State Elimination as Model Transformation Problem

Sinem Getir
Software Engineering, Humboldt University Berlin
getir@informatik.hu-berlin.de
Duc Anh Vu
Software Engineering, Humboldt University Berlin
vuducanh@informatik.hu-berlin.de
Francois Peverali
Model-driven Software Development, Humboldt University Berlin
peveralf@informatik.hu-berlin.de
Daniel Strüber
Institute for Computer Science, University of Koblenz and Landau
strueber@uni-koblenz.de
Timo Kehrer
Department of Computer Science, Humboldt University Berlin
timo.kehrer@informatik.hu-berlin.de

Abstract

State elimination has been proposed in the literature as a viable technique for transforming finite state automata (or finite state machines) into equivalent regular expressions. In this TTC case, we consider this well-known technique as a model transformation problem, aiming at evaluating the suitability, performance and scalability of dedicated model transformation techniques w.r.t. this problem.

1 Introduction

Finite state automata (or finite state machines) are used in many applications such as program analysis, data validation, pattern matching, speech recognition as well as in the behavioral modeling of systems and software. Furthermore, quantitative extensions of automata are used to evaluate system behavior together with software profiles in software engineering. Such models are, like Markov chains, probabilistic automata, adding probabilistic distributions to state transitions [BKL08]. Even though the semantically equivalent counterparts of (probabilistic) finite state automata, namely (stochastic) regular expressions, are generally not considered as behavioral models, they may be used in the aforementioned applications interchangeably with finite state automata. In practice, regular expressions are often transformed into finite state automata, e.g. to obtain models from code, to extract behavioral models at runtime, etc. While this is not an expensive task, the transformation of a (probabilistic) finite state automaton into an equivalent (stochastic) regular expression is much more expensive and challenging.

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

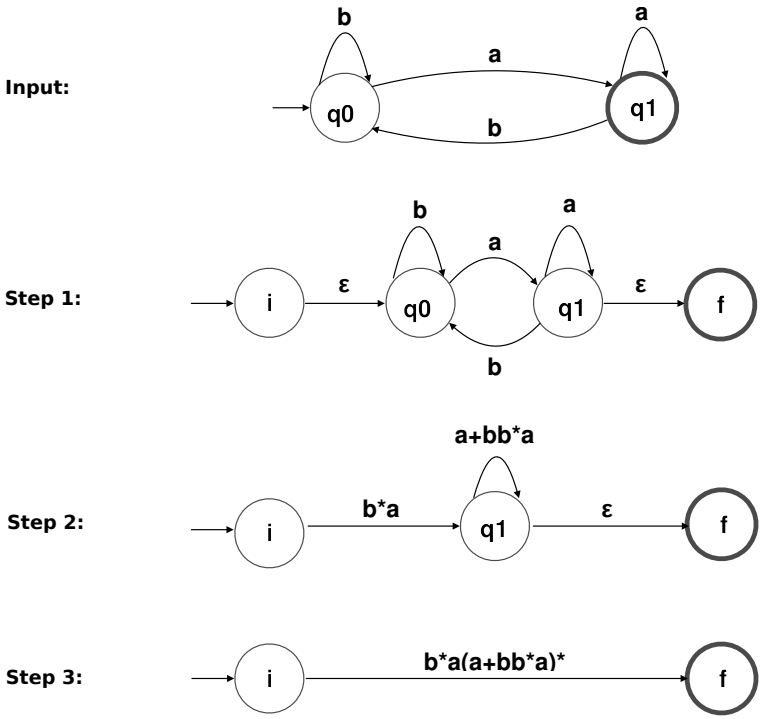

Figure 1: State Elimination example.

State elimination [DK01] has been proposed in the literature as a viable solution for doing this kind of transformation. In this TTC case, we consider this well-known technique as a model transformation problem, aiming at evaluating the suitability, performance and scalability of dedicated model transformation techniques w.r.t. this problem.

Section 2 briefly recalls basic principles of state elimination and presents a reference implementation based on a commonly used software package for experimenting with formal languages. Section 3 rephrases the involved problems as model transformation problems and provides the respective task descriptions, some of which are being considered as optional extensions to the main task of doing state elimination. Section 4 presents a set of quality characteristics to evaluate proposed solutions. All resources provided along with the case can be obtained from `https://github.com/sinemgetir/state-elimination-mt/`

2 Case Description

In this section, after recalling some basic definitions, we describe the state elimination algorithm, a reference implementation using JFLAP [PPR96], and our probabilistic implementation that is an extension to JFLAP in detail.

2.1 State Elimination

In the following, let Σ be a finite set of symbols, called alphabet, and Σ^* the words over Σ. A regular expression α over Σ represents a regular language $L(\alpha) \subset \Sigma^*$ for which, according to Kleene's theorem, there exists a finite state automaton accepting this language. An FSA is a tuple $(Q, \Sigma, \delta, q_0, F)$ where Q is a finite set of states, Σ is an alphabet, $\delta : Q \times \Sigma \to Q$ is a transition function, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is a set of final states. We say that an FSA is *uniform* if it has has a unique initial state with no incoming transitions and a unique final state with no outgoing transitions. Uniforming an FSA may be performed as follows:

- If $q_0 \in F$ or if there exists an $a \in \Sigma$ and $q \in Q$ so that $\delta(q, a) = q_0$, then add a new state i to Q with $\delta(i, \epsilon) = q_0$ and set i as the new initial state instead of q_0.
- If $|F| \geq 1$ or if there exists $p \in F$, $a \in \Sigma$ and $q \in Q$ so that $\delta(p, a) = q$, then add a new state f to Q, add the transition $\delta(p, \epsilon) = f$ for all $p \in F$ and set $F = \{f\}$.

The state elimination algorithm [DK01] takes a FSA as input and calculates a regular expression describing its accepted language as output. Given an FSA, state elimination works on its unified form. The idea is to iteratively eliminate states which are neither initial nor final. In each step, a new equivalent generalized transition graph (GTG) is created until only the initial and final states are left. A GTG generalizes an FSA in that transitions may be labelled by regular expressions instead of single symbols.

Let $\mathcal{R}(\Sigma)$ be the universe of regular expressions over a given alphabet Σ, then the transition function is generalized to $\delta : Q \times \mathcal{R}(\Sigma) \to Q$. Roughly speaking, regular expressions used as transition labels are utilized to keep track of eliminated states. The regular expression labelling the transition between the two states that remain at the end of the iteration represents the final output of the algorithm.

In Algorithm 1, let α_{xy} be the regular expression for the transition from state x to state y. Moreover, given an FSA $A = (Q, \Sigma, \delta, q_0, F)$, we refer to its unified form as $A' = (Q', \Sigma, \delta', i, \{f\})$ which we will interpret as GTG during state elimination.

Algorithm 1 State elimination

Input: A uniform FSA A'.
Output: A regular expression defining the regular language accepted by A'.

1: **repeat**
2: Randomly choose a state $k \in Q' \setminus \{i, f\}$ and remove it from A'.
3: Then for all pairs $p, q \in Q' \setminus \{k\}$ the new regular expression α'_{pq} for the transition from p to q is:

$$\alpha'_{pq} = \alpha_{pq} + \alpha_{pk}\alpha^*_{kk}\alpha_{kq}$$

4: **until** There are only the initial state i and the final state f left.
5: **return** α_{if}

Based on Algorithm 1, a small example that demonstrates how to convert an FSA into a regular expression is presented in Figure 1. Step 1 uniforms the given FSA, while steps 2 and 3 showcase the iterative elimination of states.

2.2 Reference Implementation

We use the common software package JFLAP [PPR96] to transform an FSA into an equivalent regular expression using the state elimination method. The difference between Algorithm 1 and the JFLAP implementation is that, for the latter one, the initial state can have incoming transitions and the final state can have outgoing transitions. Considering that, the transformation can be divided into three steps:

(1) Conversion of the given FSA into a simple FSA: A simple FSA is an FSA which is uniform and, in addition, for all pairs of states there is exactly one transition. The simple automaton is created in the following steps:

- If the automaton has more than one final state or the initial state is a final state, we create a new state s and for all final states f, add an $\epsilon - transition$ for $f \longrightarrow s$ and set s as the only final state. Note that in JFLAP $\epsilon - transitions$ are defined as $\lambda - transitions$ with the empty string as the label.
- If there are several transitions between two states, we combine them into a single transition; the new label is obtained as a disjunction of all transition's labels.
- If there are no transitions between two states, we add an empty transition between them. An empty transition is a transition with the label $\emptyset$.

(2) Conversion to a GTG. Converting a simple automaton obtained from step (2) into an equivalent GTG with only the initial state and the final state is described by Algorithm 2, which in turn uses the subroutine described in Algorithm 3 in order to obtain a regular expression for a state to be eliminated.

(3) Deriving the regular expression. We now have an equivalent GTG with exactly two states, the initial and the (different) final state. Algorithm 4 describes how we get the final regular expression.

Algorithm 2 convertToGTG

Input: A simple automaton A.
Output: An equivalent GTG A' with only the initial state and the final state.

```
for k : A.getStates() do
    newTransitions ← []
    if k is not initial or final then
        for p, q : A.getStates() do
            if p ≠ k and q ≠ k then
                pq ← getExpressionForElimination(k, p, q, A)
                newTransitions.add(pq)
        remove k and all its incoming and outgoing transition from A
        A.add(newTransitions)
```

Algorithm 3 getExpressionForElimination

Input: The state k to be eliminated, the state p that reaches k, the state q that is reached from p via k, the automaton A.
Output: The regular expression for the transition from p to q without k.

```
pq ← the regular expression for the transition from p to q
pk ← the regular expression for the transition from p to k
kk ← the regular expression for the transition from k to k
kq ← the regular expression for the transition from k to q
return (pq) + (pk)(kk)*(kq))
```

Algorithm 4 deriveFinalExpression

Input: The GTG A' with only the initial state and the final state.
Output: An equivalent regular expression for A'.

```
i ← A'.getInitialState()
f ← A'.getFinalState()
ii ← the regular expression for the transition from i to i
if ← the regular expression for the transition from i to f
ff ← the regular expression for the transition from f to f
fi ← the regular expression for the transition from f to i
return ((ii)*(if)(ff)*(fi))*(ii)*(if)(ff)*
```

2.3 Extension to Probabilistic Automata

The case of transforming an FSA into an equivalent regular expression may be extended to converting probabilistic (finite state) automata (PA) into stochastic regular expressions.

A PA is a tuple (Q, Σ, P, q_0, F) where Q, Σ, q_0 and F are defined analogously to FSAs, and $P : Q \times Q \times \Sigma \to [0, 1]$ is a transition probability function assigning a probability to each transition such that

- $\forall q \in Q \setminus F : \sum_{q' \in Q} \sum_{a \in \Sigma} P(q, q', a) = 1$, and
- $\forall q \in F : \sum_{q' \in Q} \sum_{a \in \Sigma} P(q, q', a) = 0$

We extend the state elimination algorithm to PAs and provide a reference implementation using JFLAP. When a PA is converted to a regular expression, this regex is also enriched with probabilities and rates, generally referred to as stochastic regular expression (SRE). We present the syntax briefly as follows:

$$E := \alpha \;\Big|\; \sum_i E_i[n_i] \;\Big|\; E_1 : E_2 \;\Big|\; E^{*f}$$

where $\alpha \in \Sigma \cup \{\epsilon\}$, $n_i \leq 1000 \in \mathbb{N}_0$, $f \in [0, 1] \subset \mathbb{Q}$ and E_i is a SRE:

- *Atomic Action* α: The action α is an atomic action.

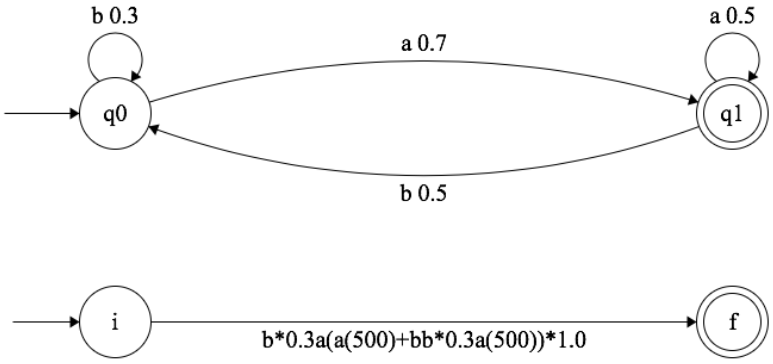

Figure 2: State Elimination example with probabilities.

- *Choice/Sum* $\sum_i E_i[n_i]$: One of the provided terms is randomly chosen. The n_i denotes the choice number for each term. The ith term is chosen with the probability $\frac{n_i}{\sum_j n_j}$.
- *Kleene Closure* E^{*f}: The term E is repeated random number of times. Each iteration occurs with a probability of f.The termination probability is $1 - f$.
- *Concatenation* $E_1 : E_2$: The terms E_1 and E_2 are successively interpreted. As the concatenation is associative, parenthesis may be omitted.

State elimination for PAs follows the same basic principles as for FSAs (see Section 2.2). The extended steps for the probabilistic version of the state elimination appears with the probability calculation. In the beginning, the probabilities of all outgoing transitions are recalculated where the probability of the loops remain unchanged. During the "label join", the probabilities are multiplied, remain unchanged and are uniformed to rates (e.g. Constant=1000) when expressions are concatenated, starred and choiced, respectively. During the recalculation of probabilities, we ignore loop transitions from a state to itself, epsilon and empty transitions. We recalculate the probabilities of the remaining outgoing transitions of each state using the follow steps: First, for all transitions, we add their probabilities to get the new "100%". Second, to get the new probability of each transition, we divide its old probability with the sum computed in the first step.

The reference implementation can be found in the resources provided along with this TTC case. Figure 2 showcases the probabilistic extension of state elimination by means of an example. Note that the probability of the last remaining transition is 1.0, so we omitted it in the figure.

3 Task Description

The basic idea of this TTC case is to rephrase the well-known problems sketched in Section 2 into dedicated model transformation challenges. A generic meta-model for transition graphs which may be interpreted as FSAs, CTGs and PAs is presented in Figure 3. The nodes of a transition graph represent the states while its edges represent the transitions of an FSA, CTG or PA. States are attached unique identifiers. Transitions may be labelled (carrying either symbols or regular expressions) and, in case of PAs, equipped with probabilities. Since virtually all model transformation tools available in the model transformation research community are based on the Eclipse Modeling technology stack, we provide an implementation of the meta-model presented in Figure 3 in EMF Ecore. Of course, other meta-modeling technologies are allowed in solutions as well. This requires a conversion of the input models serving as experimental data in terms of the evaluation (see Section 4), which we provide in EMF format.

3.1 Main Task

The main task is to implement a state elimination for FSAs as a model transformation. In this basic setting, we assume that the FSA given as input is uniform. Thus, the task boils down to iterative state removal until there

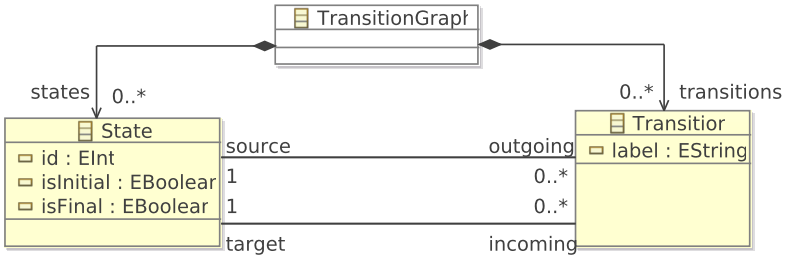

Figure 3: Generic transition graph meta-model for representing FSAs, CTGs and PAs.

is only the initial state and final state left, thereby updating regular expressions attached to transitions.

3.2 Extension 1

As a first additional yet optional task, we ask to uniform the FSM before applying state elimination. The task is to obtain a uniform FSA containing a unique initial state and a unique final state from a non-uniform FSA containing multiple initial and final states.

3.3 Extension 2

A second optional extension is to provide a model transformation taking a probabilistic finite state automation as input and producing a stochastic regular expression as output. Extension 2 can be considered independently from Extension 1.

4 Evaluation Criteria

To evaluate the quality of the proposed solutions, we give a set of quality characteristics which we consider to be relevant for the presented transformation case. The characteristics are inspired by the ISO/IEC 9126-1 standard, which defines quality characteristics in the broader scope of software engineering [BBC+04, JKC04]. We also draw inspirations from more recent work dedicated to the quality of model transformations [SG12, GSS14, LMT14]. We refine each quality characteristic into measurable attributes for each of the tasks presented in Section 3. To obtain concrete measures for their solutions, participants are kindly invited to use the evaluation framework provided with the case resources. This way, most of the measures may be obtained in an automated manner.

4.1 Correctness

A first important and rather obvious quality characteristic is the correctness of the provided solutions.

A transformation solving the main task (see Section 3.1) is correct if the regular expression obtained as a final result of the state elimination is correct. This means that it is semantically equivalent to the regular expression obtained from the reference solution using JFLAP presented in Section 2.2. Although semantically equivalence of regular expressions is decidable in general, we abstain from providing such an automated equivalence proof in our evaluation framework. We choose a testing approach instead, providing sets of positive and negative test cases, i.e., sets of strings to which the produced regular expression should match and should not match. This way, a result is considered to be correct if it passes all tests.

A solution to the first extension task (see Section 3.2) is correct if the produced model represents a uniform FSA which is equivalent to the one provided as input. This can be confirmed by checking equality of the resulting model to the one produced by a reference implementation in our evaluation framework. This correctness test may be automated using the model comparison tool EMF Compare [BP08], which should report no changes between a produced output model and the corresponding reference output model.

Correctness of solutions to the second extension task (see Section 3.3) can be checked in a similar way as for the main task. We use the same test cases as for the main task, ignoring probabilities of the stochastic regular expressions produced as output of the second extension task.

All tasks are scored by means of the provided test cases. A point is given for each test case which passes successfully, points are summarized over all test cases. This means that a correct solution for the main task may

get a maximum number of 11 points, while correct solutions to the first and second extension task may get a maximum of 3 and 9 points, respectively.

4.2 Suitability

Since this TTC case basically rephrases a set of well-known problems into a dedicated model transformation challenge, we are particularly interested in the suitability of the transformation specifications solving these problems. Since general quality metrics from traditional programming languages, such as complexity or modularity, are hardly generalizable for a set of different model transformation languages, we choose a rather simple qualitative criterion which can be referred to as the *level of abstraction*. We follow the approach presented in [LMT14]: The level of abstraction is classified as *High* for primarily declarative solutions, *Medium* for declarative-imperative solutions, and *Low* for primarily imperative solutions. All tasks presented in Section 3 are evaluated in the same way.

Since we acknowledge that classifying the abstraction level of a solution according to its declarativeness may be biased by subjective preferences, the scores that may be obtained for this quality characteristic are relatively low to not too much influence the overall results. All tasks and extension tasks are scored evenly with zero to three points. Zero means the task has not been tackled at all. If it has been solved, solutions may be scored with one, two, or three points, depending on the abstraction level ranging from low, medium, to high.

4.3 Performance

Performance is measured in terms of the *execution time* of a transformation solving the task. This includes the loading of input models and printing or serializing the produced output. The measuring of execution times can be automated by using the evaluation framework provided along with the case description. For the main task and each of its extensions, execution times should be measured for all input models already introduced in Section 4.1.

Performance results of a solution are scored by comparing the execution times with the other solutions. We consider the average execution time for the first ten input models (see *leader3_2* to *leader4_4* in Table 1). The fastest solution gets 14 points (maximum number of points for correctness and suitability of a solution to the main task). The remaining solutions get proportional scores w.r.t. their average execution times.

4.4 Scalability

Since the reference solution using JFLAP suffers from scalability problems when dealing with large input models (cf. Section 4.5), we are particularly interested in whether modern model transformation technologies can improve the scalability of the state elimination. In addition to the FSA models provided for evaluating correctness and scalability, we provide additional larger models comprising up to around one million states. Solutions should figure out which is the *largest model* they are still able to handle. Scalability is only considered for the main task presented in Section 3.1. As we can see in Table 1, the reference solution based on JFLAP scales up to model *leader4 4* containing 812 states.

Scalability is scored by comparing the solutions with each other. The solution which scales up to the largest input model gets a score of six points (analogously to performance). The rest of the solutions get scores which are proportional w.r.t. to the number of participating solutions.

4.5 Results for the Reference Solution

In this section, we present the evaluation results obtained for the reference solution using JFLAP. Please note that we give the results to better illustrate our evaluation criteria, while it is not intended to compare solutions using dedicated model transformation technologies with the reference solution, but mutually among themselves. We use models produced by the Synchronous Leader Election Protocol from the Prism Benchmark Suite [KNP12]. These models are Discrete Time Markov Chains (DTMC), but we interpret them as FSAs an PAs by using the target state names of transitions as labels. In case of FSAs, we ignore probabilities. For example, the DTMC triple (0,1,0.87) is interpreted as a transition from state *s0* to state *s1* assigned with the label "s1".

Table 1 summarizes our experimental results obtained for the reference solution using JFLAP. Participants in this case should use the same input models, all of them are available as EMF models conforming to the generic meta-model introduced in Section 3. In Table 1, sizes of input models are indicated by the number of states, execution times are reported in seconds. The used timeout duration was one hour per model. We ran all

Table 1: Evaluation results obtained for the reference solution using JFLAP. Sizes of input models are indicated by the number of states, execution times are reported in seconds.

	Correct	**Abstraction**	**Exec. Time (s)**	**Scalability**
model name (FSA uniform)				
leader3_2 (26)	yes	low	0,09	
leader4_2 (61)	yes	low	0,14	
leader3_3 (69)	yes	low	0,49	
leader5_2 (141)	yes	low	3,46	
leader3_4 (147)	yes	low	4,37	
leader3_5 (273)	yes	low	58,60	
leader4_3 (274)	yes	low	57,78	
leader6_2 (335)	yes	low	143,12	
leader3_6 (459)	yes	low	461,64	
leader4_4 (812)	yes	low	4786,58	leader4_4
leader5_3 (1050)	-	-	timeout	
model name (FSA non-uniform)				
zeroconf multiple initial states (1296)	yes	low		
zeroconf multiple final states (1296)	yes	low		
zeroconf multiple inital & final states (1296)	yes	low		
model name (Probabilistic Automata)				
leader3_2 (26)	yes	low		
leader4_2 (61)	yes	low		
leader3_3 (69)	yes	low		
leader5_2 (141)	yes	low		
leader3_4 (147)	yes	low		
leader3_5 (273)	yes	low		
leader4_3 (274)	yes	low		
leader6_2 (335)	yes	low		
leader3_6 (459)	yes	low		

experiments on a macOS with 1,6 GHz Intel Core i5 processor and 8 GB of memory (Java 1.8 with the maximum heap size of 2 GB).

Acknowledgments

This work was partially supported by the DFG (German Research Foundation) under the Priority Programme SPP1593: Design For Future – Managed Software Evolution.

References

[BBC+04] P Botella, X Burgués, JP Carvallo, X Franch, G Grau, J Marco, and C Quer. Iso/iec 9126 in practice: what do we need to know. In *Software Measurement European Forum*, volume 2004, 2004.

[BKL08] Christel Baier, Joost-Pieter Katoen, and Kim Guldstrand Larsen. *Principles of model checking*. MIT press, 2008.

[BP08] Cédric Brun and Alfonso Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, 9(2):29–34, 2008.

[DK01] Ding-Shu Du and Ker-I Ko. *Problem Solving in Automata, Languages, and Complexity*. John Wiley and Sons, 2001.

[GSS14] Christine M Gerpheide, Ramon RH Schiffelers, and Alexander Serebrenik. A bottom-up quality model for QVTO. In *Int. Conference on the Quality of Information and Communications Technology*, pages 85–94. IEEE, 2014.

[JKC04] Ho-Won Jung, Seung-Gweon Kim, and Chang-Shin Chung. Measuring software product quality: A survey of iso/iec 9126. *IEEE software*, 21(5):88–92, 2004.

[KNP12] M. Kwiatkowska, G. Norman, and D. Parker. The PRISM benchmark suite. In *Proc. 9th International Conference on Quantitative Evaluation of SysTems (QEST'12)*, pages 203–204. IEEE CS Press, 2012.

[LMT14] Kevin Lano, Krikor Maroukian, and Sobhan Yassipour Tehrani. Case study: Fixml to java, c# and c++. In *TTC@STAF*, pages 2–6, 2014.

[PPR96] M. Procopiuc, O. Procopiuc, and S. Rodger. Visualization and interaction in the computer science formal languages course with jflap. pages 121–125, 1996.

[SG12] Eugene Syriani and Jeff Gray. Challenges for addressing quality factors in model transformation. In *Int. Conference on Software Testing, Verification and Validation*, pages 929–937. IEEE, 2012.