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
   ```

The former command downloads the `Docker` image of `TRAC` while the latter starts a container with an interactive terminal.

2- Reproducibility

2.1 How the table 1 what generated

The `Table 1: Features in the Azure benchmark` In the `TRAC` paper present some features extracted from the implementation of the example. The Azure repositoty describe the example and give an implementation in `Solidity`:
Hello Blockchain, Simple Marketplace, Basic Provenance, Digital Locker, Refrigerated Transportation, Asset Transfer, Room Thermostat, Defective Component Counter, Frequent Flyer Rewards Calculator.

The example in the table bellow point to the `.sol` file implementing the example, and we give the line where the feature can be found and also the line in the `TRAC` repository that models the feature.

Example (link to .sol)	Line in Code for the feature	How TRAC handle it
Simple Marketplace	BI : ✅ L44	BI : ✅ `b:B` > c.makeOffer (Line 2 and 6)
Hello Blockchain	BI : ✅ Line 19 & 31	BI : ✅ `RqT:Resquester`, `RpD:Responder` (Line 1 and 3)
Bazaar Item Listing	ICI : ❌ BazaarItem (Line 78) ItemList(Line 40) BI : ✅ BazaarItem (Line 76) ItemList(Line 33)	
Ping Pong Game	ICI : ❌ (Line 18 and 82) BI : ✅ (Line 16 and 67)	
Defective Component Counter	PP : ✅ Line 26	PP : ✅ `m:M` (Line 1)
Frequent Flyer Rewards Calculator	BI : ✅ Line 20 PP : ✅ Line 18 & 21	BI : ✅ `ar:AirRep` (Line 1) PP: ✅ `participant FL f` [Line 1]
Room Thermostat	PP : ✅ Line 16 & 18 & 19	PP: ✅ `participant I i`, `participant U u` (Line 1)
Asset Transfer	BI : ✅ Line 18, RR : 🔽 Line 97	BI: ✅ `b:B` (Line 3) RR: 🔽 reject goes to `S01` at that stage if we assume `b` is `new` them it somehow destroy previous `b` as we rebind it to `new B`

Example (link to .sol)	Line in Code for the feature	How TRAC handle it
[Basic Provenance]()	BI : ✅ Line 19 PP : ✅ Line 17 RR : 🔽 Line 51	BI: ✅ `cp:Conterparty` ([Line 1]()) PP: ✅ `participant SupplyOwner so` (Line 1, 2,3) RR: 🔽 Since the protocol does not evolve after `S2` (final state) we assume all participants are reintroduced if we restart the protocol
[Refrigerated Transportation]()	BI : ✅ Line 32 PP : ✅ Line 28 RR : 🔽 Line 143 MRP : 🔽 Line 119	BI: ✅ o:O ([Line 1]()) PP: ✅ `participant D d`, `participant SC sc`, `participant OBS obs` (Line 1, 5) RR: 🔽 Since the protocol does not evolve after `Success` (final state) we assume all participants are reintroduced if we restart the protocol MRP: 🔽 This are participants of same role, they are assign same values
[Digital Locker]()	BI : ✅ Line 21 PP : ✅ Line 19 RR : 🔽 Line 102 MRP : 🔽 Line 76, 91	BI: ✅ `o:O` ([Line 1]()) PP: ✅ participant Banker ba (Line 1) RR: 🔽 Since `RejectSharingLock` goes back to `S2`, participant `cau` can only invoke function when the new one will be introduce in `S4` MRP: 🔽 `AcceptSharingLock` we directly pass the new participant as parameter so there is not a role changing but introducing new one

2.2- Run the Azure repository examples

The `simplemarket_place` example taken from [Azure repository]() is already within designed examples directory (`Examples/dafsms_txt/azure`) as well as the [other examples]() from the Azure blockchain-workbench.

To run the "simplemarket_place" example with `TRAC`:

1. Navigate to the `TRAC` Directory: from the `Docker` container execute `cd src`.
2. Execute the Example:

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

This command tells `TRAC` to proceed to the check of the "simplemarket_place" example. The output is `(!) Verdict: Well Formed` telling that the DAFSMs given as input is well formed.

Example	Command to run the example	Verdict
Asset transfer	`python3 Main.py --filetype txt "azure/asset_transfer"`	`(!) Verdict: Well Formed`
Basic provenance	`python3 Main.py --filetype txt "azure/basic_provenance"`	`(!) Verdict: Well Formed`
Defective component counter	`python3 Main.py --filetype txt "azure/defective_component_counter"`	`(!) Verdict: Well Formed`
Digital locker	`python3 Main.py --filetype txt "azure/digital_locker"`	`(!) Verdict: Well Formed`
Frequent flyer rewards_calculator	`python3 Main.py --filetype txt "azure/frequent_flyer_rewards_calculator"`	`(!) Verdict: Well Formed`
Hello blockchain	`python3 Main.py --filetype txt "azure/hello_blockchain"`	`(!) Verdict: Well Formed`
Refrigirated transport	`python3 Main.py --filetype txt "azure/refrigirated_transport"`	`(!) Verdict: Well Formed`
Room thermostat	`python3 Main.py --filetype txt "azure/room_thermostat"`	`(!) Verdict: Well Formed`

2.3- Run the randomized examples

1. Navigate to `TRAC` Directory: Ensure you're in the `src` directory of `cd src`
2. Run of examples of the `TRAC` tool paper

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

This will run the 135 random DAFSMs in the folder `Examples/random_txt/tests_dafsms_1` with subfolders, each folder having 5 tests and a CSV file(`list_of_files_info.csv`) containing metadata of those 5 examples. The check will start, going through each file and performing the well-formedness check. (`this process can be long depending on your environment`). While running the checks further csv files will be generated and merged (to `merged_list_of_files_info`) when all checks are completed.
Now you can plot the data to visualize different running time by executing the following command

```
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
```

The command will generate the graphs in `section 4 of the paper`. All plots images are saved in the `Examples/random_txt/tests_dafsms_1` directly.

3- Usage

3.1- Format of DAFSMs

The definition of the DAFSMs model is given in `section 2 of TRAC paper` more precisely what is the structure of a DAFSM.

Let's consider the Simple Market Place(SMP) example, given in `section 1`

The deploy transition in looks like this:

`_ {True} o:O > starts(c,string _description, int _price) {description := _description & price := _price} {string description, int price, int offer} S0`

Deploy transition introduce new participant `o` of role `O`, which `starts` the coordinator `c` by passing a description and a price. These values are assigned to declared variable `string description` , `int price` and `int offer`. here the precondition(guard `g`) is `True`.

- states: (`_` to `S0`) here `_` is a special state only used to deploy the coordinator
- guard(g_0): `{True}`
- Participant: `o:O` new participant `o` of role `O`
- function: `starts` a keyword to deploy the coordinator
- coordinator id: `c`
- parameter: `string _description, int _price`
- declaration: `{string description, int price, int offer}` where we are declaring states variables `only in the deploy transition`
- assignments: `{description := _description & price := _price}`

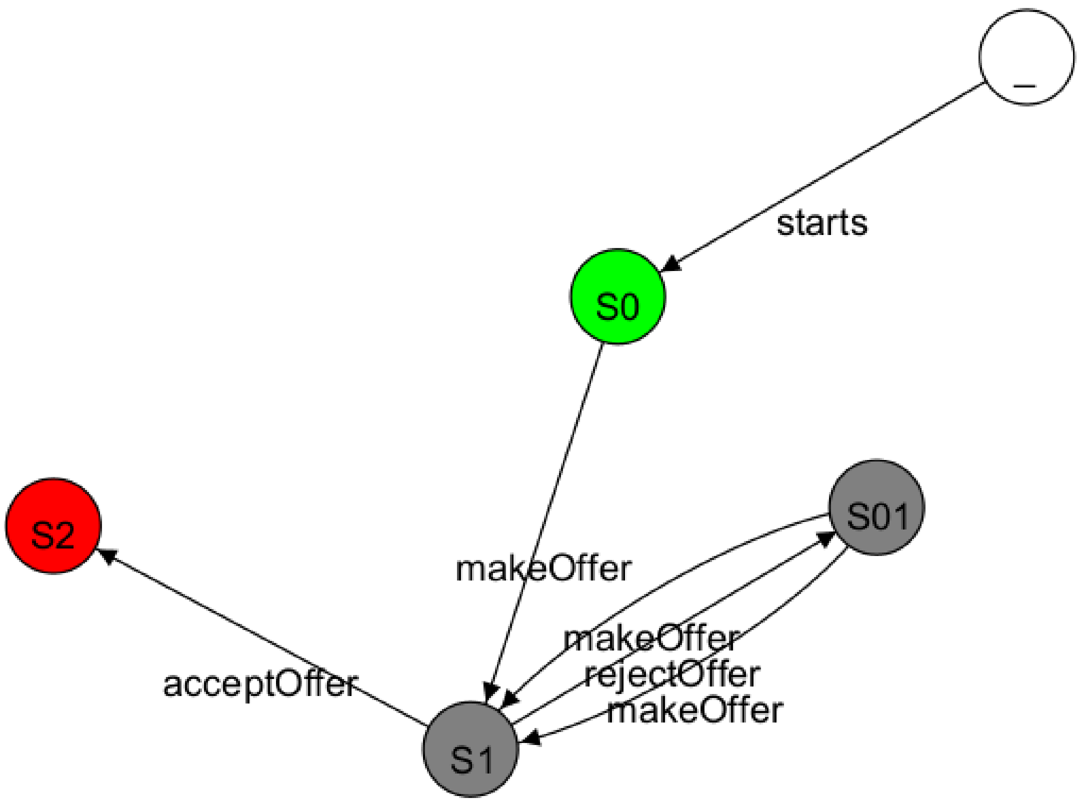
To make an offer, we have the transition `S0 {_offer > 0} b:B > c.makeOffer(int _offer) {offer := _offer} S1` that allow new participant `b` of role `B` to make an offer by passing a price `_offer` as parameter to the function, the guard requires `_offer` to be `> 0` to update the value of the state variable `offer` and move the protocol to `S1`

To accept the offer, the transition `S1 {True} o > c.acceptOffer() {} S2+` can be invokes by the previously introduced `o` to accept the offer and move to a final state `S2` as it has the sign `+` after.

To reject the offer, the transition `S1 {True} o > c.rejectOffer() {} S01` can be invoked by `o` and move the protocol back to a state where new byer or existing buyer can now make an offer. So we have these 2 transitions: `S01 {_offer > 0} any b:B > c.makeOffer(int _offer) {offer := _offer} S1` can be invoke only by any existing participant with role `B`. and `S01 {_offer > 0} b:B > c.makeOffer(int _offer) {offer := _offer} S1` can be invoke only by fresh one. This allow the function `makeOffer` to be available to both `new participant` and `existing ones`.

The `TXT` file for the SMP example should be this :

```
_ {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
```



3.2- Different commands

We ran the `simplemarket_place` example in a section above.
Non Well Formed Examples
Let's modify the previous simple market place to make it not well-formed.
Modify transition `S1 {True} o > c.acceptOffer() {} S2+` to `S1 {True} x > c.acceptOffer() {} S2+` this modification say `x` can accept the offer, here, `x` is never introduce and therefore the new given DAFSMs should not be well formed. Run

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

After running the check, we have this output:

```
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
```

This tells that the participant `x` has not been introduced.

The `CallerCheck` found a path from the initial state `_` to `S1` where there is not an introduction of participant `x` (`The Path : _-starts-S0>S0-makeOffer-S1 does not contain the participant x : []`)

This line `--For _acceptOffer_0:   Check result ::  False` tells that the check of the model failed when checking the first occurrence of the function `acceptOffer`

This line `--- Participants       : False` tells the test which failed if `Participant`

Let's do another modification:
Modify transition `S1 {True} o > c.rejectOffer() {} S01` to `S1 {False} o > c.rejectOffer() {} S01`
and transition `S1 {True} o > c.acceptOffer() {} S01` to `S1 {False} o > c.acceptOffer() {} S01`
Here we are creating DAFSMs where from `S1` there is no possible outgoing transition to progress base on the `guard satifiability`.
The Tool should spot the fact that the model has `inconsistency`.Run

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

After running the check, we have an output:

```
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
```

This tells that the consistency rule is violated with transition `S0 {_offer > 0} b:B > c.makeOffer(int _offer) {offer := _offer} S1` reaching `S1`.

This line `--For _makeOffer_0:   Check result ::  False` tells that the check of the model failed when checking the first occurrence of the function `makeOffer`

This line `--- A-Consistency: False` tells the test which failed if `Consistency`

The line `Simplify of the Not Formula:  Not(And(Not(_offer <= 0), offer == _offer)) ::  True` is the `Simplify Z3` formula to check the `negation of the satisfiability formula` In this case the negation is `True`.

Main File

The `Main.py`, can take some configurations as follows:

- `file_name`: Specifies the name of the file (JSON or TXT) to process, without its extension. This is the primary input for `TRAC` to analyse.
- `check_type`: Optional. Defines the type of check to perform on the input file. It can be one of four options:
  - `1` for Well-Formedness Check,
  - `fsm` for printing the Finite State Machine (FSM). The default is `1`.
- `--filetype`: Optional. Indicates the type of the input file, either `json` or `txt`. The default is `json`.
- `--non_stop`: Optional. Determines the mode of checking, where `1` (default) continues checking even after errors are found, and `2` stops immediately when an error is detected.
- `--time_out`: Optional. Sets a timeout for the operation in seconds. The default is `0`, meaning no timeout.
  This detailed explanation provides a comprehensive guide on how to utilize `Main.py` for different operations within the `TRAC` tool.

Generating random examples
To generate DAFSMs examples with `Generate_examples.py`, follow these steps:

1. Run Generate_examples.py : Use the command below, adjusting parameters as needed.

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

Replace `your_directory_name` with the desired directory to store test files, and adjust `--num_tests` to the number of examples you wish to generate.

2. Parameters:
   The parameters for `Generate_examples.py` enable customization of the DAFSMs example generation process. If not specified, values for these parameters are generated randomly:
   - `--directory`: Specifies the directory to save generated examples.
   - `--num_tests`: The number of tests to generate.
   - `--num_states`: The number of states per test.
   - `--num_actions`: The number of actions.
   - `--num_vars`: The number of variables.
   - `--max_num_transitions`: The maximum number of transitions.
   - `--max_branching_factor`: The maximum branching factor.
   - `--num_participants`: The number of participants.
   - `--incremental_gen`: Enables incremental generation.

  - `--merge_only_csv`: Merges results into a single CSV without generating new tests.
  - `--steps`: The increment steps for generating tests.
  - `--num_example_for_each`: The number of examples to generate for each configuration.

3. Output: Examples are created in a subdirectory within `Examples/random_txt`. A CSV at the root of this directory contains metadata for each generated example, including paths, number of states, actions, variables, branching factors, and timings.

This process allows for the automated generation and analysis of DAFSMs examples, facilitating comprehensive testing and verification of DAFSMs with `TRAC`.

Running a sets of examples

To execute multiple examples with `Random_exec.py`, the command format and parameters are as follows:

```
python3 Random_exec.py --directory <subdir> --merge_csv --add_path --
number_test_per_cpu <num> --number_runs_per_each <runs> --time_out <nanoseconds>
```

- `--directory`: Specifies a subdirectory in `Examples/random_txt` where the examples and `list_of_files_info.csv` are located.
- `--merge_csv`: Merges individual CSV results into `merged_list_of_files_info.csv`.
- `--add_path`: Just count the number_path to each test in the CSV.
- `--number_test_per_cpu`: Determines how many tests are run in parallel per CPU.
- `--number_runs_per_each`: Specifies how many times to run each test.
- `--time_out`: Sets a timeout limit for each test.

The process splits tests for parallel execution, outputs results to CSV files, and merges them upon completion. Results are stored in a subdirectory within `Examples/random_txt/<subdir>` to preserve data. Execution time varies with the test suite size.

Plotting Results

To plot results using `Plot_data.py`, follow these command-line instructions, customizing them based on your needs:

```
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.
- `--shape`: Choose the plot shape: `2d`, `3d`, or `4d`.
- `--file`: Specify the CSV file name without the extension, defaulting to `merged_list_of_files_info`.
- `--fields`: Set the column(s) to plot against time, default is `num_states`.
- `--pl_lines`: Define which time metric to plot, with defaults including participants time, non-determinism time and a-consistency-time.
- `--type_plot`: Choose the type of 2D plot, with `line` (values `line`, `scatter`, `bar`)as the default.

This command allows for versatile plotting configurations, adjusting for different dimensions and aspects of the data captured in the CSV file. All plots are saved in the directory directly.

3.3- Run you own examples

Now that your first example is completed, you can design some DAFSMs and play around with the command by just changing the name of the file in the command (`python3 Main.py --filetype txt "xxxxxxxxx"`)

/!\ All manually executed examples should be kept in the folder `Examples/dafsms_txt` you can create sub-dirs, just be assured to give the exact path to the command `Main.py`.

Some examples are in the `Examples/other_tests` where you can find different test cases.

4- Documentation

The full documentation in HTML format can be found locally in the sub-dir `docs`

CSV Header Description

1. path: The path of the file.
2. num_states: Number of states in the FSM.
3. num_actions: Number of actions in the FSM.
4. num_vars: Number of variables in the FSM.
5. max_branching_factor: Maximum branching factor in the FSM.
6. num_participants: Number of participants in the FSM.
7. num_transitions: Number of transitions in the FSM.
8. seed_num: Seed number used for randomization.
9. min_param_num: Minimum number of parameters.
10. average_param_num: Average number of parameters.
11. max_param_num: Maximum number of parameters.
12. min_bf_num: Minimum number of branching factors.
13. average_bf_num: Average number of branching factors.
14. max_bf_num: Maximum number of branching factors.
15. num_paths: Number of paths in the FSM.
16. verdict: Verdict of the verification process.
17. participants_time: Time taken for checking participants.
18. non_determinism_time: Time taken for non-determinism check.
19. a_consistency_time: Time taken for action consistency check.
20. f_building_time: Time taken for formula building.
21. building_time: Time taken for building.
22. z3_running_time: Time taken for running Z3.

23. total: Total time taken for the process.
24. is_time_out: Indicates if there was a timeout during processing.

5- Tips

All commands provided, such as running tests, generating examples, executing multiple examples, and plotting results with various scripts like `Main.py`, `Generate_examples.py`, `Random_exec.py`, and `Plot_data.py`, come equipped with a `--help` option. Utilizing `--help` will display detailed usage instructions and available options for each command, aiding users in understanding and effectively utilizing the tool's features.