Synthesizing abstract transformer

PANKAJ KUMAR KALITA, Indian Institute of Technology Kanpur, India
SUJIT KUMAR MUDULI, Indian Institute of Technology Kanpur, India
LORIS D'ANTONI, University of Wisconsin–Madison, USA
THOMAS REPS, University of Wisconsin–Madison, USA
SUBHAJIT ROY, Indian Institute of Technology Kanpur, India

Contents

1 GETTING STARTED

This section provides instructions for setting up the artifact and running the evaluation. The artifact is packaged as a docker image and Section 1.1 provides steps to setting up docker-engine in your system. Your system should meet the following minimum requirements for artifact evaluation.

- OS : Ubuntu 18.04.
- RAM : 32GB or higher is recommended.
- CPU (cores): CPU with eight or more logical cores. (Intel i7 8th Gen or higher.)
- Secondary Memory: We recommend at least 20 GB of free space for all the experimental evaluations to be conducted.
- Internet Access: We require a working internet connection to download the artifact.

Extended version of the paper is available here [1]

1.1 Starting with the Docker

Use the following instructions to install Docker in Ubuntu. One can skip the following instructions to install docker if it is already installed.

```
$ sudo apt-get install docker.io
```

One can verify whether docker has been installed or not.

```
$ sudo docker run hello-world
```

1.1.1 Running Docker from Zenodo. Download the tar file containing docker image from the Zenodo. Now open terminal from the downloaded folder and type following command to load the image.

```
$ sudo docker load < amurth_oopsla2022.tar.gz
```

This will load the docker image. Use following code to run the image.

```
$ mkdir genResult
$ sudo docker run -it -v $(pwd)/genResult:/root/genResult amurth_image:V1.0
```

1.1.2 Docker pull from the Docker Hub. Please pull our docker image from the docker hub using the following code snippet.

```
$ sudo docker pull pkalita595/amurth_oopsla22
```

After completion of the docker download, we are ready to start the docker using the following command.

```
$ mkdir genResult
$ sudo docker run -it -v $(pwd)/genResult:/root/genResult pkalita595/amurth_oopsla22
```

Directory `genResult` can be used to store relevant logs/plots to be accessed from outside the container. After starting the docker, please change the working directory to `/root` using the following.

```
$ cd /root
```

1.2 Artifact structure

In `/root` directory of the artifact, there are three directories as shown below.

```
$ ls
 amurth genResult safe-alpha
```

There are two types of experiments that have been reported in the paper. They are described as follows.

(1) First, we synthesize different transformers for string, arithmetic, and logical operations, as shown in Sections 6.1 and 6.2 of the paper, using our tool Amurth. Implementation of Amurth is in the `amurth` directory.
(2) Second is the analysis of synthesized transformers of string domain in a string analyzer, i.e., SAFE_{str} in Section 6.1.3 in the paper. Implementation of the synthesized transformer in the string domain inside SAFE_{str} can be found in the `safe-alpha` directory. In `safe-alpha` directory, we have provided scripts to generate plots shown in Figure 10 from the paper, Figures 11, 12, 13 from the extended version [1].

1.3 Running a small set of benchmarks for Amurth

In this section, we will get start running Amurth to synthesize a few transformers. We have provided a script `run_mini_ex.sh` inside `/root/amurth/src` directory. Running this script will run `amurth.py` on the given subset of benchmarks. This script will take around 15-20 minutes to produce output. The reviewer can run this file using bash as shown below.

```
$ pwd
/root
$ cd amurth/src
$ bash run_mini_ex.sh
```

This command will print transformers of each benchmark while running, and at the end of execution, it will produce two tables, the same as shown in Table 2 and Table 3 from the paper. Produced output will have fewer entries as `run_mini_ex.sh` runs few benchmarks only. Details of each benchmark as shown in Table 3 and 5 from the extended version [1] can be found in `detailedLog` inside `amurth/log` directory. In Section 2.1.1 we have shown how to run a single benchmark.

1.4 Running a small set of benchmarks for SAFE_{str}

This section will show how to run SAFE_{str} for a small set of benchmarks. Implementation of SAFE_{str} is present in the `/root/safe-alpha` .

Please change the directory to `/root/safe-alpha` as shown below.

```
$ cd /root/safe-alpha/
```

We have provided a script `run_small_exp.sh` in the `safe-alpha/benchmarks` directory. Please run the following commands to generate the plots. This will take around 15-20 minutes to generate the plots.

```
$ cd benchmarks
$ rm -rf plots/*
$ bash run_small_exp.sh
```

This will make a job list of a few benchmarks using a combination of constant string and character inclusion domain (coci) and run SAFE_{str} for those benchmarks. In SAFE_{str}, id for constant string is `co` instead of `CS`. It will generate plots similar to Figure 10 of the paper inside the directory `/root/safe-alpha/benchmarks/plots` . For this case, plots are being stored in the directory `/root/safe-alpha/benchmarks/plots/coci` as the experiment was on constant string and character inclusion domains. One can copy the content of the `plots` to the directory `/root/genResult/` to access from outside of the docker.

1.5 Claims supported by this artifact

This artifact supports two claims in the paper

(1) AMURTH can synthesize each L-transformer for fixed-bitwidth interval and string domains within 2000s and can produce Table 2, 3 form the paper, Table 3, 5 from the extended version [1] and Figure 10 from the paper, Figures 11, 12, 13 from the extended version [1]. One can observe a more prolonged time taken by AMURTH in your machine due to the docker container.
(2) AMURTH generates sound and precise abstract transformers for different operations across various domains discussed in the paper.

2 STEP BY STEP INSTRUCTIONS

2.1 Experiment for transformer synthesis using AMURTH

2.1.1 Running a simple example. In this section we will see how to run a single benchmark using AMURTH. Please change the directory to `amurth`.

```
$ cd /root/amurth
```

In `config` directory we put `json` files for all the operations we have synthesized. For each domain and each operation we have provided a `json` file. All of our source code present in `src` directory. Please change the current directory to `src`.

```
$ cd src
```

We have provided the python script `run_single_ex.py` to run a single benchmark from the set of benchmarks present in `config` directory. This script takes two arguments, i.e., `<domain name>` and `<operation name>`. If we want to synthesize transformer for `absolute` (`abs`) operator in signed interval domain (SI) you will type as follows.

```
$ python run_single_ex.py SI abs
```

This will run `amurth.py` with the `json` file related to the `abs` operator for signed interval domain (SI). This will run for some time and in the end, will produce the transformer.

Following is the list of domains and operators AMURTH supports along with the IDs to pass to the `run_single_ex.py`. Please refer to Sections 6.1 & 6.2 and Table 1 in the paper for the details of the domains.

Domain name	$\mathcal{CS}$	$\mathcal{SS}_k$	$\mathcal{CI}$	$\mathcal{PS}$	$\mathcal{SH}$	$\mathcal{A}_{uintv}$	$\mathcal{A}_{sintv}$	$\mathcal{W}$
ID	CS	SSK	CI	PS	SH	UI	SI	WI

Following is the list of IDs of all the operations.

Arithmetic and logical	`add`	`sub`	`mul`	`and`	`or`	`xor`	`shl`	`lshr`	`ashr`
String	`concat`	`charAt`	`contains`	`toLower`	`toUpper`	`trim`			

AMURTH generates some temporary files while running. Sketch files generated during each iteration of precision check, MaxSAT synthesize can be found in `amurth/temp/sketch`. If reviewer runs experiment using `run_single_ex.py`, then complete log of the operation can be found in `amurth/temp/` directory. Naming convention of the log will be like `log_<operation name>.txt`.

Here <operation name> is the name of operation present in the `tosynthesize` field in the respective `json` file. *In case of failure please run again.*

2.1.2 Synthesizing transformer for all the operation and domains. We provide a script to run all the json files (benchmarks) which will synthesize the transformer for each operations.

```
$ cd /root/amurth/src
$ bash run_all.sh
```

This script will run for around 14 hours, and in the end, it will produce a table, as shown in Table 2 and Table 3 from the paper. `detailedLog` and log files generated during the run will be stored in `/root/amurth/log/` . Entries shown in Table 3 and 5 in the extended version [1] can be found in `detailedLog` .

We have also provided following additional scripts for different types of benchmarks.

- `run_string_domains.sh` : Runs all the string domains benchmarks.
- `run_unsigned_interval.sh` : Runs all the unsigned interval domain benchmarks.
- `run_wrapped_interval.sh` : Runs all the wrapped interval domain benchmarks.
- `run_all_interval.sh` : Runs all the interval domain benchmarks.

2.1.3 Understanding input and output of Amurth.

Understanding input. Amurth (`amurth/src/amurth.py`) takes a `json` file. `josn` file for addition in signed interval is shown below.

```
{
        "basepath": "/root/amurth/",
        "dsl": "interval/L_arithmetic.sk",
        "abstract_domain": ["interval/arithmetic.sk", "interval/plus.py",
                            ["absleft", "absright"]],
        "abstract_value": [ ["left", ["int", "0"]],
                ["right", ["int", "0"]]],
        "aux_fun": ["aux_function.c", "aux_function.sk"],
        "tosynthesize": { "add":["int", "int", "int"]}
}
```

Let us describe each element of the `json` file one by one.

- `basepath`: It is the basepath of the tool installed
- `dsl`: It shows the file present in `/root/amurth/dsl/` which contains the dsl of the given domain for the operation in sketch.
- `abstract_domain`: First element `interval/arithmetic.sk` contains the partial order of lattice, template of abstract transformers, different functions to define the domain etc. for sketch synthesis engine.
  `interval/plus.py` contains the bootstrapping example to start the synthesis.
  Files related to these field will be available in `/root/amurth/abstract_domain` directory.
  Last element `["absleft", "absright"]` shows the name of the transformer to be synthesized. Since we are currently talking about the interval domain, it needs two transformers for both left and right limits.
- `abstract_value`: Defines the abstract value and its type.
- `aux_fun`: Describes the file containing auxiliary functions required for synthesis.

- `tosynthesize`: This contains the function to be synthesized and its prototype. For example, here add takes two integer and returns an integer. Definition of these functions are present in `amurth/external_lib/logicalSpec*.sk`.

To try out new function, one need to change the name of the function in `tosynthesize` field with its prototype and need to add its definition in `amurth/external_lib/logicalSpec.sk` for interval and `amurth/external_lib/logicalSpecString.sk` for string domains. One also has to update DSL and domain defining files inside `amurth/abstract_domain` according to the requirements.

Understanding output. Let us now try to understand the output of the tool. Running AMURTH with a `json` file will produce the transformer in the end. For example, in the case of `add` we will get the following transformer.

```
int  absleft(int left1, int right1, int left2, int right2)
{
  int _out1;
  _out1 = left1 + left2;
  return _out1;
}
int  absright (int left1, int right1, int left2, int right2)
{
  int _out1;
  _out1 = right1 - (0 - r2);
  return _out1;
}
```

Now, `absleft` and `absright` are two transformer for both left and right limit of the signed interval domain. Both transformer takes four argument, i.e., `left1, right1, left2, right2`. `left1`, `right1` represents low and high limit of first argument of `add`, whereas `left2`, `right2` represents the second argument.

Encoding of string in AMURTH. In AMURTH we encode a string as an array of integers. For the case of `trim` operation we consider one of the number, i.e., 10 as space. `contains` in $\mathcal{CI}$ returns $\top$, $\bot$, 1 or 0. We used following encoding to distinguish between the return types.

```
#define FALSE 0
#define TRUE  1
#define TOP   2
#define BOT   3
```

2.2 Running experiment for SAFE_{str}

In this section, we will try to run an experiment to get plots, as shown in Figure 10 from the paper, Figures 11, 12, 13 from the extended version [1].

2.2.1 Plot for $\mathcal{CS} \times \mathcal{CI}$ (`coci`) as shown in Figure 9 in the paper. Please follow the instructions below to run the experiment in SAFE_{str} for $\mathcal{CS} \times \mathcal{CI}$ (`coci`) domain.

```
$ cd /root/safe-alpha/benchmarks
$ bash run_coci_exp.sh
```

Running `run_coci_exp.sh` will generate plot as shown in Figure 10 inside following directory.

```
/root/safe-alpha/benchmarks/plots/coci
```

This will take around 7 hours to completely execute the script.
Please copy the content of `/root/safe-alpha/benchmarks/plots/coci` to `/root/genResult` to view the figures/plots.

2.2.2 Running full experiment in SAFE$_{str}$ (For Figure 10 from the paper, Figures 11, 12, 13 from the extended version [1]). Please follow the instructions below to run the entire experiment in SAFE$_{\text{str}}$.

```
$ cd /root/safe-alpha/benchmarks
$ bash run_full_exp.sh
```

The script `run_full_exp.sh` will take around 30 hours. After completion of execution of this script, it will generate four directories inside `safe-alpha/benchmarks/plots/` . Each directory in `plots` , i.e., `plots/coci` , `plots/cops` , `plots/co` and `plots/ss` contains plots for Figure 10 from the paper, Figures 11, 12, 13 from the extended version [1], respectively. Each figure contains four plots, i.e., Fixpoint iteration, Time, Program States and Imprecision index. So each four directories will contain four figures containing the plots.

Benchmarks for SAFE$_{\text{str}}$ contains some `HTML` files and few `JavaScript` files. `run_full_exp.sh` takes huge amount of time as `JavaScript` benchmarks are costly. We have also provided a script, `run_html_exp.sh` inside `/root/safe-alpha/benchmarks` to only run HTML benchmarks.

2.3 Structure of the Amurth

Directory `/root/amurth` contains all the source code and benchmarks for synthesizing abstract transformers. Following is the detailed breakdown of the tool directory structure.

- `abstract_domain` : Contains files that defines the domains in sketch and files for bootstrapping examples.
- `aux_function` : Contains auxiliary functions required during synthesis.
- `config` : It contains `json` files for each operation in different domains. In case Amurth we describe each input to the tool with the help of `json` files. One can assume that each `json` file is one benchmark. `json` files for each domains are organised in the directories inside `config` .
- `dsl` : It contains DSL (domain-specific language) for each domains in separate directories.
- `external_lib` : Contains definitions of the concrete operators.
- `include` : Include files for sketch to run.
- `src` : This contains all the source code for Amurth. `amurth.py` is the main driver file to run Amurth.
- `temp` : Contains temporary files created during the running of the tool.

2.4 Structure of the SAFE$_{\text{str}}$

SAFE$_{\text{str}}$ repository is available at `/root/safe-alpha` in the docker container.

- String domain implementation can be found inside following directory.

  ```
  /src/main/scala/kr/ac/kaist/jsaf/analysis/typing/domain
  ```

- Benchmarks (`HTML, JavaScript`) can be found in `/root/safe-alpha/benchmarks` directory.

REFERENCES

[1] Pankaj Kumar Kalita, Sujit Kumar Muduli, Loris D'Antoni, Thomas Reps, and Subhajit Roy. 2021. Synthesizing Abstract Transformers. https://doi.org/10.48550/ARXIV.2105.00493