples* section of this documentation to get a first grasp on using WhyML.

Provided a file *trust.why* containing a goal to verify, the command `caisar verify --prover=PROVER trust.why` will verify the goals using the specified prover. A list of supported provers is available in *Overall Design*. The prover must already be registered by CAISAR. Currently, only one prover can be selected at a time; future work will allow selecting multiple provers and orchestrate verification strategies. Internally, CAISAR will translate the goals into a form that is acceptable by the targeted prover, generating a file (the `%f` defined in the previous section).

3.3 Built-in properties

In addition to all the theories provided by Why3, CAISAR provide additional theories that model commonly desirable properties for machine learning programs. All those predicates are located in the file `stdlib/caisar.mlw`. Among them are theories to describe classification datasets, local and global robustness around a neighborhood.

CHAPTER

FOUR

CAISAR BY EXAMPLES

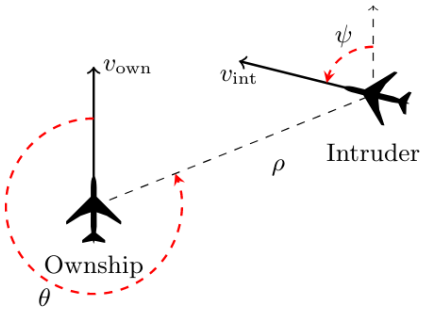
This page regroups some example use cases for CAISAR. All examples will describe the use case, the formalization using the WhyML specification language, and the CAISAR execution.

4.1 Functional properties of ACAS-Xu

ACAS-Xu stands for Aircraft Collision Avoidance System. Introduced for instance in [Manfredi2016], it is a specification of a program which aim to output signals for an aircraft in a situation where there is a potential for collision. In the rest of this tutorial, we will use the flavour ACAS-Xu defined in [Katz2017], where the authors aim to verify a neural network implementing part of the ACAS-Xu specification. Its low dimensionality and well defined semantics make it a *de facto* benchmark for machine learning verification.

4.1.1 Use case presentation

The system considers a 2D plane with two entities: the monitored airplane (the "ownship") and an incoming airplane (the "intruder").



In the original implementation, the system state has seven inputs:

- v_{own}: speed of ownship
- v_{int}: speed of intruder
- ρ: distance from ownship to intruder
- θ: angle to intruder relative to ownship heading direction
- ψ: heading angle of intruder relative to ownship heading direction
- τ: time until vertical separation
- a_{prev}: previous advisory

It has five outputs, that correspond to the different direction advisories the system can give:

- COC: Clear Of Conflict
- WL: Weak Left
- SL: Strong Left
- WR: Weak Right
- SR: Strong Right

In the original paper, the authors consider 45 neural networks, for several values of τ and a_{prev}, that operate on five inputs only while maintaining the same number of outputs. We will consider five-inputs networks in the remaining of this example.

Properties

There are several functional properties one may want to verify on this system, for instance:

- Guarantee that the system will never output COC advisory when the intruder is nearby,
- Guarantee that the system will never output an advisory that may result in a collision,
- Guarantee that the system will not output a strong advisory where a weak variant would be enough.

Authors of [Katz2017] propose ten properties to verify. We will reproduce the first and third properties here, and then show how to use CAISAR for verifying whether a given neural network respects them.

Property ϕ_1

- **Definition.** If the intruder is distant and is significantly slower than the ownship, the score of a COC advisory will always be below a certain fixed threshold.
- **Input constraints:**
  - $\rho \geq 55947.691$,
  - $v_{own} \geq 1145$,
  - $v_{int} \leq 60$.
- **Desired output property:**
  - $COC \leq 1500$.

Property ϕ_3

- **Definition.** If the intruder is directly ahead and is moving towards the ownship, the score for COC will not be minimal.
- **Input constraints:**
  - $1500 \leq \rho \leq 1800$,
  - $-0.06 \leq \theta \leq 0.06$,
  - $\psi \geq 3.10$,
  - $v_{own} \geq 980$,
  - $v_{int} \geq 960$.
- **Desired output property:**
  - COC is not the minimal score.

Modelling the problem using WhyML

The first step for verifying anything with CAISAR is to write a specification file that describe the problem to verify as a so-called *theory*. A theory can be seen as a namespace inside which are defined logical terms, formulas and verification goals. In particular, being based on the Why3 platform for deductive program verification, CAISAR supports the Why3 specification language WhyML, and inherits the Why3 standard library of logical theories (integer, float and real arithmetic, *etc.*) and basic programming data structures (arrays, queues, hash tables, *etc.*).

Let us try to model the property ϕ_1 defined earlier. We will call our theory `ACASXU_P1`.

We will need to write down some numerical values. As of now, CAISAR allows writing values using floating-point arithmetic only. Why3 defines a float type and the relevant arithmetic operations according to the IEEE floating-point standard in a theory, astutely called `ieee_float`. Specifically, we will import the `Float64` sub-theory, that defines everything we need for 64-bit precision floating-point numbers. We thus import it in our theory using the `use` keyword.

Our file looks like this so far:

```
theory ACASXU_P1
  use ieee_float.Float64
end
```

We would like to verify our propety given a certain neural network. To do this, CAISAR extends the Why3 standard library for recognizing neural networks in ONNX and NNet formats. Given a file of such formats, CAISAR internally builds a theory named as the original neural network file, that contains the sub-theories `AsTuple` and `AsArray` that provide logical symbols for describing the input-output interface of a neural network as tuples and array, respectively. We will only consider the `AsTuple` sub-theory for this tutorial.

In particular, the theory built by CAISAR is equivalent to the following WhyML file:

```
theory NeuralNetworkFilename
  theory AsTuple
    type t
    (* Tuple with as many elements as there are input *)
    function nn_apply (t,_,...)
    (* Tuple with as many elements as there are outputs *)
      : (t,_,...)
  end
  (* Other stuff *)
end
```

Note how the `AsTuple` theory defines the `nn_apply` function symbol that logically describes the input-output interface of a neural network using tuples. More importantly, CAISAR defines this function to take in input a tuple with as many elements as the inputs expected by the original neural network, and return a tuple with as many elements as the outputs provided by the original neural network.

As our neural network takes five inputs and provides five outputs, adding `use filename.AsTuple` to our theory will provide a `nn_apply` function symbol that takes a five-elements tuple as input, and provides a five-elements tuple as output. Assuming we have a neural network named `ACASXU_1_1.onnx`, the WhyML file looks like this:

```
theory ACASXU_P1
  use ACASXU_1_1.AsTuple
  use ieee_float.Float64
end
```

Now is the time to define our verification goal, that will call `P1_1_1` for property ϕ_1 on neural network $N_{1,1}$.

We first model the inputs of the neural network $\rho, \theta, \psi, v_{own}, v_{int}$ respectively as the floating-points constants x_i for $i \in [0..4]$. Moreover, we constraint these to the range of floating-point values each may take. According to the original

authors, values were normalized during the training of the network, and so we adapt the values they provide in their repository. Then, we define the result of the application of `net_apply` on the inputs by taking advantage of the WhyML pattern-matching, and define the output constraint we want to enforce on the floating-point constant y_0 that we use to model the advisory *COC*.

The final WhyML file looks like this:

```
theory ACASXU_P1
  use ACASXU_1_1.AsTuple
  use ieee_float.Float64

  goal P1_1_1: forall x0 x1 x2 x3 x4.
     (0.59999999999999997779553950749686919152736663818359375000000000000:t) .<= x0 .<=␣
↪(0.67985776870000003135885435767704620957374572753906250000000000000:t) ->
     (-0.5:t) .<= x1 .<= (0.5:t) ->
     (-0.5:t) .<= x2 .<= (0.5:t) ->
     (0.45000000000000001110223024625156540423631668090820312500000000000:t) .<= x3 .<=␣
↪(0.5:t) ->
     (-0.5:t) .<= x4 .<= (-0.
↪45000000000000001110223024625156540423631668090820312500000000000:t) ->
     let (y0, _, _, _, _) = nn_apply x0 x1 x2 x3 x4 in
     y0 .<= (3.99112564589999996300662132853176444768905639648437500000000000000:t)
end
```

This file is available, as is, under the `/examples/acasxu/` folder as property_1.why. The corresponding neural network in ONNX format is available under the `/examples/acasxu/nets/onnx/` folder as ACASXU_1_1.onnx.

Verifying the property with CAISAR

Once formalized, the specified property can be assessed by using CAISAR. We will use the *open-source* provers CAISAR supports for verifying properties of neural networks so to take advantage of the federating approach: whenever one prover cannot provide an answer, another may instead. In particular, we will use Marabou and nnenum.

Assuming the prover locations are available in `PATH`, the following are the CAISAR verification invocations using Marabou first and nnenum afterwords, for verifying the ACAS-Xu property ϕ_1:

```
$ caisar verify --prover Marabou -L examples/acasxu/nets/onnx --format whyml examples/
↪acasxu/property_1.why -t 10m
[caisar] Goal P1_1_1: Timeout
```

```
$ caisar verify --prover nnenum -L examples/acasxu/nets/onnx --format whyml examples/
↪acasxu/property_1.why -t 10m
[caisar] Goal P1_1_1: Valid
```

Note that the previous commands set the verification time limit to 10 minutes (*cf.* `-t` option), and the additional location `examples/acasxu/nets/onnx` (*cf.* `-L` option) for letting CAISAR correctly locate the neural network file `ACASXU_1_1.onnx` that is used by the `ACASXU_P1` theory in `property_1.why`.

Under the hood, CAISAR first translates each goal into a compatible form for the targeted provers, then calls the provers on them, and finally interprets and post-processes the prover results for displaying them in a coherent form to the user.

Marabou is not able to prove the property valid in the specified time limit, while nnenum does. In general, the result of a CAISAR verification is typically either `Valid`, `Invalid`, `Unknown` or `Timeout`. CAISAR may output `Failure` whenever the verification process fails for whatever reason (typically, a prover internal failure).

Using more advanced WhyML constructs

Let us model the ACAS-Xu property ϕ_3, and verify it for the neural networks $N_{1,1}$ and $N_{2,7}$ [Katz2017].

From the modelling standpoint, the main evident difference concerns the desired output property, meaining that *COC* should not be the minimal value. A straightforward way to express this property is that the corresponding floating-point constant y_0 is greater than or equal to at least one of the other five outputs. This can be formalized in first-order logic as a disjunction of clauses, that can be directly encoded into WhyML as follows:

```
y0 .>= y1 \/ y0 .>= y2 \/ y0 .>= y3 \/ y0 .>= y4
```

The delicate point is how to model the same property for two different neural networks. Of course, we could define a theory with two identical but distinct verification goals or two entirely distinct theories in a same WhyML file. However, these two solutions are not advisable in terms of clarity and maintainability.

Reassuringly enough, WhyML provides all necessary features to come up with a better solution. First, WhyML allows for naming used (sub-)theories in order to distinguish identical logic symbols coming from different theories. This is critical for identifying the correct `nn_apply` symbols in the two verification goals we will define. Second, WhyML allows for the hypotheses on the floating-point constants modelling the neural network inputs to be exported from the verification goal into the theory general context as axioms.

In the end, the WhyML file looks like this:

```
theory ACASXU_P3
  use ACASXU_1_1.AsTuple as N11
  use ACASXU_2_7.AsTuple as N27
  use ieee_float.Float64

  constant x0:t
  constant x1:t
  constant x2:t
  constant x3:t
  constant x4:t

  axiom H0:
     (-0.30353115609999997692725060005614068359136581420898437500000000000:t) .<= x0 .<=␣
↪(-0.29855281189999999247319806272571440786123275756835937500000000000:t)

  axiom H1:
     (-0.00954929659999999985720009476608538534492254257202148437500000000:t) .<= x1 .<=␣
↪(0.00954929659999999985720009476608538534492254257202148437500000000:t)

  axiom H2:
     (0.49338032360000000364763650395616423338651657104492187500000000000:t) .<= x2 .<=␣
↪(0.5:t)

  axiom H3:
     (0.29999999999999999888977697537484345957636833190917968750000000000:t) .<= x3 .<=␣
↪(0.5:t)

  axiom H4:
     (0.29999999999999999888977697537484345957636833190917968750000000000:t) .<= x4 .<=␣
↪(0.5:t)

  goal P3_1_1:
```

(continues on next page)

(continued from previous page)

```
    let (y0, y1, y2, y3, y4) = N11.nn_apply x0 x1 x2 x3 x4 in
    y0 .>= y1 \/ y0 .>= y2 \/ y0 .>= y3 \/ y0 .>= y4

  goal P3_2_7:
    let (y0, y1, y2, y3, y4) = N27.nn_apply x0 x1 x2 x3 x4 in
    y0 .>= y1 \/ y0 .>= y2 \/ y0 .>= y3 \/ y0 .>= y4
end
```

Note how the two verification goals `P3_1_1` and `P3_2_7` are clearly almost identical, but for the `nn_apply` logic symbol used, identifying respectively the `ACASXU_1_1.onnx` and `ACASXU_2_7.onnx` neural networks.

We can then verify the resulting verification problem as before:

```
$ caisar verify --prover Marabou -L examples/acasxu/nets/onnx --format whyml examples/
↪acasxu/property_3.why -t 10m
[caisar] Goal P3_1_1: Timeout
[caisar] Goal P3_2_7: Valid
```

```
$ caisar verify --prover nnenum -L examples/acasxu/nets/onnx --format whyml examples/
↪acasxu/property_3.why -t 10m
[caisar] Goal P3_1_1: Valid
[caisar] Goal P3_2_7: Valid
```

It is interesting to remark that, since Marabou does not support disjunctive formulas, CAISAR first splits a disjunctive goal formula into conjunctive sub-goals, then calls Marabou on each sub-goals, and finally post-processes the sub-results to provide the final result corresponding to the original goal formula.

4.2 (Local) Robustness of MNIST dataset

CAISAR provides a convenient way for verifying (local) robustness properties of neural networks on datasets, for classification problems only, in a specific `CSV` format. In particular, each of the `CSV` lines is interpreted as providing the classification label in the first column, and the dataset element features in the other columns.

We recall that a neural network is deemed robust on a dataset element whenever it classify with a same label all other elements being at an l_∞-distance of at most $\epsilon \geq 0$ from it. More in general, a neural network is deemed (locally) robust on a dataset whenever the former property is valid on all the dataset elements. The CAISAR standard library specifies such a property in terms of the predicate `robust`, which CAISAR implements as a builtin.

In the following, we will describe how to use CAISAR for verifying a neural network robust on (a fragment of) the MNIST dataset.

4.2.1 Use case presentation

MNIST is a dataset of handwritten digits normalized and centered to fit into grayscale images of 28×28 pixels, along with the classification labels [LiDeng2012]. Although it is mostly irrelevant as dataset for benchmarking machine learning models for computer vision tasks, MNIST is still valuable for assessing robustness properties by means of formal method tools.

CAISAR provides in mnist_test.csv a fragment (100 images) of the MNIST dataset under the `examples/mnist/csv` folder. Each line in this file represents an MNIST image: in particular, the first column represents the classification label, and the remaining 784 columns represent the greyscale value of the respective pixels.

Properties

Generally speaking, the property we are interested in verifying is the local robustness of a machine learning model on the elements of a set. That is, the model classifies all elements of a set being at an l_∞-distance of at most $\epsilon \geq 0$ with a same label. A general formulation of this latter states that, given a classifier C, a set X, and some perturbation $\epsilon \geq 0$, it must hold that $\forall x, x' \in X.\ \|x - x'\|_\infty \leq \epsilon \Rightarrow C(x) = C(x')$.

Since we actually deal with a *dataset* of finite elements for which we also know the expected labels, we will instead verify a slightly different property: given a classifier C, an element $x \in X$ such that $C(x) = y$, and some perturbation $\epsilon \geq 0$, it must hold that $\forall x'.\ \|x - x'\|_\infty \leq \epsilon \Rightarrow C(x) = y = C(x')$. Obviously, such a property must be verified for all elements of a dataset.

Modelling the problem using WhyML

As described for the example on *Functional properties of ACAS-Xu*, we first need to write a specification file containing a WhyML theory to describe the verification problem. In principle, we need to formalize the local robustness property as well as the notions of classifier and dataset.

The CAISAR standard library caisar.mlw provides some utilities for dealing with verification problems about classification datasets. Of particular interest for us is the `robust` predicate, defined in the theory `DataSetProps` as follows:

```
predicate robust (m: model) (d: dataset) (eps: t) =
  forall i: int. 0 <= i < d.data.length -> robust_at m d.data[i] eps
```

Note that the predicate is defined over a `model`, a `dataset` and a floating-point value `eps`. The latter determines the perturbation ϵ. The other two are custom WhyML types that respectively formalize the notions of classifier and dataset in CAISAR. These types are respectively defined in the `Model` and `DataSetClassification` theories.

Moreover, it is defined in terms of the predicate `robust_at` that formalizes the local robustness property:

```
predicate robust_at (m: model) (d: datum) (eps: t) =
  forall x': features.
    let (x, _) = d in
    linfty_distance x x' eps ->
    predict m x = predict m x'
```

Note that a `datum` is a dataset element given as a pair of *features* and (classification) *label*. Morever, `linfty_distance` is a predicate that describes how two arrays of floating-point values (*i.e.* `features`) are considered close up-to a pertubation `eps`, while `predict` is a function that formalizes the execution of a model on some features to obtain the corresponding classification label.

In order to use the `robust` predicate in our WhyML specification, we need values of types `model` and `dataset`. For the former, CAISAR makes available the constant `model` upon interpreting the `AsArray` sub-theory that is built by the extension of the Why3 standard library for recognizing neural network ONNX and NNet formats. For the latter, the CAISAR standard library provides the constant `dataset` in `DataSetClassification` that will be interpreted as the actual dataset the user needs to provide via the command-line interface when launching a CAISAR verification.

Assuming we have a neural network named `MNIST_256_2.onnx` for MNIST classification, the final WhyML file for specifying its local robustness on a dataset, with each element's feature pertubed by 1%, looks like this:

```
theory MNIST
  use MNIST_256_2.AsArray
  use ieee_float.Float64
  use caisar.DataSetClassification
  use caisar.DataSetProps
```

(continues on next page)

(continued from previous page)

```
  goal robustness:
   let normalized_dataset = min_max_scale true (0.0:t) (1.0:t) dataset in
   robust model normalized_dataset (0.
↪0100000000000000020816681711721685132943093776702880859375000000:t)
end
```

Note the presence of the `min_max_scale` function defined in `DataSetClassification` for normalizing all feature values in $[0, 1]$. Besides classic *Min-Max scaling*, CAISAR also provides `z_norm` function for *Z-normalization*.

This file is available, as is, under the `/examples/mnist/` folder as property.why. The corresponding neural network in ONNX format is available under the `/examples/mnist/nets/` folder as MNIST_256_2.onnx.

Verifying the property with CAISAR

Now we may verify whether the previous robustness specification holds on the MNIST fragment `mnist_test.csv` by means of the nnenum prover. This can be done via CAISAR as follows:

```
$ caisar verify --prover nnenum -L examples/mnist/nets --format whyml --dataset=examples/
↪mnist/csv/mnist_test.csv examples/mnist/property.why
[caisar] Goal robustness: Invalid
```

The result tells us that there exists at least one image in `mnist_test.csv` for which nnenum is sure that the model `MNIST_256_2.onnx` is not robust with respect to 1% perturbation. At the moment, CAISAR is not able to tell which are the images in the dataset that cause such result.

BIBLIOGRAPHY

[GABCL2022] Girard-Satabin, J., Alberti, M., Bobot, F., Chihani, Z., Lemesle, A., *CAISAR: A platform for Characterizing Artificial Intelligence Safety and Robustness*, Proceedings of the Workshop on Artificial Intelligence Safety 2022 (AISafety 2022)

[Manfredi2016] G. Manfredi and Y. Jestin, *An introduction to ACAS Xu and the challenges ahead*, 2016 IEEE/AIAA 35th Digital Avionics Systems Conference (DASC), 2016, pp. 1-9, doi: 10.1109/DASC.2016.7778055

[Katz2017] Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J. (2017). *Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks.* CAV 2017, doi: 10.1007/978-3-319-63387-9_5

[LiDeng2012] Li Deng, *The MNIST Database of Handwritten Digit Images for Machine Learning Research*, IEEE Signal Process. Mag., 2012, pp. 141-142, doi: 10.1109/MSP.2012.2211477