The SDMLib Solution to the TTC 2017 State Elimination Case

Alexander Weidt
Kassel University
alexander.weidt@uni-kassel.de

Albert Zündorf
Kassel University
zuendorf@uni-kassel.de

1 Introduction

The Transformation Tool Contest 2017 State Elimination Case [SECase] asks for the reduction of a Finite State Automaton into an equivalent regular expression. Basically, one replaces states with transitions that combine the regular expressions attached to incomming and outgoing transitions of the removed state. This paper shows the SDMLib [SDMLib] solution to this case.

2 The rules

At first, the example automaton has to be normalized: The automaton must have exactly one inital state and exactly one final state.

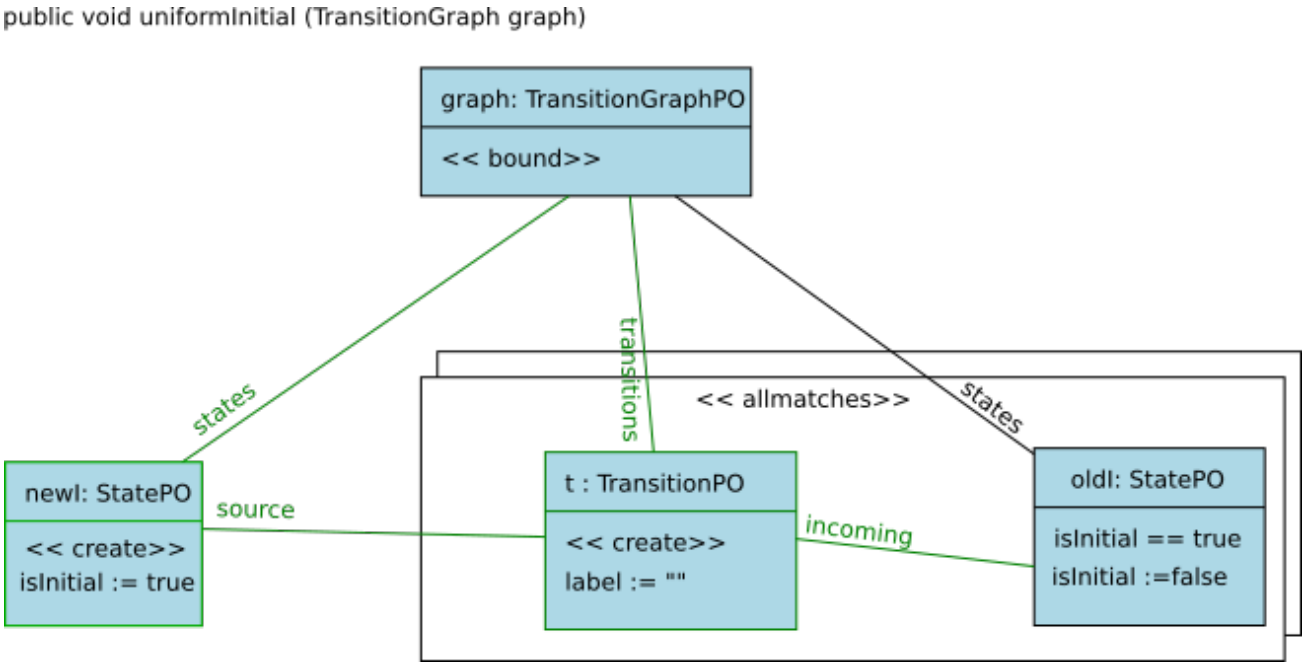

Figure 1: Adding single intial state

Figure 1 shows in graphical notation the SDMLib transformation rule that adds a new initial state to the current automaton and connects this new state to all previous initial states with empty transitions.

In: A. Editor, B. Coeditor (eds.): Proceedings of the XYZ Workshop, Location, Country, DD-MMM-YYYY, published at http://ceur-ws.org

Transformation `uniformInitial()` starts by matching the pattern object `graph` to the `TransitionGraph` object passed as parameter. Next, pattern object `newI` creates a new initial state. Then the sub-pattern rendered within two stacked boxes is executed. The pattern object `oldI` searches for old initial `State` objects attached to our `TransitionGraph` via a `states` link and with an `isInital` attribute equal to `true`. The `<<allmatches>>` sterotype attached to the sub-pattern shown in Figure 1 causes the sub-pattern to be executed for each match of `oldI`. Thus, for each old initial state a new `Transition t` with empty label is created. The new transition connects the new initial state `newI` and the current old initial state. In addition, the `isInitial` attribute of `oldI` is set to `false`.

We use a similar rule called `uniformFinal()` to introduce a single final state into the current automaton. The case description also proposes to add theta transitions between all states that are not yet connected via a transition. Then the reduction algorithm is save to assume that each pair of states is connected via a transition. Similarly, multiple transitions between two states are combined into a single transition. This reduces the number of case distinctions within the algorithm but increases the runtime. Thus, the SDMLib approach does not perform these normalization steps but deals with these cases in its transformation rules.

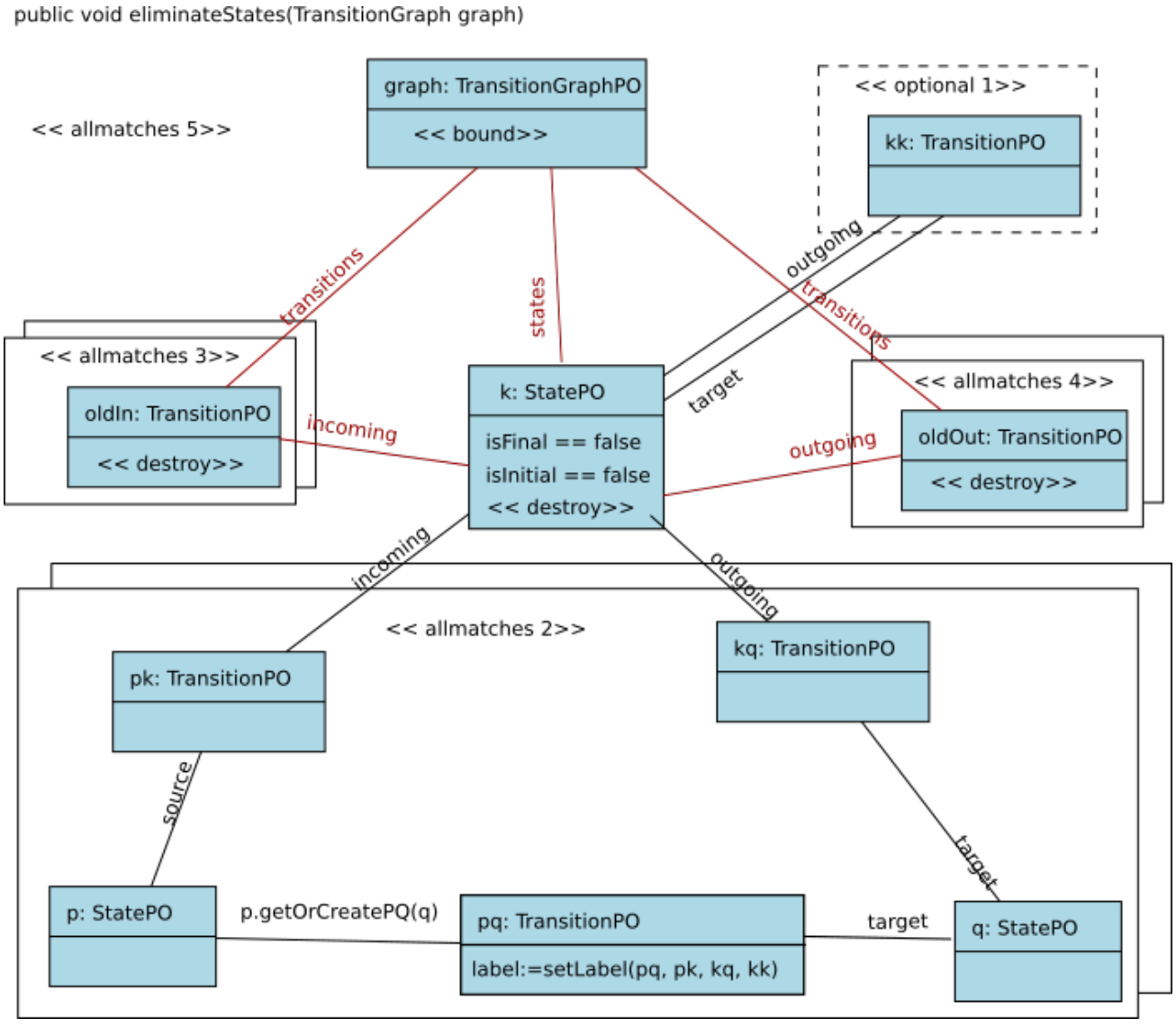

Figure 2: Eliminate States

Figure2 shows our `eliminateStates()` transformation. Again we start with a pattern object `graph` that matches the root of our finite automaton, passed as parameter. Next we look up `State k`, the state that shall be

replaced by transitions. `State k` shall neither be `isFinal` nor `isInitial`. Then, the sub-pattern `<<allmatches 2>>` looks up a `State p` that has a `Transition pk` going to `State k` and a `State q` that is reached from `State k` via `Transition kq`. The sub-pattern `<<allmatches 2>>` is iterated, that means it is executed for all possible matches. For each match, we call the pseudo path expression `p.getOrCreatePQ(q)`. This operation tries to find an existing `Transition pq` that connects the `States p` and `q`. The path expression creates such a `Transition`, otherwise. Operation `setLabel(pq, pk, kq, kk)` generates a label for transition `pq` computed as `pq.label + pk.label kk.label* kq.label`. The cases that `pq.label` is empty or that `kk` is null or that `kk.label` is empty are handled, specifically. Once all matches of sub-pattern `allmatches 2` are handled, sub-pattern `allmatches 3` and `allmatches 4` remove all old `Transitions` leading to or leaving `State k`, respectively. Finally, `State k` itself is destroyed. The whole top level pattern is iterated itself through the stereotype `<<allmatches 5>>`. Thus, transformation `eliminateStates` eliminates all states except the unique initial and final states introduced at the beginning. These initial and final states are then connected by exactly one `Transition` and this `Transition` has a regular expression as label that is equivalent to the original finite automaton.

3 Results

Figure 3 shows some measurements for our solution. We have executed the transformation on a laptop with about 1.6 Ghz. Intel I5 processor giving the Java process 6 GB heap space. We noticed that the evaluation of the regular expression uses a lot of time, thus we show our results with and without the time used for the evaluation of the resulting regular expression. We compare our results with the JFLAP algorithm times reported in the call for solution. We are waiting for the results of the other teams.

	SDMlib mit Evaluation	SDMlib ohne Evaluation	JFLAP
leader3_2	0.0385514360	0.052127528	0.09
leader4_2	0.0866275960	0.081792921	0.14
leader3_3	0.0993657960	0.063173148	0.49
leader5_2	1.1262002710	0.239402017	3.46
leader3_4	0.1605883720	0.098499445	4.37
leader3_5	0.5394560420	0.294551276	58.6
leader4_3	35.3694409170	7.921832035	57.78
leader6_2	24.6645100420	3.029549227	143.12
leader3_6	0.9771501780	0.771568491	461.64
leader4_4	8452.815857866	too big for memory	timeout

Figure 3: Execution Times (in seconds)

Overall, the solution was quite straight forward although our state elimination transformation needs quite a number of sub-pattern and we use some helper methods to handle different cases, uniformly.

References

[SDMLib] SDMLib - Story Driven Modeling Library *www.sdmlib.org* May, 2017.

[SECase] Sinem Getir, Duc Anh Vu, Francois Peverali, and Timo Kehrer State Elimination as Model Transformation Problem *www.transformation-tool-contest.eu/TTC_2017_paper_4.pdf* May, 2017.