ATVA'24 Artifact: `Learning Broadcast Protocols with LeoParDS`

Noa Izsak [1], Dana Fisman [1], and Swen Jacobs [2]

izsak@post.bgu.ac.il, dana@cs.bgu.ac.il, jacobs@cispa.de

1. Ben Gurion University, Beer-Sheva, Israel

2. CISPA Helmholtz Center for Information Security, Saarbrücken, Germany

This artifact serves as an example template for artifacts submitted to ATVA'24 artifact evaluation.

Table of Contents

Quickstart

Requirements

The following software is required for running this artifact:

- docker

Setup Steps

- Install Docker by following the instructions at https://docs.docker.com/get-docker/.
- Download the artifact Learning-Broadcast-Protocols-with-LeoParDS-artifact.zip from zenodo
- Unzip the artifact:

  Note: This will unzip the artifact directory structure and files into the current working directory, including the compressed docker image `image.tar`.

  ```
  unzip Learning-Broadcast-Protocols-with-LeoParDS-artifact.zip
  ```

1. Build the Docker image:

```
docker build -t bpcode:bpcode .
```

Or import the image file:

```
docker load -i image.tar
```

This will import the docker image named `<image>`. This might take a few minutes.

2. Run the Docker container:

If you would like to copy the produced results / outputs to your local machine (for example to take a look at the produced figures), the easiest way is to mount your local directory in the docker container.

To do so, run the container in the directory you want the files to be stored with the additional `-v` flag:

```
docker run -d -it --name bpcodecontainer -v .:/storage bpcode:bpcode
```

This will make you current directory accessible in the container under the path `/storage`.

Otherwise, run:

```
docker run -d -it --name bpcodecontainer bpcode:bpcode
```

3. Access the Docker container:

```
docker exec -it bpcodecontainer /bin/bash
```

Note This will start a new Docker container.

Smoke Test Steps

After completing the above steps, within the Docker environment:

4. Execute the smoke test:

```
python Test_run.py
```

If you want to copy to the produced output file (and you started the container with your local directory mounted, see step 2 *Run the Docker container*), simply execute in the Docker enviornemnt:

```
cp BP_results_subsume_cs_*.csv /storage
```

and

```
cp BP_results_non_cs_*.csv /storage
```

Output of Smoke Test Execution

Click here for the Output of Smoke Test Execution

The output should including visual printing, along with the creation of 10 files:

- Files named `BP_results_non_cs_{i}.csv` and `BP_results_subsume_cs_{i}.csv` for i in $[1, 2, 3, 4, 5]$.
  - The first 5 files, (`BP_results_non_cs_{i}.csv`) contain randomly generated words, so the execution time may vary but should be completed in less than 5 minutes.
  - The other 5 files, (`BP_results_subsume_cs_{i}.csv`).

The entire process should take about 5 minutes to complete.

In `BP_results_subsume_cs_{i}.csv` the `output_BP` might slightly vary between runs, but `right_output` should always be **True** (refer to figure below):

failed_cor	timeout	amount_o	amount_o	origin_BP	output_BP	cutoff	CS_develo	CS_positiv	CS_negati	words_ad	longest_w	solve_SMT	right_output
FALSE	FALSE	2	2	states: 2,	states: 2,	4	0.067967	447	87	{'positive':	8	11.64656	(None, None, True)

Figure 1: image

For `BP_results_non_cs_{i}.csv` the `output_BP` is influenced by randomly generated words, but for the provided test inputs, it should terminate with `right_output` and `minimal_right_output` as **True** (refer to figure below):

failed_cor	timeout	amount_o	amount_o	origin_BP	output_BP	cutoff	CS_develo	CS_positiv	CS_negati	words_ad	longest_w	solve_SMT	right_outp	amount_o	minimal_(	minimal_s	minimal_right_output
FALSE	FALSE	2	3	states: 2,	states: 3,	27	0.000182	1	9	{'positive':	15	0.287027	TRUE	2	states: 2,	2.0896	TRUE

Figure 2: image

Note: Example outputs are provided, but they may vary. Refer to `Runing example for Test_run` folder.

The critical aspect is ensuring `right_output` is **True**, and for the second case, `minimal_right_output` is also **True**

Cleanup

1. Stop the existing Docker container:

```
docker stop bpcodecontainer
```

2. Remove the Docker container:

```
docker rm bpcodecontainer
```

3. Remove the Docker image:

```
docker rmi bpcode:bpcode
```

Available Bagde

The artifact has been uploaded to Zenodo and is accessible at zenodo (DOI: `10.5281/zenodo.10968038`).

Artifact Directory Structure

- `Dockerfile`: The docker file to build the artifact image
- `commands_to_run.text`: The commands you need to run after unziping the artifact
- `LICENSE`: Artifact license
- `Readme.md`: This file
- `paper.pdf`: An updated version of the submitted paper
- Directory `Result`
  - `result.csv`: Archive containing all files used for the evaluation
- Directory `Runing example for Test_run`
  - `BP_results_non_cs_1.csv`
  - `BP_results_non_cs_2.csv`
  - `BP_results_non_cs_3.csv`
  - `BP_results_non_cs_4.csv`
  - `BP_results_non_cs_5.csv`
  - `BP_results_subsume_cs_1.csv`
  - `BP_results_subsume_cs_2.csv`
  - `BP_results_subsume_cs_3.csv`
  - `BP_results_subsume_cs_4.csv`
  - `BP_results_subsume_cs_5.csv`
- `image.tar`: The docker image to replicate the evaluation.
- `Test_run.py`: Script to run a smoke test evaluation

- `ReplicateExperimentalResults.py`: Script to reproduced our evaluation
- `random_generator.py` : Script we used to run the evaluation
- `BP_Class.py` : Where BP_class is defined, that is the BP object itself
- `BP_gen.py` : Where BP_generator class is defined, that allow a user to generate a random BP
- `BP_Run.py` : Functions that allow us to infer on BPs for diffrent needs, i.e.:
  - subsume a CS (`run_subsume_cs`),
  - without subsuming a CS (`run_no_cs`),
  - without subsuming a CS and having a given positive precentage in the random generated sample (`run_no_cs_pos_perc`)
- `BP_Learn.py` : Where LearnerBp class is defined and use as the object that the inference procedure occurs on
- `State_Vector.py` : An helper class
- `Trie.py` : An helper class
- `main.py` : A function where you can plot the evaluated results as we present in the paper

Functional Badge

In our paper, we executed `generator_and_check_subsume_cs`, and for each input that terminated according to our criteria as explained in the paper, we added it to the data-set we examined. Due to the random generation of Broadcast Protocols (BPs), it is possible that running the tool independently may not reproduce the exact set of BPs. However, this variability is acceptable, as our aim is to encourage other fields to utilize this tool for their own purposes and explore new applications.

We have included the experiment-generated data under the Results folder in `results.csv`.

Note: Generating this data required several days of compute time across multiple compute nodes using a cluster. The resulting files are substantial in size, and regenrate these files will demand significant computing resources.

See ***Steps to Replicate the Experimental Results*** for instructions.

Steps to Replicate the Experimental Results

We ran all experiments on a cluster with Intel Xeon E5-2620 v4 CPUs. We allocated one CPU core and 30GB of RAM. For our experiments we randomly generated 4149 BPs with a number of states in $[2, 20]$, number of actions in $[0, 8]$. For each of these BPs we generated a random sample with a random number of words, F_w, in $[5, 100]$; with a bound of $\overline{\mathrm{M}}_l = 20$ on the length of the words, and a bound of 20 for the number of processes. The ratio of positive examples in the sample ranges between 0 and 1.

Note: Due to the fact the running the whole data-set should take several days, we present both the way to run the whole expirement and a subset of it that will take up to two hours

To reproduce our experimental results, follow these steps:

1. Start the Docker Container

- Install Docker by following the instructions at: //docs.docker.com/get-docker/.
- Download the artifact `Learning-Broadcast-Protocols-with-LeoParDS-artifact.zip` from zenodo
- Unzip the artifact:

  Note: This will extract the artifact directory structure and files into the current working directory, including the compressed Docker image `image.tar`.

  ```
  unzip Learning-Broadcast-Protocols-with-LeoParDS-artifact.zip
  ```

1. Build the Docker image:

```
docker build -t bpcode:bpcode .
```

Or import the image file:

```
docker load -i image.tar
```

This will import the docker image named `<image>`. This might take a few minutes.

2. Run the Docker container:

If you would like to copy the produced results / outputs to your local machine (for example to take a look at the produced figures), the easiest way is to mount your local directory in the docker container.

To do so, run the container in the directory you want the files to be stored with the additional `-v` flag:

```
docker run -d -it --name bpcodecontainer -v .:/storage bpcode:bpcode
```

This will make you current directory accessible in the container under the path `/storage`.

Otherwise, run:

```
docker run -d -it --name bpcodecontainer bpcode:bpcode
```

3. Access the Docker container:

```
docker exec -it bpcodecontainer /bin/bash
```

Note: This will start a shell in the Docker container where the following steps should be executed.

2. Execute Evaluation Runs

For a representative subset, use input paramter `1`:

```
python ReplicateExperimentalResults.py 1
```

Running the representative subset should take about 2 hours

For the entire data-set, use input parameter `0`:

```
python ReplicateExperimentalResults.py 0
```

Note: Running the whole data-set will require significant computing time (several days)

If you want to copy to the produced output file (and you started the container with your local directory mounted, see step 2 *Run the Docker container*), simply execute:

```
cp results_infer.csv /storage
```

3. Extract Numbers presented in the Paper

To extract the data (Table as well as the figures) presented in the paper:

```
python main.py
```

The table data will be displayed on the output screen, and the figures will be saved as `plot 6(a).png`, `plot 6(b).png`, `plot 7(a).png`, and `plot 7(b).png` within the project directory.

If you want to copy to the produced output files to your local machine (and you started the container with your local directory mounted, see step 2 *Run the Docker container*), simply execute:

```
cp *.png /storage
```

4. Cleanup

1. Stop the existing Docker container:

```
docker stop bpcodecontainer
```

2. Remove the Docker container:

```
docker rm bpcodecontainer
```

3. Remove the Docker image:

```
docker rmi bpcode:bpcode
```

Reusable Badge

In the following subsections, we provide detailed instructions on how to use different components of the code. Note that the code is thoroughly documented, so any helper function not described here can be understood by examining the code. However, to cater to those who may not be familiar with coding, especially in Python, we explain the main components with running examples, as also discussed in the paper.

Building the Docker Image

To build the Docker image, follow these steps:

1. Install Docker by following the instructions at: https://docs.docker.com/get-docker/.
2. The artifact contains all the necessary files to build the Docker image from scratch. Execute the following commands:
   1. Build the Docker image:
      `docker build -t bpcode:bpcode .`
   2. Save the Docker image to a tar file:
      `docker save -o image.tar bpcode`

Now you have created a Docker image!

- To use this Docker image, refer to the "Setup Steps" section. Specifically, in step 1 of the "Setup Steps" you can load the Docker image using: `docker load -i image.tar`

Learning-Broadcast-Protocols-with-LeoParDS

This repository contains the artifacts for the paper "Learning Broadcast Protocols with LeoParDS". It contains all the necesery code and inforamtion for using the code as showen in the paper and for new reserach and experements to be done.

More details on the algorithm and Broadcast Protocols (BPs) can be found in the paper, the rest of this file will explain the input and output file format as well as how to use our program.

BPGen - BP_generator:

```
class BP_generator:
  def __init__(self, min_number_of_states, min_number_of_act, max_number_of_states=None,
               max_number_of_act=None, print_info=False):
    ...
    self.number_of_act = na #as defined in the function
    self.number_of_states = ns #as defined in the function
```

```
    self.bp: BP_class = self.generate()

  def generate(self) -> BP_class:
    ...
    # returnes a BP object that has @number_of_states states and each state
    # has a unique action so it wouldn't be hidden and another @number_of_act
    # randomly distributed actions
```

Creating a random BP with number of states between 2 and 3 and number of actions between 1 and 2

```
bp = BP_generator(2, 1, 3, 2)
```

So a possible randomly generated BP is:

```
THE BP:
 initial state: 0
 actions: {0: {'a': 0, 'c': 1}, 1: {'b': 0}}
 receivers: {0: {'a': (0, False), 'b': (1, False), 'c': (1, False)},
             1: {'a': (1, False), 'b': (1, False), 'c': (0, False)}}
```

This is the result of `print(bp.bp)`

An illustration of this BP is as follows:

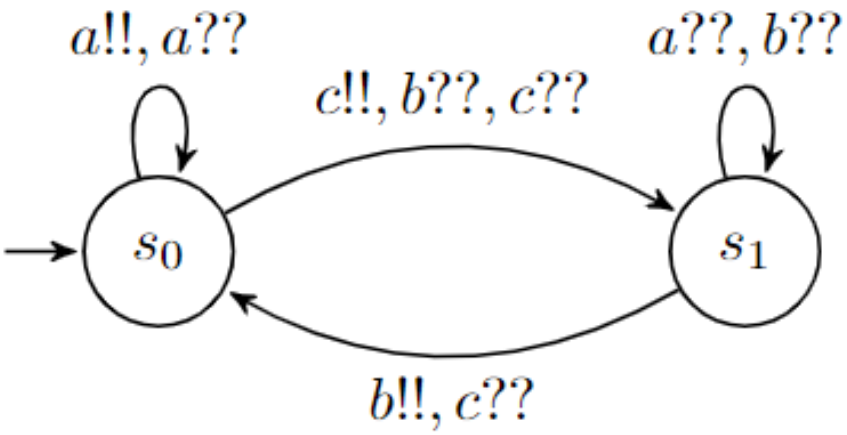

Figure 3: image

CSGen

CSGen - run_cs_to_a_limit

Under `BP_Learn.py` in class `BP_run`, you can find the following function:

```
def run_cs_to_a_limit(self, cutoff_limit, sample_limit, minimal=False):
```

The BP that we create this CS for is $self.bp$, cutoff_limit is $\overline{M}_p$ from the paper, represnting a bound on the number of processes. This is, in order to ensure termination also in case the given BP does not have a cutoff. We also have as input a `sample_limit`, in case you run on a computer with low memory resources, you can bound the size of the sample so if it is too high it will stop. This function is more direct as there are less inputs to tune, if you do want to make it more personal for diffrent uses see the fucntion below.

CSGen - run_subsume_cs

Under `BP_Learn.py` in class `BP_run`, you can find the following function:

```
def run_subsume_cs(self, words_to_add, are_words_given, cutoff_lim=None,
                   time_lim=None, word_lim=None, minimal=False):
  """
  a run that add amount of words_to_add to the cs and run it
  :param minimal: A parameter that is fed to the learning procedure
  :param word_lim: Limitation on amount of word to be generated,
                   in order to help with limited resources
  :param time_lim: If given, this is time limitation in sec
  :param cutoff_lim: If given, then cutoff limitation for running
  :param are_words_given: A boolean value representing whether we
                          create a sample (not necessarily a CS)
  for words_to_add amount or is the words are already given to us
  :param words_to_add: Number of words if are_words_given==False
                       or the set of words if are_words_given==True
  """
    ...
```

As in the previous run function, the BP that we create this CS for is $self.bp$, `cutoff_lim` is $\overline{M}_p$ from the paper, represnting a bound on the number of processes. This run function create a CS for the BP by the algorithm that we created and then potantialy padding it with additional words (`words_to_add`), either create it by itself or randomly generate new words (depends on `are_words_given`).

RSGen

RSGen - no positive ratio

RWGen - run_no_cs Under `BP_run.py` in class `BP_run`, you can find the following function:

```
def run_no_cs(self, words_to_add, words_are_given, maximal_procs=20,
              maximal_length=20, minimal=False):
  """
  if words_are_given==True then words_to_add are sample dictionary.
  otherwise, words_to_add is an int of number of words to add.
  a run that add amount of words_to_add to the cs and run it
  :param minimal: whether we want to invoke BPInfMin or not
  :param words_to_add: int if words_are_given=False, otherwise a dictionary of sample
  :param words_are_given: boolean value
  :param maximal_procs: maximal allowed processes for word in the sample
  :param maximal_length: maximal allowed length of word in the sample
  ...
  """
  char_set = {'positive': {}, 'negative': {}}
  start_time = time.perf_counter()
  if not words_are_given:
      char_set, words_added = self.create_sample(words_to_add, char_set,
                                                 maximal_procs, maximal_length)
  else:
      char_set = words_to_add
      words_added = words_to_add
  end_time = time.perf_counter()
  learn_bp = LearnerBp(char_set, self.bp, end_time - start_time, words_added)
  ...
  learn_bp.learn(minimal)
  ...
```

Where the function `create_sample` expand the given sample (in our case, `char_set` which is empty) in `words_to_add` amount where `maximal_procs` is the maximal number of processes to be considered in the sample resp. `maximal_length` for the length of the sample.

`learn_bp` is a LearnerBp object that the learning procedure will happen upon. sending it to `learn_bp.learn()` starts the learning procedure.

Defaultively, `.learn()` is for BPInf, if we want BPInfMin we will call it with `.learn(minimal=True)`. This sample (that is not necessarily a CS), is for $self.bp$. `words_to_add` represents F_w from the paper, `maximal_procs` represents $\overline{\mathrm{M}}p$ and `maximal_length` represnts $\overline{\mathrm{M}}_l$. Where 20 is the defaultive value for both of them.

RSGen - with positive ratio

RPWGen - run__no__cs__pos__perc Under `BP_run.py` in class `BP_run`, you can find the following function:

```
def run_no_cs_pos_perc(self, words_to_add, pos_perc, length_limit=20,
                       procs_limit=20, minimal=False):
  """
  :param minimal: whether we want to invoke BPInfMin or not
  :param words_to_add: int amount of words to add
  :param pos_perc: positive % of total words
  :param length_limit: longest word limit
  :param procs_limit: maximal procs limit
  """
  char_set = {'positive': {}, 'negative': {}}
  start_time = time.perf_counter()
  char_set, words_added = self.create_sample_pos_perc(words_to_add, char_set,
                                                      pos_perc, length_limit,
                                                      procs_limit)
  end_time = time.perf_counter()
  learn_bp = LearnerBp(char_set, self.bp, end_time - start_time, words_added)
  ...
  learn_bp.learn(minimal)
  ...
```

Similar to the above, but with the pos_perc option. This sample (that is not necessarily a CS), is for $self.bp$. `words_to_add` represents F_w from the paper, `pos_perc` represents F_r from the paper, value between $[0, 1]$, `procs_limit` represents $\overline{\mathrm{M}}p$ and `length_limit` represnts $\overline{\mathrm{M}}_l$. Where 20 is the defaultive value for both of them.

An Example:

Given the folloeing BP:

let's call this BP B_1, so for RSGen with this BP and parameters:

- $F_w = 5$,
- $\overline{\mathrm{M}}_l = 5$,
- $\overline{\mathrm{M}}p = 3$
- $F_r = 0.2$

The output could be the sample $S = \{(aabab, 2, F), (abbb, 2, F), (baa, 3, T), (bba, 2, F), (ba, 1, F)\}$.

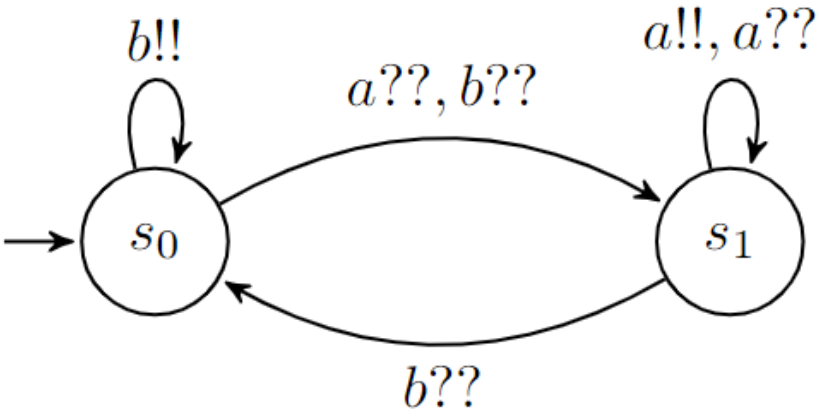

Figure 4: image

BPInf and BPInfMin

For both of them we run the function `learn` that is under `BP_Learn.py`. The boolean parameter `minimal` in the learn function determend whether we run BPInf (minimal=False) or BPInfMin (minimal=True)

```
def learn(self, minimal=False):
```

Running examples:

More examples and explanation about the fucntions can be find in the code

Example 1:

The results will be saved in the csv file : BP_results_cs_subsumed Note, since this is random generation procedure, we cannot geratnee termination, therfore, running this procedure may result in "print(f"The random generated BP has either no cutoff or it is greater then {cutoff}")". Id it do converage then the procedure will return an appropriate random generated BP and an Infered BP from the sample

```
def run_a_random_bp_example(minimal=False):
  if minimal:
      df = pd.DataFrame(columns=min_column)
  else:
      df = pd.DataFrame(columns=non_min_column)
```

```
cutoff = 15
timer_c = 900
word_lim = 1500
bp = BP_generator(3, 1, max_number_of_act=2)
learner = BP_run(bp.bp)
bp_min_acts, bp_min_rec, bp_acts, bp_rec, solution = learner.run_subsume_cs(0, False,
                                                                            cutoff, timer_c,
                                                                            word_lim=word_lim,
                                                                            minimal=minimal)
if solution['failed_converged']:
    new_row = pd.DataFrame([solution], columns=min_column)
    df = pd.concat([df if not df.empty else None, new_row], ignore_index=True)
    df.to_csv(f'BP_results_cs_subsumed.csv', index=False)
    print(f"The random generated BP has either no cutoff or it is greater then {cutoff}")
else:
    bp_learned = BP_class(len(bp_acts), bp_acts, 0, bp_rec)
    solution['right_output'] = equivalent_bp(bp.bp, bp_learned, solution['cutoff'])
    print(f"Are the two BP's are equivalent?:", solution['right_output'])
    if minimal:
        bp_learned = BP_class(len(bp_min_acts), bp_min_acts, 0, bp_min_rec)
        solution['minimal_right_output'] = equivalent_bp(bp.bp, bp_learned,
                                                         solution['cutoff'])
        print(f"Are the two BP's are equivalent?: minimal:",
                                             solution['minimal_right_output'])
    new_row = pd.DataFrame([solution],  columns=min_column)
    df = pd.concat([df if not df.empty else None, new_row], ignore_index=True)
    df.to_csv(f'BP_results_cs_subsumed.csv', index=False)
```

Where `learner.run_subsume_cs(0, False, cutoff, timer_c)` which is equivalent to the CS development by the algorithm that was developed for maximal `cutoff` and time_bounding of `timer_c`

A fucntion that do so is `run_a_random_bp_example()`

A posible output can be:

```
SAT
self known actions  ['d', 'a', 'b', 'c']
self known states  [0, 1, 2]
minimal False
this is the BP:
acts:{0: {'d': 1, 'a': 1}, 1: {'b': 0}, 2: {'c': 0}}
rec:{0: {'d': (1, True), 'a': (1, True), 'b': (1, True), 'c': (1, True)},
     1: {'d': (0, True), 'a': (1, True), 'b': (0, True), 'c': (0, True)},
     2: {'d': (0, True), 'a': (0, True), 'b': (1, True), 'c': (0, True)}}
SMT values constrains
Are the two BPs are equivalent?: (None, None, True)
```

As we expected, the minimal and the `defaultive returned` BPs are of the same size (3), because for CS the algorithm guarantees it.

The csv file will look as follows: |failed_converged | timeout | amount_of_states_in_origin | amount_of_states_in_output | origin_BP | output_BP | cutoff | CS_development_time | CS_positive_size | CS_negative_size | words_added | longest_word_in_CS | solve_SMT_time | right_output | |——— |——-|——-|———-|———-|——-|——-|——-|——-|——-|——-|——-|——-|——-|
|FALSE|FALSE|3|3|“states: 3,actions: {0: {‘a’: 1, ‘d’: 1}, 1: {‘b’: 0}, 2: {‘c’: 0}},initial: 0,receivers: {0: {‘a’: 1, ‘b’: 2, ‘c’: 1, ‘d’: 1}, 1: {‘a’: 0, ‘b’: 0, ‘c’: 1, ‘d’: 0}, 2: {‘a’: 2, ‘b’: 2, ‘c’: 2, ‘d’: 0}}”|“states: 3, actions: {0: {‘d’: 1, ‘a’: 1}, 1: {‘b’: 0}, 2: {‘c’: 0}},initial: 0,receivers: {0: {‘d’: 1, ‘a’: 1, ‘b’: 1, ‘c’: 1}, 1: {‘d’: 0, ‘a’: 1, ‘b’: 0, ‘c’: 0}, 2: {‘d’: 0, ‘a’: 0, ‘b’: 1, ‘c’: 0}}”|2|0.000421|16|28|{‘positive’: {}, ‘negative’: {}}|5|0.0233275|(None, None, True)|

Example 2:

We can also do so for a given BP, written according to our structure The following BP B_1 is written as follows:

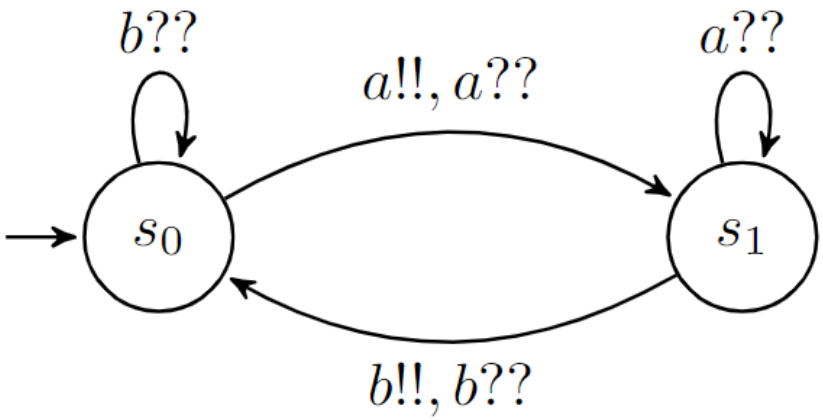

Figure 5: image

```
bp1 = BP_class(2, {0: {'a': 1}, 1: {'b': 0}}, 0, {0: {'a': 1,
'b': 0}, 1: {'a': 1, 'b': 0}})
```

I.e., it has 2 states, action a is broadacsted from state 0 and landed in state 1, and action b is broadacsted from state 1 and land on state 0. And from both states, the receiving transition a?? land on state 1 while the receiving transition on action b, which is b?? land on state 0.

A fucntion that do so is `run_a_given_bp_example(bp1)` for a given bp

And creating a CS and inferring for it will be:

```
SAT
self known actions  ['a', 'b']
self known states  [0, 1]
minimal False
this is the BP:
acts:{0: {'a': 1}, 1: {'b': 0}}
rec:{0: {'a': (1, True), 'b': (0, True)}, 1: {'a': (0, True), 'b': (0, True)}}
SMT values constrains
Are the two BPs are equivalent?: (None, None, True)
```

About Broadcast Protocols:

Broadcast protocols (in short BPs) are a powerful concurrent computational model, allowing the synchronous communication of the sender of an action with an arbitrary number of receivers.

The basic model assumes that communication and processes are reliable, i.e., it does not consider communication failures or faulty processes. BPs have mainly been studied in the context of parameterized verification, i.e., proving functional correctness according to a formal specification, for all systems where an arbitrary number of processes execute a given protocol.

The challenge in reasoning about parameterized systems such as BPs is that a parameterized system concisely represents an **infinite family** of systems: for each natural number n it includes the system where n indistinguishable processes interact. The system is correct only if it satisfies the specification for any number n of processes interacting.

Formal definition - BP

A **broadcast protocol** $B = (S, s_0, L, R)$ consists of a finite set of states S with an initial state $s_0 \in S$, a set of labels L and a transition relation $R \subseteq S \times L \times S$, where $L = \{a!!, a?? \mid a \in A\}$ for some set of actions A. A transition labeled with $a!!$ is a broadcast **sending transition**, and a transition labeled with $a??$ is a broadcast **receiving transition**, also called a **response**.