COORDINATION 2024: Artefact submission for the paper #8

This document specifies the instructions for the AEC of COORDINATION 2024 for the evaluation of our artefact submission. We set a `Docker` container for `TRAC` in order to simplify the work of the AEC (the `README` file at https://github.com/loctet/TRAC contains the instructions for the manual installation procedure).

Table of content

1. Installation

Follow the instructions at https://docs.docker.com/ to install `Docker` on your system.

To install and run TRAC using `Docker`:

1. Pull the `Docker` image:

   ```
   docker pull loctet/trac_dafsms:v1
   ```

2. Run the container:

   ```
   docker run -it loctet/trac_dafsms:v1
   ```

   (you might need to run the above commands as `root`). The former command downloads the `Docker` image of `TRAC` while the latter starts a container with an interactive terminal.

2. Reproducibility

2.1 How Table 1 has been created

We now describe how the information in Table 1 of our COORDINATION has
been determined.
We recall that Table 1 reports how our framework captures the features of the smart contracts in the Azure repository described at the following links:

- Hello Blockchain
- Simple Marketplace
- Basic Provenance
- Digital Locker
- Refrigerated Transportation
- Asset Transfer
- Room Thermostat
- Defective Component Counter
- Frequent Flyer Rewards Calculator.

For each smart contract, the table below reports

- where the features are met in the `Solidity` implementation in the Azure repository
- the lines in the DAFSM model where the feature is captured (if at all)

Example (link to .sol)	Line in Code for the feature	How TRAC handle it
Simple Marketplace	BI : Lines 21 & 44	BI: Line 1, 2 & 6

Example (link to .sol)	Line in Code for the feature	How TRAC handle it
Hello Blockchain	BI: Lines 19 & 39	Bi: Line 1 & 3
Bazaar	ICI: BazaarItem (Line 78) ItemList(Line 40) BI : BazaarItem (Line 76) ItemList(Line 33)	
Ping Pong	BI : Line 16 & 67 ICI : Lines 18, 29, 41, 47, 77, 82 & 88	
Defective Component Counter	BI: Line 17 PP: Line 15	BI: Line 1 PP: Line 1
Frequent Flyer Rewards Calculator	BI : Line 20 PP : Line 18	BI: Line 1 PP: Line 1
Room Thermostat	PP : Line 16	PP: Line 1
Asset Transfer	BI : Line 18, PP: Line 49 RR : Lines 97 & 171	BI: Lines 1& 3 PP: Lines 2& 7 RR:
Basic Provenance	BI: Line 19 PP: Line 17, 26 RR : Line 38, 51	BI: Line 1 PP: Line 1, 2, 3 RR:
Refrigerated Transportation	BI: Line 32 PP: Line 28, 89 RR : Line 118, 143 MRP : Lines 33, 34, 119 & 142	BI: Line 1 PP: Line 1, 5 & 9 RR: MRP:
Digital Locker	BI : Line 21 PP: Lines 19, 68 RR : Lines 102,126, 127, 139 & 149 MRP : Lines 76, 91	BI: Line 1 PP: Line 1 RR: MRP:

2.2. How to check the well-formedness of the Azure benchmarks

The DAFSM models for each smart contract but for `Bazaar` and `Ping Pong` of the Azure repository can be found in the directory `Examples/dafsms_txt/azure`.

To check a model with `TRAC`, navigate to the directory `src` and execute the `Main.py` as done with the `Docker` commands below on the `simple-marketplace` smart contract:

```
cd src
python3 Main.py --filetype txt "azure/simplemarket_place"
```

The latter command produces the following output

```
    --Parsing Txt to generate Json file

    Checking the well formness of the model----

    (!) Verdict: Well Formed
```

reporting that the DAFSMs for the `simplemarket_place` is well formed. For the other smart contracts it is enough to execute the python script on the corresponding DAFSM.

2.3. How to check the randomly generated models

The 135 randomly generated models used in last part of Section 4 of our paper are in `src/Examples/random_txt/tests_dafsm_1` splitted in subfolders each containing 5 DAFSMs and a `list_of_files_info.csv` file with metadata on the DAFSMs (we detail the metadata in section 4 below). Our performance analysis can be reproduced by executing the following commands in the `Docker`:

```
performancecd src
python3 Random_exec.py tests_dafsms_1 --number_runs_per_each 10 --number_test_per_cpu 5 --time_out 300000000000
```

Note that the results may vary due to different hardware/software configurations than those we used (cf. page 12 of the paper).
The latter command above specifies the target directory `tests_dafsms_1`, the number of repetitions for each experiment, the number of experiments analyzed by each cpu, and the time out in nanoseconds. While running the checks further `csv` files will be generated and finally merged into a single file called `src/Examples/random_txt/tests_dafsm_1/merged_list_of_files_info.csv`. Notice if the target directory in the command above is not changed, this `csv` file will be overwritten at each execution. The current content of the `csv` files when starting the `Docker` contains the values plotted in Figures 2 and 3 of our paper.

The plots can be obtained by executing

```
python3 ./plot_data.py examples_1 --file merged_list_of_files_info --field num_states,num_transitions,num_paths --pl_lines
participants_time,non_determinism_time,a_consistency_time,z3_running_time --shape 2d --type_plot scatter
```

in the `Docker`; the plots are `png` images saved in the directory `Examples/random_txt/tests_dafsms_1`.

3. Usage

3.1. Format of DAFSMs

The DAFSMs model (Definition 1 of our paper) is renderer in `TRAC` with a DSL which represents a DAFSM as sequences of lines, each specifying a transition of the DAFSM. We explain the format of transitions through the Simple Market Place contract (→ following Example 1 of our paper), which in our DSL is

```
_ {True} o:O > starts(c,string _description, int _price) {description := _description & price := _price} {string description, int price, int offer} S0
S0 {_offer > 0} b:B > c.makeOffer(int _offer) {offer := _offer} S1
S1 {True} o > c.acceptOffer() {} S2+
S1 {True} o > c.rejectOffer() {} S01
S01 {_offer > 0} any b:B > c.makeOffer(int _offer) {offer := _offer} S1
S01 {_offer > 0} b:B > c.makeOffer(int _offer) {offer := _offer} S1
```

hereafter called `SMP`; the names of states in `SMP` differ from those in Example 1, but this is immaterial for the analysis.

In general a transition consists of

- a source and a target state; a trailing `+` denotes final states (like `S2+` above)
- a guard specified in the notation of `Z3`
- a qualified participant `p : P` corresponding to *v* p : P in the paper, `any p : P`, or just `p`
- a call to an operation of the contract
- a list of `&`-separate assignments.

The first line of `SMP` is a special transition corresponding to the edge entering the initial state in Example 1 barred for

- the fact that the source state is `_` is used to identify the initial state
- the additional `_description` parameter, omitted in the paper for readability

The guard `True` in the transition is the *precondition* while the list of assignments `{description := _description & price := _price}` is followed by an explicit declaration of the contract variables to capture the assumption in the first item of Page 3 of the paper; the transition introduces a fresh participant `o` with role `O` which renders the object-oriented mechanism described just above Definition 1.

Conventionally, parameters start with `_` to distinguish them from contract variables.

3.2. Examples of non well-formed models

As seen in Section 2.2, `SMP` is well-formed; we now apply `TRAC` to detect non well-formed models. The file `azure/simplemarket_place_edit_1` contains a modified DAFSM obtained by replacing the `acceptOffer` transition of `SMP` with

```
S1 {True} x > c.acceptOffer() {} S2+
```

Executing in the `Docker`

```
python3 Main.py --filetype txt "azure/simplemarket_place_edit_1"
```

produces

```
The Path : _-starts-S0>S0-makeOffer-S1 does not contain the participant x : []
Error from this stage:S1_acceptOffer()_S2
--For _acceptOffer_0:   Check result ::  False
--- Participants       : False

(!) Verdict: Not Well Formed
```

stating that participant `x` has not been introduced. In fact, the `CallerCheck` finds a path to `S1` where participant `x` is not introduced (first line of the output above) identified in the trasition from `S1` to `S2` with label `acceptOffer` (second line of the output). The last three lines of the output inform the user that well formedness does not hold for the use of a non introduced participant.

The file `azure/simplemarket_place_edit_2` modifies `SMP` by replacing the transitions `acceptOffer` and `rejectOffer` respectively with

`S1 {False} o > c.acceptOffer() {} S01` and `S1 {False} o > c.rejectOffer() {} S01`

Executing now the command below in the `Docker`

```
python3 Main.py --filetype txt "azure/simplemarket_place_edit_2"
```

produces

```
Error from this state:S01_makeOffer(int _offer)_S1
--For _makeOffer_0:   Check result ::  False
--- A-Consistency: False

   Simplify of the Not Formula:  Not(And(Not(_offer <= 0), offer == _offer))  ::  True

   (!) Verdict: Not Well Formed
```

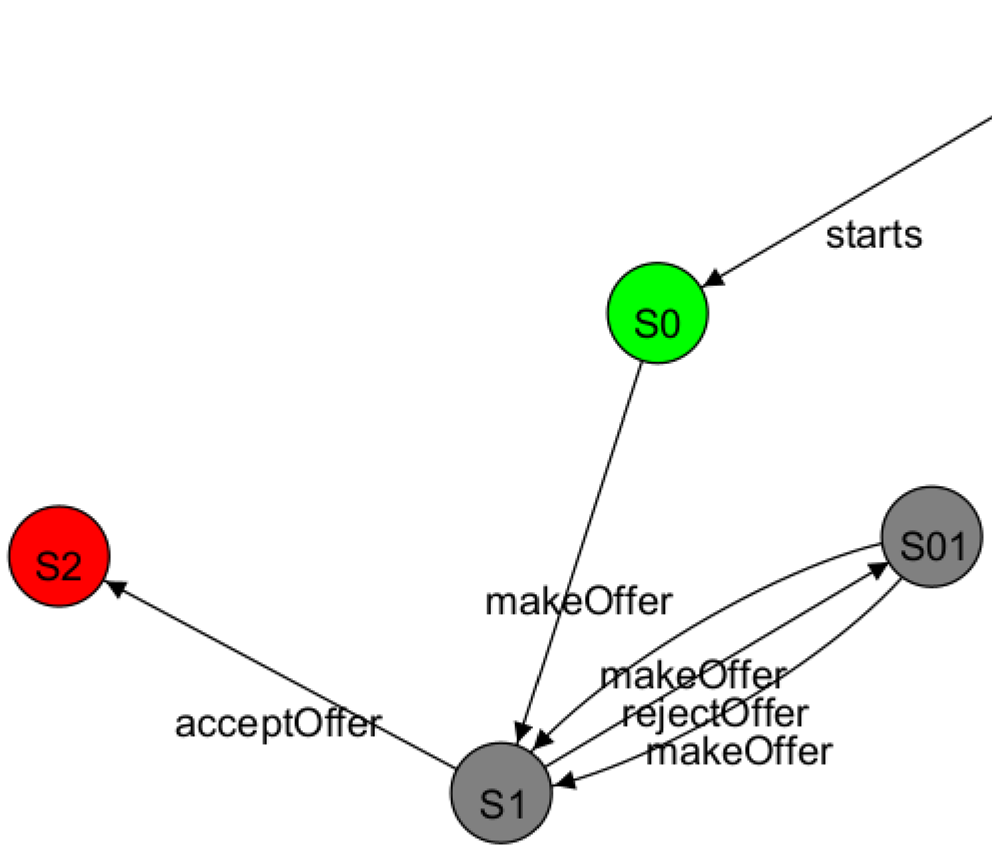
which tells that consistency is violated by the transition

```
S0 {_offer > 0} b:B > c.makeOffer(int _offer) {offer := _offer} S1
```

The simplification operated by `Z3` on the formula in the last but one line of the output yields (the formula) `True`.

The `Main.py` script used above accepts the `check_type <chk>` optional parameters where `<chk>` can take two qualifiers; `check_type` defaults to `1` which checks well-formedness and can be set to `fsm` to generate a visual representation of a DAFSM as a `png` file.

The image below for `SMP` is generated by invoking the `GraphStream` library (https://graphstream-project.org/) from our `GraphGen` component (cf. Figure 1 in the paper)



(labels are simplified for readability). The description in Section 3.1 of the paper wrongly states that `GraphGen` is "a third-party component", but in fact it should read that `GraphGen` is a wrapper to invoke `GraphStream`. Unfortunately, the image cannot be visualised from inside the `Docker` because `GraphGen` uses the functionality of `GraphStream` that displays the graph in an interactive window. So, to see the model it is necessary to use `TRAC` from outside the `Docker`.

3.3. Commands for performance evaluation

To evaluate the performances of `TRAC`, we created a randomizer that contains a generator of random models in our DSL, a program that applies `TRAC` on the generated models, and a visualiser to plot data from `csv` files. In the following, we explain how to perform each step.

Generating random examples

The following command generates 100 random models and saves them in the directory `Examples/random_txt/your_sub_dir_name`:

```
python3 Generate_examples.py --directory your_sub_dir_name --num_tests 100
```

The generation process can be customised setting optional parameters of `Generate_examples.py`; if not specified, all but the last four parameters default to randomly generated values:

- `--num_tests <num>` the number of tests to generate
- `--num_states <num>` the number of states per test
- `--num_actions <num>` is the number of actions
- `--num_vars <num>` is the number of variables
- `--max_num_transitions <num>` is the maximal number of transitions that should be at least the number of states (minus 1)

- `--max_branching_factor <num>` is the maximum branching factor should be greater or equal than 1; in corner cases, the branching factor is predominant and may lead to exceed the maximum number of transitions
- `--num_participants <num>` is the maximum number of participants variables
- `--steps <num>` the increment steps for generating tests (meaningful only if `--incremental_gen` below is set to true; default: `10`)
- `--incremental_gen [True|False]` enables/disables incremental generation of models (default: `False`)
- `--merge_only_csv [True|False]` if set to `True` merges results into a single `csv` file; all other parameters are ignored when this is flag is used (default: `False`)
- `--num_example_for_each <num>` is the number of models to generate for each configuration (default: `5`).

To generate the models used in Section 4 of the paper, we ran the following command:

```
python3 Generate_examples.py --directory tests_dafsms_1 --steps 5 --num_example_for_each 5 --num_tests 30 --incremental_gen True
```

Warning: the directory `Examples/random_txt/tests_dafsms_1` in the `Docker` is populated with the models and `csv` files generated for the experiments reported in the paper. Executing the command above in the `Docker` would overwrite the files generated for the experiments in our paper.

The `csv` file containing metadata of each generated example. The full description of the metadata is in section 4 below.

Well-formedness check of the generated examples starts immediately after the generation is completed.

The overall process allows the auto-generation and checking of DAFSMs models, facilitating the evaluation of `TRAC`.

Running a set of examples

The following command configuration allows the check of a set of examples in a given sub-repository `<subdir>` in `Examples/random_txt`:

```
python3 Random_exec.py <subdir> --number_test_per_cpu 5 --number_runs_per_each 10 --time_out 300000000000
```

The latter command takes all the models metadata information in file `list_of_files_info.csv` within the `<subdir>`, allocates 5 models to each CPU for verification, and checks each model 10 times to have an average measured time, each CPU will output a `csv` file `list_of_files_info_{id}.csv` at the end, and all generated `csv` will be merged into `merged_list_of_files_info.csv` upon completion of all the examples.
The checking process can be customized by setting some parameters of `Random_exec.py`. The full list of available parameters follows:

- `--merge_csv [True|False]` only merges individual generated `csv` results into `merged_list_of_files_info.csv`
- `--add_path [True|False]` count the number path for each model in the `list_of_files_info.csv`
- `--number_test_per_cpu <num>` determines how many tests are to run in parallel per CPU (default is `5`)
- `--number_runs_per_each <num>` specifies how many times to run each model check (default is `10`)
- `--time_out <num>` sets a timeout limit (in nanoseconds) to perform each model check (default is `300000000000` 5 minutes).
  The checking process splits tests for parallel execution, each thread output results into a `csv` file and merges them upon completion. Results are stored in a subdirectory within `Examples/random_txt/<subdir>` to preserve data.

Plotting Results

To plot data using `Plot_data.py`, the following command can be customized with some given parameters below:

```
python3 Plot_data.py <directory> --shape <shape> --file <file_name> --fields <fields_to_plot> --pl_lines <lines_to_plot> --
type_plot <plot_type>
```

- `<directory>` the directory where the test data CSV is located, relative to `./Examples/random_txt/` where the `merged_list_of_files_info.csv` is
- `--file <str>` specify the CSV file name without the extension, defaulting to `merged_list_of_files_info`
- `--shape <str>` choose the plot shape: `2d` or `3d`
- `--fields <list>` set the column(s) in the `csv` to plot against time, default is `num_states`
- `--pl_lines <list>` define which time metric column(s) in the the `csv` to plot against the `--fields`, with default `participants_time`,`non-determinism_time`,`a-consistency-time`
- `--type_plot <str>` choose the type of 2D plot, with `line` as the default. (values `line`, `scatter`, `bar`)
- `--scale [log|linear]` Y scale function, with default `log`. (values `log`, `linear`)

To generate the plots of section 4 of the paper, we ran the following commands:

```
python3 Plot_data.py tests_dafsms_1 --file merged_list_of_files_info --field num_states,num_transitions,num_paths --pl_lines
participants_time,non_determinism_time,a_consistency_time,z3_running_time --shape 2d --type_plot scatter --scale linear

python3 Plot_data.py tests_dafsms_1 --file merged_list_of_files_info --field num_states,num_transitions,num_paths --pl_lines
participants_time,non_determinism_time,a_consistency_time,z3_running_time --shape 2d --type_plot scatter --scale log
```

All generated plots are stored in the directory `<directory>`. Since it's not possible to visualize these plots directly from within Docker, you'll need to copy them to a directory outside of `Docker`. Use the following command:

```
docker ps
docker cp <containerID>:/home/TRAC/src/Examples/random_txt/<directory> <localPath>
```

The first line in the command above outputs a list of running containers. Locate the container ID of the `loctet/trac_dafsms:v1` image, and replace `<containerID>` with it in the second line. `<localPath>` represents the path on your local machine where you want to copy the plots to. Once the copy is completed, you can view the plots outside of `Docker`.

3.4. Run your own examples

Now that the verification of some examples is completed, you can design some DAFSMs and check if they are well-formed by giving the name of the file to the following command (`python3 Main.py --filetype txt "xxxxxxxxx"`)

```
/!\ Note that all manually executed examples should be placed in the folder  `Examples/dafsms_txt`. You can create
sub-dirs, just be assured to run the above command with the exact name of the example `< subdir>/< example>`.
```

Some examples can be found in `Examples/other_tests` testing some scenarios that are not found in the Azure repository.

4. Documentation

To access the complete documentation, including detailed code explanations and usage instructions, please download the zipped file containing the HTML format documentation from the GitHub repository. After downloading, unzip the file to access the Sphinx-generated documentation. This documentation offers valuable insights into the inner workings of the TRAC system, facilitating a deeper understanding of its features and capabilities.

CSV Header Description

- `path` the path to the model file
- `num_states` number of states
- `num_actions` number of actions
- `num_vars` number of variables
- `max_branching_factor` maximum branching factor
- `num_participants` number of participants
- `num_transitions` number of transitions
- `seed_num` seed number used for randomization
- `min_param_num` actual minimum number of parameters
- `average_param_num` actual average number of parameters
- `max_param_num` actual maximum number of parameters
- `min_bf_num` actual minimum number of branching factors
- `average_bf_num` actual average number of branching factors
- `max_bf_num` actual maximum number of branching factors
- `num_paths` number of paths
- `verdict` verdict of the verification process
- `participants_time` time taken for checking participants
- `non_determinism_time` time taken for non-determinism check
- `a_consistency_time` time taken for action consistency check
- `f_building_time` time taken for formula building
- `building_time` time taken for building
- `z3_running_time` time taken for running Z3
- `total` total time taken for the process
- `is_time_out` indicates if there was a timeout during processing.

5. Tips

Each of the commands - `Main.py`, `Generate_examples.py`, `Random_exec.py`, and `Plot_data.py` - is equipped with a `--help` option. When using the `--help` option, detailed usage instructions and available options for each command will be displayed.

```
    python3 Main.py --help
```

This functionality is designed to assist users in comprehending and efficiently utilizing the tool's features.