ARBAC analyser

Giacomo Rosin (875724)

April 13, 2022

1 Introduction

The goal of the assignment was to build an ARBAC (Administrative Role Based Access Control) analyser for small policies, able to parse a role reachability problem specification and return its solution (true or false).

2 Usage

The ARBAC analyser can be used as follows:

```
python3 arbac-analyser.py [policy.arbac]
```

When the path to a policy is given as parameter, the analyser will read the specified file:

```
python3 arbac-analyser.py ./policies/policy1.arbac
```

When no path is provided, the analyser will read the ARBAC specification from standard input:

```
cat ./policies/policy1.arbac | python3 arbac-analyser.py
```

Note that this program need a version of Python greater or equal to 3.8, and the used third party packages (only `lark`) need to be installed. You can obtain them by running:

```
pip install -r requirements.txt
```

3 Implementation details

3.1 Parsing

The ARBAC reachability problem specifications are parsed using a context-free grammar written using Lark (a parsing toolkit for Python), that uses

EBNF-inspired grammars. If the parsing is successful, a parse tree is generated. By defining a subclass of a `lark.Transformer` you can transform the parse tree into a data structure of your choice. The grammar for the ARBAC role reachability specification language is defined in the `arbac_analyser/parser/arbac.lark` file, and the related Python code is in the `arbac_analyser/parser/arbac_parser.py` file.

The data structures used to store the parsed information are defined in the `arbac_analyser/types/arbac.py` file. There is only one thing worth noting about data structures. The user-to-role assignment is modelled through a set, in particular an immutable set, of user-to-role associations. A set is used, and not a list, since the ordering of the user-to-role association is not important, and thus the semantic of the == operator is the one wanted (same elements contained, regardless of the order). Other more efficient data structures could be used, but set was the simpler one. An immutable set (a Python "`frozenset`") is used because, later, the set of user-to-role assignments needs to be put into another set (the "visited" set), for the role reachability algorithm (normal set can't be stored in another set).
The other data structures are quite straightforward to understand.

3.2 Pruning algorithms

Role reachability is a PSPACE-complete problem. So you need to use some technique to deal with complex ARBAC policies. To simplify the input policies, some pruning algorithm have been implemented. Pruning algorithms permit to reduce the search space, without restricting the ARBAC model, and without relying on approximate analysis techniques. The pruning algorithms implemented are the following:

- *Forward slicing*: Computes an over-approximation of the reachable roles, and then simplifies the ARBAC system according to it, in a way to preserve the solution to the role reachability problem.
- *Backward slicing*: Computes an over-approximation of the relevant roles to assign the goal, and then simplifies the ARBAC system according to it, in a way to preserve the solution to the role reachability problem.
- *Slicing*: Applies repetitively the forward slicing algorithm, followed by the backward slicing algorithm, until the ARBAC system stabilises to a fixed point.

3.3 Role reachability

The algorithm implemented to solve role reachability searches all the possible user-to-role assignments of the given ARBAC system, stopping when no new states are found, returning false, or when a state that contains the goal role is found, returning true. To visit all the state space, the implemented algorithm

keeps a queue of user-to-role assignments to analyse and a set of visited user-to-role assignments (set has been choosen since the Python implementation can check the presence of an item in O(1) on average, while still keeping the insertion time O(1) on average). At the beginning, the queue contains only the initial state. At every iteration over the queue, an user-to-role assignment is extracted and, if it was not already visited, the presence of a user with the goal role is tested. Then if it is not present, the state is added to the list of visited states and all the possible reachable states are generated, and added to the queue. To generate all the possible new user-to-role assignments reachable from the current user-to-role assignment, for each pair target user and policy rule (can assign or can revoke) a new user-to-role assignment is generated if the preconditions to apply the rule are met.

4 Results

In the following table are listed the solutions to the 8 reachability problems (`policies/policy{1..8}.arbac`), along with the time and the memory needed to solve them.

Policy	**Result**	**Time (s)**	**Memory (Mb)**
Policy 1	Reachable	0.30	39.044
Policy 2	Not reachable	28.28	215.352
Policy 3	Reachable	0.09	17.916
Policy 4	Reachable	0.40	50.336
Policy 5	Not reachable	390.06	1492.268
Policy 6	Reachable	0.10	20.400
Policy 7	Reachable	0.29	32.560
Policy 8	Not reachable	379.49	1492.440

Table 1: Analysis of the 8 policies

Time and memory usage measured using Gnu time (`/usr/bin/time`) and averaged over 3 runs on an Intel i5-8400.