Transformation of Finite State Automata to Regular Expressions using Henshin

Daniel Strüber

University of Koblenz and Landau
strueber@uni-koblenz.de

Abstract

We present a solution to the State Elimination Case of the Transformation Tool Contest 2017, based on the Henshin model transformation language. The main task is to convert a finite state automaton (FSA) into a regular expression; two extensions include the simplification of FSAs as well as the conversion of probabilistic FSAs. The distinguishing feature of our solution is its largely declarative specification, based on Henshin's concepts of rules and composite units for specifying modifications and control flow. We present the results of the performance and scalability evaluation based on the provided benchmark suite. Similar to the reference solution, our solution did not scale up to the largest test models in the benchmark suite; yet it achieved a speed-up by two magnitudes for some of the larger input models.

1 Introduction.

Finite state automata (FSAs) are a modeling concept with many practical applications, including program analysis, pattern matching, speech recognition and behavioral software modeling. In its basic form, a FSA comprises a set of states and a set of labelled transitions between the states; some of the states are designated as initial and final states. An important feature of FSAs is their fundamental relationship to regular expressions. While converting a regular expression into a corresponding FSA is easy, the opposite task is more sophisticated; current solutions suffer from scalability limitations [GVPK17].

The TTC 2017 *state elimination* case [GVPK17] aims to study how transformation tools may contribute to more efficient solutions. The case description specifies three tasks—a main task and two extensions—dealing with three kinds of FSAs: (i) *basic* FSAs with at least one initial and at least one final state, (ii) *simple* FSAs with exactly one initial and exactly one final state, and (iii) *probabilistic* FSAs, in which transitions are annotated with labels and probabilities. Simple FSAs are a subset of basic ones, whereas probabilistic FSAs generalize basic ones. The main task is to transform a simple FSA to a regular expression. Extension 1 involves transforming a basic to a simple FSA, and extension 2 deals with transforming a probabilistic FSA to a stochastic regular expression.

In this paper, we present a complete solution based on the Henshin model transformation language [ABJ+10]. Henshin is a graph-based transformation language providing support for the declarative specification of in-place model transformations. The basic features of Henshin's tool set are a suite of editors and an interpreter kernel; more sophisticated features include code generation for parallel graph pattern matching and support for various transformation analyses.

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

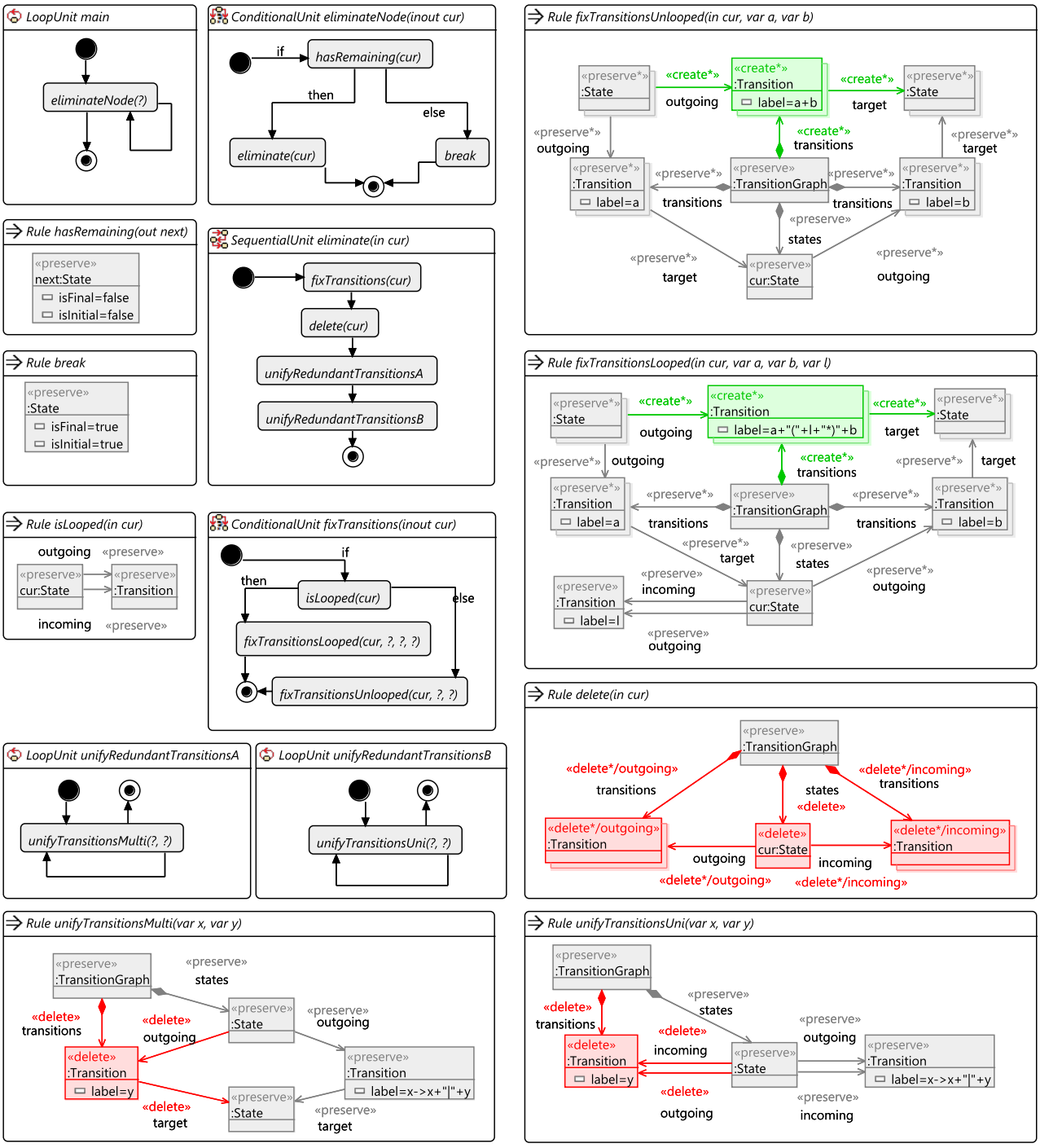

Figure 1: Solution for main task: state elimination in FSAs.

A distinguishing feature of Henshin is its expressive visual syntax which aims to facilitate usability during transformation development [SBG+17]. To provide a largely declarative solution, we use Henshin's concepts for the specification of control flow and model modifications. Control flow is specified using *composite units* that orchestrate the execution of a set of sub-units and rules, for example, by applying them in sequential order or in a counted loop. Model modifications are specified using *rules*, expressing basic match-and-change patterns.

2 Solution

In what follows, we present our solution in detail. We start with the main task and continue with the two extension tasks. The solution is available via GitHub at `https://github.com/dstrueber/stateelim-henshin`. The URL of the associated SHARE image is made available on the GitHub site.

2.1 Main Task: Converting FSAs to Regular Expressions

Our solution to the main task follows the state elimination algorithm presented in [GVPK17]. We implemented this algorithm using 8 rules, which are orchestrated in a control flow of 6 composite units, as shown in Fig. 1.

The overall goal is to remove all non-initial and non-final states from the FSA. To achieve this goal, our starting point is the loop unit *main* which executes its inner unit *eliminateNode* as often as possible. The aim of *eliminateNode* is twofold: first, it checks whether any state is to be removed and, if this is the case, identifies it. This is done using the rule *hasRemaining*, a simple rule for matching a non-initial and non-final state. Second, if such state is found, it is stored in the parameter *cur* (for *current*) and passed to the sequential unit *eliminate*. The *eliminate* unit proceeds in the following steps: first it "fixes" the incoming and outgoing transitions of state *cur* by replacing them, then it deletes *cur*, and finally, it unifies any redundant transitions arising in the process.

To fix the transitions, conditional unit *fixTransitions* first checks if *cur* is a looped state, that is, a state with a loop transition. Depending on the result, one of the rules *fixTransitionsUnlooped* and *fixTransitionsLooped* is triggered, which only differ in their treatment of the loop. Both rules make use of rule-nesting: they have a *kernel rule*—the "flat" states in the visual syntax—and a *multi-rule*—the "layered" states with asterisk signs. When applying either rule to the input model, the kernel rule is matched first, and then, based on the identified match, the multi-rule is applied with a *for-all* semantics, that is, as often as possible. Based on the provided state *cur*, all pairs of incoming and outgoing transitions are identified, and a new transition is created for each of these pairs. The label of the newly created transition is composed of the labels of the pair (plus in the loop case, the loop label), which are stored and propagated using the variables *a*, *b* (and *l*). Rule *delete* works in a similar manner by deleting state *cur* via the kernel rule, and its adjacent transitions via separate multi-rules.

As a result of the *fixTransitions* step, the model may temporarily contain multiple transitions between the same pair of states. These redundant transitions are now unified, using separate loop units for cases where the transitions are non-loops and loops, respectively. Each of these units calls a rule which identifies redundant pair of transitions, removes one of the transitions, and joins its label with the label of the remaining transition.

The outer *main* unit terminates when there are no remaining nodes to be eliminated. To escape from the loop, a *break* rule is required, which specifies an impossible pattern (here, a node that is initial and final at the same time), so it always evaluates to *false*. After the whole process, there is only the initial and the final state left, possibly with various transitions between them. To obtain the final regular expression, a short Java routine puts together their labels according to Listing 4 in [GVPK17].

2.2 Extension 1: FSA simplification.

The conversion of a basic FSA into a simple one involves two steps: first, if there are multiple initial states, these states become non-initial states with an incoming ϵ-transition from a newly created unique initial state. Second, if there are multiple final states, these states become non-final states with

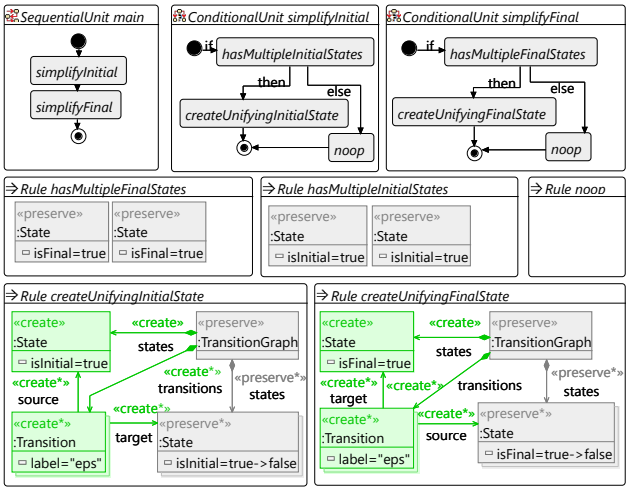

Figure 2: Solution for extension 1: Simplifying a FSA.

an outgoing ϵ-transition to a newly created final initial state. Our solution represents these steps via two sub-units *simplifyInitial* and *simplifyFinal*, which are applied in sequential order using a sequential *main* unit.

The *simplifyInitial* unit uses a rule *hasMultipleInitialStates* to check if the simplification treatment becomes necessary, and, if, this is the case, performs the simplification using the rule *createUnifyingInitialState*. Specifically, rule *hasMultipleInitialStates* checks whether two separate initial states exist in the input FSA. Rule *createUnifyingInitialState* again uses the concept of rule-nesting to achieve a for-all semantics. The kernel rule specifies the creation of an additional initial state. The multi-rule matches all earlier initialState, turns them into non-initial states and creates an incoming ϵ-transition for each of them.

The treatment of final states based on unit *simplifyFinal* is completely dual.

2.3 Extension 2: Converting probabilistic automata.

The solution for extension 2 extends the solution for the main task with small modifications concerning the treatment of labels and probabilisties. Since the case description does not specify how the labels and probabilities are to be computed (in fact, the probabilities in the computed regular expression are ignored in the correctness evaluation), we followed the specification of the reference implementation. Details about the reference implementation's specification were obtained in our communication with the case authors (see `https://github.com/sinemgetir/state-elimination-mt/issues/3`).

According to this specification, the conversion process includes a preprocessing of the input automaton where the probabilities of the outgoing transitions for each state are recalculated, such that (i) loop, ϵ, and empty transitions are ignored, (ii) the probabilities of the other transitions are added up to derive "the new 100%", and (iii) each transition obtains its original probability divided by "the new 100%" as its new probability. Fig. 3 shows our transformation for implementing this specification: Unit *recalculateProbs* specifies that its two contained rules are applied in sequential order. Rule *recalculateProbsTemps* uses rule nesting to iterate over all outgoing non-loop/empty/ϵ transitions of all states, so that their probabilities are stored in variable a. The sum of all values of a is stored in a newly created `Trace` object, using Henshin's `Aggregations` helper class to compute the sum. Rule *recalculateProbsUpdate* performs the same iteration as before to update the probabilities of the involved transitions. Given the old probability b, the new value is b/a, where a is the aggregate percentage stored in the `Trace` object. In this process, the `Trace` objects become obsolete and are consequently deleted.

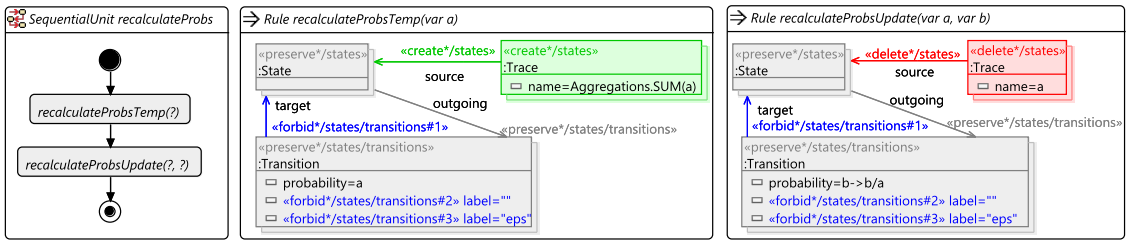

Figure 3: Solution for extension 2: pre-processing step.

The remaining conversion process is the same except for the label and probabilities handling, where we adhere to the specification: *"when we concatenate two labels we multiply their probabilities. When we 'or' two labels, we add their probabilities."*

For example, in the rule *fixTransitionsLooped*, for the transition beingly newly created, the probability needs to account for the probabilities of the original transitions, and the label needs to represent the probability of the loop transition. To this end, we modified this rule as shown in Fig. 3. During the matching process, we now store the labels as well as the probabilities of the matched transition in variables ($\{a, b, l\}$ and $\{pa, pb, pl\}$, respectively). In the multi-rule, we then use these variables in the label and probability of each newly created transition. The label includes the probability of the loop transition, the probabilities is made up from the probabilites of the input transitions.

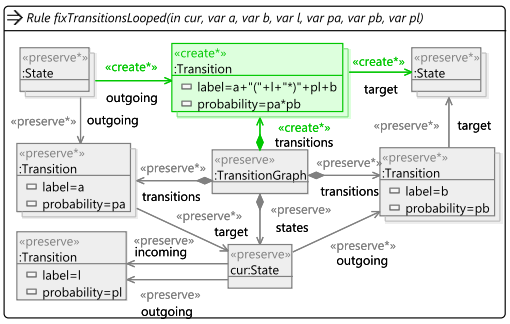

Figure 4: Solution for extension 2: excerpt.

3 Evaluation

In this section, we apply the evaluation criteria from the case description to our solution. The performance experiment was performed on a Windows 10 system (Intel Core i7-5600U, 2.6 GHz; 8 GB of RAM, Java 1.8 with 2 GB Xms). In the scalability evaluation, we ran the experiment with a timeout duration of max. 1 hour per model (following the reference solution, see `https://github.com/sinemgetir/state-elimination-mt/issues/4`).

Correctness. When applied to the provided test models, our solution produced correct results in all cases. For the main task and extension 2, we used the provided benchmark framework for verifying our solution. We slightly extended the framework so it supports the execution against user-specified timeout durations.

For extension 1, unfortunately, we could not follow the evaluation process described in the case description. By the time of submission, the FSA data structure used by the reference implementation does not support input FSAs with more than one initial state, rendering it infeasible as a baseline for the correctnes check (see `https://github.com/sinemgetir/state-elimination-mt/issues/2#issuecomment-304011132`). Instead, we perform a light-weight correctness check to validate if the changes to the numbers of states, initial states, final states, and transitions during the transformation are in line with our expectations.

Suitability. With our solution, we aimed at providing a primarily declarative solution. We achieved this goal by specifying all parts of the state elimination algorithm (Listing 2 and 3 in [GVPK17]) using Henshin's declarative rule and control flow concepts. In addition, our solution includes two minor imperative parts, written in Java: A driver to trigger the execution of the Henshin interpreter with the specification, and a part to convert the output of state elimination to an expression (Listing 4 in [GVPK17]).

Performance. Table 1 shows the performance measurements in comparison to the reference solution (focusing on cases where a comparison is possible). For the three smallest test models, we observed a slow-down. For all of the remaining models, we observed an increasingly growing speed-up, amounting to an order of two magnitudes in the case of the largest input model *leader4_4*.

Scalability. The largest case where our solution produced a result within one hour was *leader4_5*, taking 10:12 minutes for 1933 states. Given more time, we can transform larger models as well, e.g., *leader5_4* in 120:41 minutes for 4244 states. For reference, the largest model handled by the reference solution had 812 states.

	Execution time (sec.)	
Model	**Reference**	**Henshin**
leader3_2	0.09	**0.21**
leader4_2	0.14	**0.25**
leader3_3	0.49	**0.29**
leader5_2	3.46	**0.78**
leader3_4	4.37	**0.77**
leader3_5	58.60	**2.93**
leader4_3	57.78	**2.32**
leader6_2	143.12	**3.45**
leader3_6	461.64	**10.09**
leader4_4	4786.58	**48.15**

Table 1: Performance measurements.

4 Outlook

Conceptually, the state elimination case is a highly interesting scenario for our ongoing work on the variability of model transformations [SRA+16]. It features variability in two dimensions: variability in the language of the input automata (plain and probabilistic FSAs), and variability in the solution artifacts of the transformations (repeatedly, we handle a *looped* case differently from an *unlooped* case). We intend to use this scenario as a case study for extending our work towards language-level variability.

References

[ABJ+10] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: advanced concepts and tools for in-place EMF model transformations. *Model Driven Engineering Languages and Systems (MoDELS)*, pages 121–135, 2010.

[GVPK17] Sinem Getir, Duc Anh Vu, Francois Peverali, and Timo Kehrer. State Elimination as Model Transformation Problem. *10th Transformation Tool Contest (TTC)*, 2017.

[SBG+17] Daniel Strüber, Kristopher Born, Kanwal Daud Gill, Raffaela Groner, Timo Kehrer, Manuel Ohrndorf, and Matthias Tichy. Henshin: A Usability-Focused Framework for EMF Model Transformation Development. In *International Conference on Graph Transformation (ICGT)*, 2017. accepted.

[SRA+16] Daniel Strüber, Julia Rubin, Thorsten Arendt, Marsha Chechik, Gabriele Taentzer, and Jennifer Plöger. RuleMerger: automatic construction of variability-based model transformation rules. In *Int. Conf. on Fundamental Approaches to Software Engineering (FASE)*, pages 122–140, 2016.