A MontiCore Solution to the TTC 2017 State Elimination Case

Alex Lüpges[1], Janik Rapp[1] and Bernhard Rumpe[1]

[1]Software Engineering, RWTH Aachen University, Ahornstraße 55, 52074 Aachen, Germany

Abstract

This paper presents a solution to the TTC 2017 State Elimination case using the language workbench MontiCore and its extension MontiTrans. This extension provides model transformation support using transformation models conforming to automatically derived domain-specific transformation languages. We present our results and compare them with those of other entries from 2017. Our solution performs comparably to those solutions and similarly does not scale up to the largest models. The domain-specific nature offers a unique perspective for transformation developers. This case highlights possibilities for future research into model transformation composition.

Keywords

Model-Driven, Modular Model Transformations, Domain-Specific Transformation Languages, State Elimination

1. Introduction

The 2017 State Elimination case [1] of the transformation tool contest (TTC) requires the composition of multiple transformations to solve the model transformation problem of transforming finite state automata into regular expressions. It calls for transformation tools to be assessed based on their understandability, correctness, performance, and suitability.

In this paper, we present a solution using the MontiCore *Automata* domain-specific language (DSL) and its corresponding domain-specific transformation language (DSTL), based on the language workbench MontiCore [2] and its extension MontiTrans [3]. Instead of working with an Ecore metamodel as in the reference implementation, the *Automata* DSL is used to parse textual automata models. The actual state elimination steps are described through automata transformations written using the concrete syntax of the *Automata* DSL, i.e., transformation models conforming to the *AutomataTR* DSTL. The DSTL is automatically derived from the *Automata* DSL. These model transformations are applied sequentially to the input model until only a single transition with the resulting regular expression remains.

The remainder of this paper is structured as follows: In Section 2, we give a brief introduction to MontiCore and MontiTrans, as well as a more detailed introduction to the automata language and the DSTL generated from it. In Section 3, we describe the state elimination case. We then present our implementation of state elimination based on MontiCore in detail in Section 4. We evaluate our solution in Section 5, assessing the results not only based on the given evaluation criteria but also in comparison with previous solutions. Finally, we conclude our paper in Section 6 with a summary of our findings and potential future work.

2. MontiCore

MontiCore is a language workbench for textual domain-specific languages (DSLs). By incorporating model composition, the efficient development of DSLs is facilitated. In addition to generating language infrastructure, such as a parser, classes for an abstract syntax tree (AST), AST traversal, symbol table management, etc., the MontiTrans extension derives domain-specific transformation languages (DSTLs)

Joint Proceedings of the STAF 2026 Workshops: AgileMDE, GCM, ICMM, LLM4SE, TTC. Rennes, France, June 29-July 3, 2026.

luepges@se-rwth.de (A. Lüpges); rapp@se-rwth.de (J. Rapp); rumpe@se-rwth.de (B. Rumpe)

0009-0009-4542-3895 (A. Lüpges); 0009-0007-9063-6473 (J. Rapp); 0000-0002-2147-1966 (B. Rumpe)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0)

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-26

CEUR-WS.org/Vol-4338/TTC-paper4.pdf

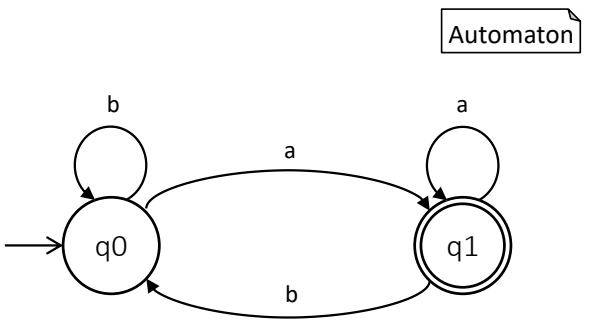
from DSLs. DSTLs allow for the modeling of patterns and homogeneous model transformations by leveraging the existing syntax of their DSLs.

The *Automata* language[1] is one of the smallest MontiCore DSLs and is frequently used in tutorials and demonstrations. Listing 1 shows an even further simplified version of the *Automata* grammar, being stripped of nested states and symbol infrastructure.

MG Automata

```
1  grammar Automata extends de.monticore.MCBasics {
2      Automaton =
3          "automaton" Name "{" (State | Transition)* "}" ;
4      State =
5          "state" Name (("<<" ["initial"] ">>" ) | ("<<" ["final"] ">>" ))* ;
6      Transition =
7          from:Name "-" input:Name ">" to:Name ";" ;
8      // Implicitly provided via the MCBasics grammar - included for readability
9      token Name =
10         ( 'a'..'z' | 'A'..'Z' | '_' | '$' )
11         ( 'a'..'z' | 'A'..'Z' | '_' | '0'..'9' | '$' )*;
12 }
```

Listing 1: (Simplified) grammar definition of the *Automata* DSL, consisisting of the start production for an automata, a state production, and a transition production.



(a) Graphical representation

Automaton

```
1 automaton SimpleAutomata {
2   state q0 <<initial>>;
3   state q1 <<final>>;
4   q0-b>q0;
5   q0-a>q1;
6   q1-a>q1;
7   q1-b>q0;
8 }
```

(b) Textual representation

Figure 1: A simple finite automaton given in a common graphical representation as well as its textual representation with the MontiCore *Automata* language.

It consists of an `Automaton` start production (lines 2-3), which refers to the `State` and the `Transition` productions. States can be marked as initial or final (lines 4-5). Transitions have an input and refer to their source and target states (lines 6-7).

The *AutomataTR* language is the domain-specific transformation language derived from the *Automata* language. Listing 2 contains some examples of this DSTL. Valid models of the *Automata* DSL are valid patterns and thus also valid models of the *AutomataTR* language (see line 1). To allow for generalization, all identifiers, such as references to the `Name` production, can be substituted using schema variables, as denoted via the dollar sign (see line 2). Whole AST elements can also be captured in a schema variable, such as a state in line 3. To improve the readability of all listings in this paper, we have highlighted all schema variables with a magenta color and all transformation operators with a grey background. By default, all parts of a pattern must be present, and any additional elements are ignored. The negation operator can be used to explicitly forbid specific sub-patterns (line 5). Similarly, optional sub-patterns can be defined via the optional operator (line 6). To match a sub-pattern multiple times, but at least once, the collection operator can be used (line 7). The `$_` anonymous schema variable can be used in multiple places, where the content of the variable is not further needed.

To perform transformations and not just pattern matching, the `[[ ... :- ... ]]` replacement operator is used. The left-hand side contains the pattern, while the right-hand side contains the

[1] https://github.com/monticore/automaton

```
TR Example
1  state q0;                          // concrete syntax
2  state $sName;                      // concrete syntax with schema variable
3  $S [[ state $sName; ]]             // element capture
4
5  not [[ state $_; ]]                // negated sub-pattern
6  opt [[ state $_; ]]                // optional sub-pattern
7  list [[ state $_; ]]               // list of sub-patterns
8
9  state [[ q0 :- q1 ]];              // replace identifier
10 [[state q0; :- q0 - i > q1; ]] // replace element
```

Listing 2: Automaton DSTL pattern and transformation examples.

replacement. It allows the matching and replacement of identifiers (line 9) and elements (line 10). By leaving the left-hand side empty, the replacement is inserted. Similarly, leaving the right-hand side empty results in the removal of an element. Each transformation model can additionally be extended with an application constraint in a `where` block, a schema variable assignment in an `assign` block, and finalizing Java code in a `do` block.

At runtime, each transformation is performed in four steps. First, the model is loaded by fully traversing all nodes to collect potential candidates. Next, a search plan is created to determine which element to match next, based on factors such as the number of elements of a given type, fixed elements, parent-child relations, additional constraints, etc. The search plan is then followed by selecting matching candidates and possibly returning either a match or performing backtracking. After each match, the replacement action can optionally be performed. Although the initial loading of the model provides primitive support for incremental transformations, the incremental matching of composed transformations is not yet supported.

3. State Elimination Case

State elimination is a well-known algorithm for converting a finite-state automaton (FSA) into a regular expression. An FSA $= (Q, s, \Sigma, F, \delta)$ consists of a set of states Q, a start state ($s \in Q$), a finite input alphabet Σ, a set of final states F, and a transition function $\delta : Q \times \Sigma \to Q$. Such an FSA is uniform if it has a unique initial state with no incoming transitions and a unique final state with no outgoing transitions. Acceptors are the simplest type of FSA and process a sequence of symbols from the input alphabet. Depending on whether the automaton is in a final state, the input is accepted or rejected.

During state elimination, the states of a uniform FSA are continuously eliminated while the connected transitions are merged step by step. Ultimately, only the initial and final states remain, connected by a single transition. The label of this transition then corresponds to the regular expression that describes the language accepted by the acceptor.

Regular expressions are primarily used to define text patterns. These patterns can then be used to search text or validate input. The length and complexity of a regular expression significantly impact the performance of the matching process. When eliminating states and merging transitions, transition labels are concatenated repeatedly, which, in the worst case, leads to an explosion of the resulting regular expression. Therefore, it is common to use heuristics to obtain the shortest possible regular expressions by optimizing the order of elimination [4].

The State Elimination Case includes one main task and two optional extensions. The main task deals with automata with only one initial and one final state. However, some of these automata must first be made completely uniform since their initial and final states may have incoming and outgoing transitions, respectively. Extension 1 considers automata that can have multiple initial and final states. Finally, Extension 2 considers probabilistic automata whose transitions are assigned a probability value.

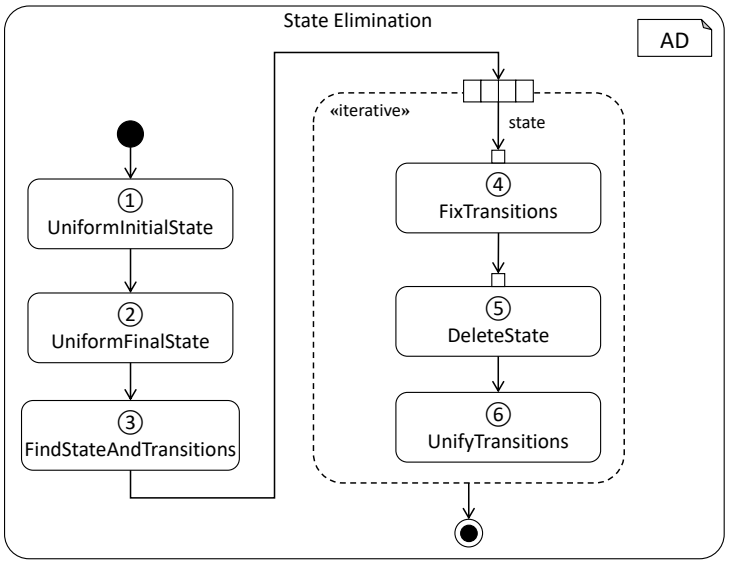


Figure 2: Activity diagram describing the composition of the required automaton transformations for state elimination

4. Solution

In this section, we describe our implementation[2] of the State Elimination case using MontiCore. In Section 3, we already provided a brief overview of the state elimination algorithm, the main task, and the extensions of the TTC case. In the following, we will take a detailed look at the individual steps of the algorithm and thereby explain our solution to the main task.

The automata provided by the case authors are available as XMI resources that conform to an Ecore metamodel. However, since MontiCore is not based on the Eclipse Modeling Framework (EMF), and we use the *Automata* MontiCore domain-specific language (DSL) instead of the Ecore metamodel, these automata must first be translated into models of the *Automata* DSL. The translation of all provided automata is performed in advance. To do this, the automata models are loaded one after another using the EMF API. For each model, we iterate over all EObjects of a resource, create either a new State or Transition AST element in the *Automata* language based on their EClass, and finally print the finished automaton to a new file using a built-in model-to-text transformation.

Our implementation of the state elimination algorithm is broken down into six transformations, one of which is complemented by Java code for sorting states. Figure 2 provides a basic overview of these transformations and their composition. In the following, we will introduce them step by step before concluding this section with their composition and actual execution.

As mentioned previously in Section 3, finite state automata (FSAs) must be uniform for successful state elimination. Therefore, an FSA may have only one initial state and one final state, whereby the initial state must have no incoming transitions and the final state must have no outgoing transitions. While the automata provided for the main task satisfy the former condition, they do not always satisfy the latter.

For this reason, we start with the ① *UniformInitialState* transformation, shown in Listing 3a. The pattern searches for the current initial state (line 2) and a transition entering that state (line 4). When the transformation is executed (on a found match), the initial modifier of the original initial state is removed (line 2), a new initial state is created (line 6), and an epsilon transition is created from the new initial state to the original initial state (line 7). The ② *UniformFinalState* transformation, shown in Listing 3b, follows the same process for the final state, except that outgoing transitions are considered here instead of incoming ones. Once these two transformations have been performed, all prerequisites in the main task are met, and state elimination can begin.

[2] https://github.com/jra-se/ttc-stateelimination

```
TR UniformInitialState
1 automaton $_ {
2   state $sName << [[ initial :- ]] >>;
3
4   $_ - $_ > $sName;
5
6   [[ :- state xInitial <<initial>>; ]]
7   [[ :- xInitial - ε > $sName; ]]
8 }
```

(a) Transformation to create a new initial state if the original initial state has incoming transitions.

```
TR UniformFinalState
1 automaton $_ {
2   state $sName << [[ final :- ]] >>;
3
4   $sName - $_> $_;
5
6   [[ :- state xFinal <<final>>; ]]
7   [[ :- $sName - ε > xFinal; ]]
8 }
```

(b) Transformation to create a new final state if the original final state has outgoing transitions.

Listing 3: Transformations to create a new initial and final state. Language elements added to the concrete syntax by the transformation language are highlighted in gray.

```
TR FindStateAndTransitions
1 automaton $_ {
2   $s [[
3     state $sName <<not [[initial]]>> <<not [[final]]>>;
4   ]]
5   list $INCOMING [[ $_ - $_ > $sName; ]]
6   list $OUTGOING [[ $sName - $_ > $_; ]]
7 }
```

Listing 4: Pattern to find an eliminable state that is neither initial nor final, as well as all incoming and outgoing transitions. Language elements added to the concrete syntax by the transformation language are highlighted in gray.

We start with the ③ *FindStateAndTransitions* transformation to find all states and their respective incoming and outgoing transitions. This transformation is used solely for pattern matching and does not modify the automaton. Listing 4 shows the pattern in which we match a state that is neither initial nor final using the `not` operator (line 3) and capture it in a variable (lines 2-4). We also capture all incoming and outgoing transitions using MontiTrans' `list` operator to group these transitions into a single match each (lines 5 and 6).

Initially, our solution did not follow a specific order for the state elimination, resulting in an exponential blow-up of the regular expressions for some inputs. There exists a multitude of heuristics to reduce the expression complexity [4]. For this reason, we first collect all eliminable states and sort them in ascending order by their degree (the sum of incoming and outgoing transitions). This ensures that states with the lowest degree are eliminated first. Although this ordering strategy does not yield optimal results, it was trivial to implement using the DSTL `list` operator to sort the matches based on the degree of the state.

A state is eliminated by first creating a new transition for each combination of that state's incoming and outgoing transitions. The ⑤ *FixTransitions* transformation handles this process and is shown in Listing 5. When called, it receives the current state as a fixed value so that it does not need to be matched again. Since the value of the new transition's label depends on the presence of self-loops, such self-loops are included as optional elements in the pattern (line 6). During the transformation step, a new transition is inserted from the start state of the incoming transition to the target state of the outgoing transition. However, the transition label must be created in the `do` block (lines 11 ff.), as the optional self-loops requires us to consider two cases. The `do` block executes after the transformation is complete and allows conditional logic to overwrite the label depending on whether self-loops are present. We apply this transformation repeatedly until all combinations of incoming and outgoing transitions of the state to be eliminated have been taken into account.

Now that all the new transitions have been created, the state to be eliminated can finally be deleted

TR FixTransitions

```
automaton $_ {
  state $cur;

  $A - $ACLabel > $cur;
  $cur - $CBLabel > $B;
  opt [[ list $LOOPS [[ $cur - $loopLabel > $cur; ]] ]]

  [[ :- $NEWT [[ $A - UNDEFINED > $B; ]] ]]
}

do {
 if ($LOOP.isPresent()) {
  m.$NEWT.setInput($ACLabel + "(" + $loopLabel_list.stream()
  .map(lbl -> "(" + lbl + ")")
  .collect(Collectors.joining("|")) + ")*" + $CBLabel);
 } else {
  m.$NEWT.setInput($ACLabel + $CBLabel);
 }
}
```

Listing 5: Transformation to create a new transition between two states, which is intended to replace a transitive path between three states. The label of the new transition depends on whether there are self-loops at the middle state. Language elements added to the concrete syntax by the transformation language are highlighted in gray.

TR DeleteState

```
automaton $_ {
  [[ state $sName; :- ]]

  [[ list [[ $_ - $_ > $sName; ]] :- ]]
  [[ list [[ $sName - $_ > $_; ]] :- ]]
}
```

Listing 6: Transformation to delete a given state, including all incoming and outgoing transitions. Language elements added to the concrete syntax by the transformation language are highlighted in gray.

TR UnifyTransitions

```
automaton $_ {
  [[ $A - $y > $B; :- ]]
  $A - [[ $x :- $xNew ]] > $B;
}

assign {
 $xNew = "(" + $x + "|" + $y + ")";
}
```

Listing 7: Transformation that combines two transitions with the same start state and end state into a single transition by deleting one and overwriting the label of the other with a combination of both labels. Language elements added to the concrete syntax by the transformation language are highlighted in gray.

using a ⑥ *DeleteState* transformation. This transformation also removes all the incoming and outgoing transitions. As Listing 6 shows, we use the `list` operator once again to delete all transitions with a single transformation execution.

Finally, we need to consolidate parallel transitions before the next state can be eliminated. These transitions can occur in the form of multiple self-loops at a single state. Additionally, parallel transitions can occur between different states, which means they have the same start state and the same end state. Since parallel self-loops share a start and end state, we do not need to distinguish between them,

requiring only the ⑦ *UnifyTransitions* transformation. The transformation shown in Listing 7 first matches two transitions that have the same start and end states (lines 2–3). In the transformation step, one of these transitions is removed (line 2), while the label of the other transition is overwritten with a combination of the labels of both transitions (line 3). This transformation is applied repeatedly until there are no more parallel transitions in the model.

The default pattern-matching strategy for both transitions involves iterating through all candidates and collecting those that meet the given constraints. For the second transition, the constraints are that the source and target state names are equal to those of the first transition. Since MontiTrans works on ASTs that do not contain edges for transitions, all transitions in the AST are potential candidates. In order to reduce this search space and thus improve the performance, we have included a hash-based lookup on the source and target state names. This is done using the TOP-mechanism [5], which enables handwritten extensions of generated code in MontiCore. Thus, given the first transition, we only iterate through those transitions with the same hash for the (source, target) pair as candidates for the second transition. While this improvement currently requires handwritten code, we aim to include it as a generated feature. This concludes the elimination of the selected state, allowing the algorithm to proceed with the next one.

The transformation definitions presented above are written as models of the *AutomataTR* DSTL, which allows the use of the *Automata* DSL's concrete syntax. MontiTrans uses these transformation models to generate Java classes that handle pattern matching and transformations. In our solution, we use a Gradle plugin to invoke this generation. Composing these transformations requires handwritten Java code that calls the transformations in the correct order and with the correct number of iterations. Additionally, matched elements must be passed between transformations if necessary. MontiCore provides a reporting framework to keep track of modified model elements. As it writes these changes to disk by default, we have disabled it for each transformation for performance reasons.

5. Evaluation

The case authors define four evaluation criteria against which solutions should be assessed. In the following, we will therefore examine the correctness, suitability, performance, and scalability of our solution.

The evaluation was performed on a Windows machine equipped with an Intel Core Ultra 7 165U processor featuring 12 cores clocked at up to 4.9 GHz and 32 GB of RAM. The implementation uses MontiCore version 7.9.0-SNAPSHOT and runs on Java 21 with Gradle 8.14.4[3].

Since this is a solution for a case from a previous year, solutions submitted at that time are also considered. These include solutions that use the tools NMF [6], Henshin [7], Eclipse Epsilon [8], and Fujaba [9]. Unfortunately, we could not run the reference implementation based on JFLAP that the case authors provided. The SDMLib implementation [10] is no longer available and was therefore also not considered. All other implementations were run unchanged on the same hardware to obtain comparable results. This means that the tools were run using their versions as specified in their submissions and that subsequent developments and improvements to the tools were not taken into account.

Correctness The goal of the state elimination algorithm is the reduction of a finite state automata (FSA) to a regular expression. A first, straightforward correctness check is therefore to verify whether only two states and a transition between them remain once state elimination terminates. Proving the semantic equivalence of the resulting regular expression with a model solution, however, is complex. Instead, the authors of the case provide a set of positive test cases for each input pattern that should be accepted by the computed regular expression, as well as a set of negative test cases that should be rejected by the regular expression. We have adopted this approach and are also checking whether the resulting regular expressions accept or reject the positive and negative test cases, respectively.

[3] A Docker image with the exact version as used for our evaluation is available as ghcr.io/jra-se/ttc-stateelimination@sha256:bf332a6feabd555b8ea387cf85b332f66253e940449279488e795aa749fc0140

Model			Regular expression		Test cases	
name	#states	#transitions	obtained	length	positive	negative
leader3_2	26	33	✓	33	24/24	29/29
leader3_3	69	95	✓	95	25/25	30/30
leader3_4	147	210	✓	210	29/29	30/30
leader3_5	273	397	✓	398	30/30	30/30
leader3_6	459	674	✓	675	27/27	30/30
leader3_8	1059	1570	✓	1571	29/29	30/30
leader4_2	61	76	✓	76	25/25	29/29
leader4_3	274	354	✓	354	27/27	30/30
leader4_4	812	1067	✓	1067	28/28	29/29
leader4_5	1933	2557	✓	2558	28/28	29/29
leader4_6	3962	5257	✓	5262	25/25	30/30
leader5_2	141	172	✓	172	14/14	28/28
leader5_3	1050	1292	✓	1292	26/26	30/30
leader5_4	4244	5267	✓	5267	26/26	29/29
leader5_5	12709	15833	✓	15833	24/24	30/30
leader6_2	335	398	✓	398	15/15	29/29
leader6_3	3759	4487	✓	4487	21/21	30/30
leader6_4	20884	24979	✓	24979	26/26	29/29
leader6_5	78784	94408	✓	94408	28/28	30/30
leader6_6	234210	280865	✓	280865	25/25	30/30

Table 1
A list of all models, along with the number of their states and transitions. The length of the resulting regular expression and the results of the positive and negative test cases are also reported.

As shown in Table 1, we derived a regular expression for all models to which we applied the state elimination algorithm. Earlier approaches resulted in regular expressions that were excessively long, even for smaller models (e.g., leader4_4) and reached the maximum string length in Java. However, due to the applied heuristics, the length of the regular expression now rarely deviates from the number of transitions, as shown in the table. Therefore, we can conclude that the regular expressions adequately represent the automata without being excessively large. The table also shows that the computed regular expressions correctly accepted all positive test cases and correctly rejected the negative ones. From this, we conclude that our state elimination implementation is correct.

Suitability Suitability is measured by the percentage of declarative specification or the level of abstraction used by the state elimination solution. We argue that the description of patterns and transformations in our solution is highly abstract. Using the concrete syntax of a DSL is independent of the suitability metric. The transformations in our solution require few imperative statements and only use `assign` blocks for string manipulation. Although the generated transformation classes allow for some abstraction, their composition, i.e., the control flow that invokes transformations, exchanging matched elements as fixed constraining elements, and controls loops, requires handwritten Java code. Overall, we would rate the suitability of our solution as *Medium*, with declarative transformation models but an imperative control flow. In the future, we aim to improve the composability of MontiTrans' model transformations, which should result in an improved suitability rating.

Performance MontiTrans uses a local search driven by a search-plan for pattern matching on the AST of the input model. This means, that for each element of a pattern, all possible candidates are considered. Table 2 lists the execution times of our solution, as well as the newly measured execution times for earlier solutions using NMF, Henshin, Epsilon, and Fujaba, and shows the advantages and

Model	Execution time (ms)				
	MontiTrans	NMF	Henshin	Epsilon	Fujaba
leader3_2	**6**	110	380	109	298
leader3_3	**26**	111	178	134	1108
leader3_4	**80**	112	329	147	2973
leader3_5	**183**	118	996	219	9771
leader3_6	**410**	125	3638	398	28221
leader3_8	**1910**	194	44560	1000	179291
leader4_2	**15**	137	34	112	1074
leader4_3	**166**	122	758	223	9560
leader4_4	**975**	137	18230	854	88185
leader4_5	**3873**	206	247064	3000	599832
leader4_6	**23638**	448	2234352	17000	2982848
leader5_2	**48**	121	122	137	2912
leader5_3	**1163**	151	38233	1000	151334
leader5_4	**22011**	453	2611192	18000	3245626
leader5_5	**234119**	2607	*timeout*	198000	*timeout*
leader6_2	**228**	117	1441	273	14217
leader6_3	**15995**	379	2271060	13000	2361836
leader6_4	**1320637**	6697	*timeout*	653000	*timeout*
leader6_5	(**12608931**)	156987	*timeout*	*timeout*	*timeout*
leader6_6	(**184740997**)	1698384	*timeout*	*timeout*	*timeout*

Table 2
Execution times of our solution (highlighted in bold), as well as for previous solutions that used other transformation tools. To obtain comparable results, the implementations of the previous solutions were run unmodified on the same hardware. A 60-minute timeout was set for each model in all other solutions. Execution times of our solution greater than 60 minutes are embraced in parentheses. The *Epsilon* implementation does not report execution times with millisecond precision once the execution time exceeds one second.

disadvantages of the search algorithm. Figure 3 plots the execution times against the number of model elements. For small FSAs with few states and transitions, there are consequently few candidates that need to be tested. Accordingly, we achieve very fast execution times for such small automata since there is no overhead for initializing a search engine.

MontiTrans does not operate incrementally, which leads to several performance problems. As the number of states and transitions increases, pattern-matching becomes more expensive, causing execution times to rise exponentially. Since the system does not operate incrementally, neither intermediate results nor partial matches are saved. Consequently, individual structures (e.g., transitions) must often be searched for entirely from scratch within the model. To work around this issue, we pass the state to be eliminated as fixed values between transformations, thereby avoiding the need to search for it again. Manual adjustments, such as replacing the candidate selection in ⑦ *UnifyTransitions* with a hash-based lookup, have also resulted in significant performance improvements. The Epsilon solution employs a similar reduction technique calling it *memoisation*[4], which results in highly similar timings compared to our solution (cf. Figure 3). The data structures on which the pattern-matching is based are not updated incrementally either. This means that the AST of the selected model must be reloaded entirely during initialization and after every modifying transformation. In the future we plan to track modifications, especially between the initial collection in ③ *FindStateAndTransitions* and the looped steps ④ *FixTransitions*, ⑤ *DeleteState*, and ⑥ *UnifyTransitions*, thus, the repeated collection of candidates can be avoided. These factors play a minor role in smaller models but become decisive in larger ones.

[4]https://eclipse.dev/epsilon/doc/epl/#:~:text=memoisation

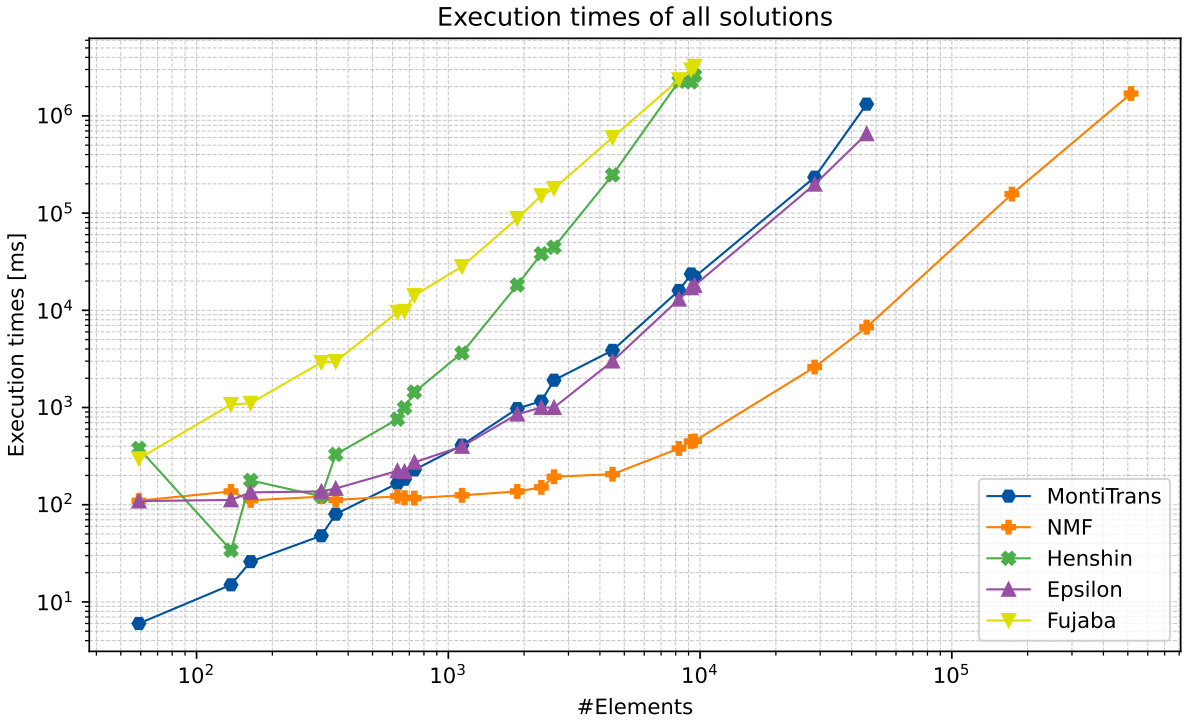


Figure 3: Execution times of our solution and of previous solutions with a 60-minute timeout, mapped to the number of elements, i.e., the sum of transitions and states.

Scalability The largest model for which our implementation was able to successfully complete state elimination was *leader6_6* with 234210 states and 280865 transitions, although at an impractical execution time of multiple days. Given a 60-minute timeout, the largest model we were able to eliminate was *leader6_4*, just as Epsilon's solution. Without the heuristic to determine the order of states to be eliminated, the largest model we were able to execute without running into Java's string length restriction was *leader3_8*.

6. Conclusion

In this paper we presented a solution to the State Elimination TTC 2017 case using the language workbench MontiCore. We were able to leverage the domain-specific transformation language for the *Automata* DSL to describe the state elimination steps, which enable an easier comprehension of transformations. By including handwritten code to reduce the search space of the pattern matching, we were able to avoid a major bottleneck. The composition of mainly generated transformations still required general-purpose Java code.

Comparing our performance results with four of the six solutions from 2017 leads to comparable timings. The results of the largest models show that efficient model transformations requires incremental composeable model transformations, and MontiTrans' rudimentary incremental support still requires work. We intend to use this case to further develop support for composable model transformations and generated optimised search operations.

Acknowledgments

Funded by the German Federal Ministry of Education and Research - 03G0922A

Declaration on Generative AI

During the preparation of this work, the authors used DeepL and Grammarly in order to: Grammar and spelling check, Paraphrase and reword. After using these services, the authors reviewed and edited the content as needed and take full responsibility for the publication's content.

References

[1] S. Getir, D. A. Vu, F. Peverali, D. Strüber, T. Kehrer, State elimination as model transformation problem, in: A. García-Domínguez, G. Hinkel, F. Krikava (Eds.), Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017, CEUR Workshop Proceedings, CEUR-WS.org, 2017, pp. 65–73. URL: https://ceur-ws.org/Vol-2026/paper4.pdf.

[2] K. Hölldobler, O. Kautz, B. Rumpe, MontiCore Language Workbench and Library Handbook: Edition 2021, Aachener Informatik-Berichte, Software Engineering, Band 48, Shaker Verlag, 2021. URL: http://www.monticore.de/handbook.pdf.

[3] K. Hölldobler, MontiTrans: Agile, modellgetriebene Entwicklung von und mit domänenspezifischen, kompositionalen Transformationssprachen, Aachener Informatik-Berichte, Software Engineering, Band 36, Shaker Verlag, 2018. doi:10.18154/RWTH-2019-00468.

[4] N. Moreira, D. Nabais, R. Reis, State elimination ordering strategies: Some experimental results, in: I. McQuillan, G. Pighizzini (Eds.), Proceedings Twelfth Annual Workshop on Descriptional Complexity of Formal Systems, DCFS 2010, Saskatoon, Canada, 8-10th August 2010, EPTCS, 2010, pp. 139–148. doi:10.4204/EPTCS.31.16.

[5] F. Drux, N. Jansen, B. Rumpe, A Catalog of Design Patterns for Compositional Language Engineering, Journal of Object Technology (JOT) 21 (2022) 4:1–13. URL: http://www.se-rwth.de/publications/A-Catalog-of-Design-Patterns-for-Compositional-Language-Engineering.pdf.

[6] G. Hinkel, An NMF solution to the state elimination case at the TTC 2017, in: A. García-Domínguez, G. Hinkel, F. Krikava (Eds.), Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017, CEUR Workshop Proceedings, CEUR-WS.org, 2017, pp. 75–79. URL: https://ceur-ws.org/Vol-2026/paper7.pdf.

[7] D. Strüber, Transformation of finite state automata to regular expressions using henshin, in: A. García-Domínguez, G. Hinkel, F. Krikava (Eds.), Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017, CEUR Workshop Proceedings, CEUR-WS.org, 2017, pp. 81–85. URL: https://ceur-ws.org/Vol-2026/paper10.pdf.

[8] M. Sharbaf, S. K. Rahimi, B. Zamani, Solving the state elimination case study using epsilon, in: A. García-Domínguez, G. Hinkel, F. Krikava (Eds.), Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017, CEUR Workshop Proceedings, CEUR-WS.org, 2017, pp. 91–95. URL: https://ceur-ws.org/Vol-2026/paper12.pdf.

[9] C. Eickhoff, S.-L. Raesch, P. Kolodziej, Yage Solution to the State Elimination as a Model Transformation Problem of the TTC'17, 2017. URL: https://transformationtoolcontest.github.io/2017/solutions/stateElimination/TTC_2017_paper_15.pdf.

[10] A. Weidt, A. Zündorf, The sdmlib solution to the TTC 2017 state elimination case, in: A. García-Domínguez, G. Hinkel, F. Krikava (Eds.), Proceedings of the 10th Transformation Tool Contest (TTC 2017), co-located with the 2017 Software Technologies: Applications and Foundations (STAF 2017), Marburg, Germany, July 21, 2017, CEUR Workshop Proceedings, CEUR-WS.org, 2017, pp. 87–89. URL: https://ceur-ws.org/Vol-2026/paper11.pdf.