Finite State Transducer Modification by Examples

Gábor Guta

July 20, 2010

Abstract

Model Transformation By Example (MTBE) is a new branch of model driven software development. Transducers (automata with output) can be used to abstract model transformation. This form of abstraction makes possible to consider applications of grammatical inference algorithms.

In this paper we investigate whether an effective inference procedure can be developed to derive a modified transducer from examples of desired input/output pairs instead of infering such a transducer from scratch. The paper starts with the description of our motivational example. Then we propose an algorithm to infer modifications of the transducers. Finally, we discuss metrics to evaluate the quality of such an inference.

Contents

1 Introduction

Model Driven Software Development (MDSD) is a promising technology to deliver high quality software efficiently [30]. Software is generated from models with the help of transformations. Current industrial practice of using MDSD focuses on the production of the software with the help of out-of-the-box transformations and provides no efficient support for the development of new transformations [1, 25]. The lack of efficient support of transformation development entails a quite steep learning curve of developing custom transformation. Our aim is to reduce effort needed to develop transformations, by providing automatic correction of them if the developers introduce minor changes into the result of the transformation.

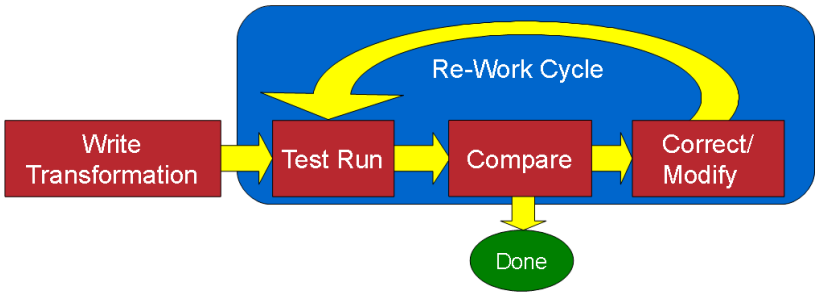


Figure 1: MDSD Process

The traditional model transformation process can be seen in Figure 1. The development of the transformation is usually preformed according to examples. Examples are pairs of source models and expected results. The transformation is crafted from expected result by inserting and surrounding instructions into them e.g. generalizing the example to get values from the source model or make the emitting of certain parts dependent from the source model. The transformation can be written in general programming languages e.g. Java, Python, etc. or in special purpose languages e.g. XSLT, Velocity, etc. After the creation of the initial transformation, it is executed on the source model of the examples. We compare the results of the execution to the expected results. If the evaluation is successful, we are done. If it does not meet our requirements, we have to correct or modify the transformation and repeat the process from the test run. During the creation of the templates we often do minor errors e.g. forgetting a white space or a semicolon. Such minor modifications can be located in the edit script with high precision and the trace information can be used to map the modification to the control flow graph of the M2T transformation. Correcting minor errors automatically may save the effort of several iterations of the re-work cycle.

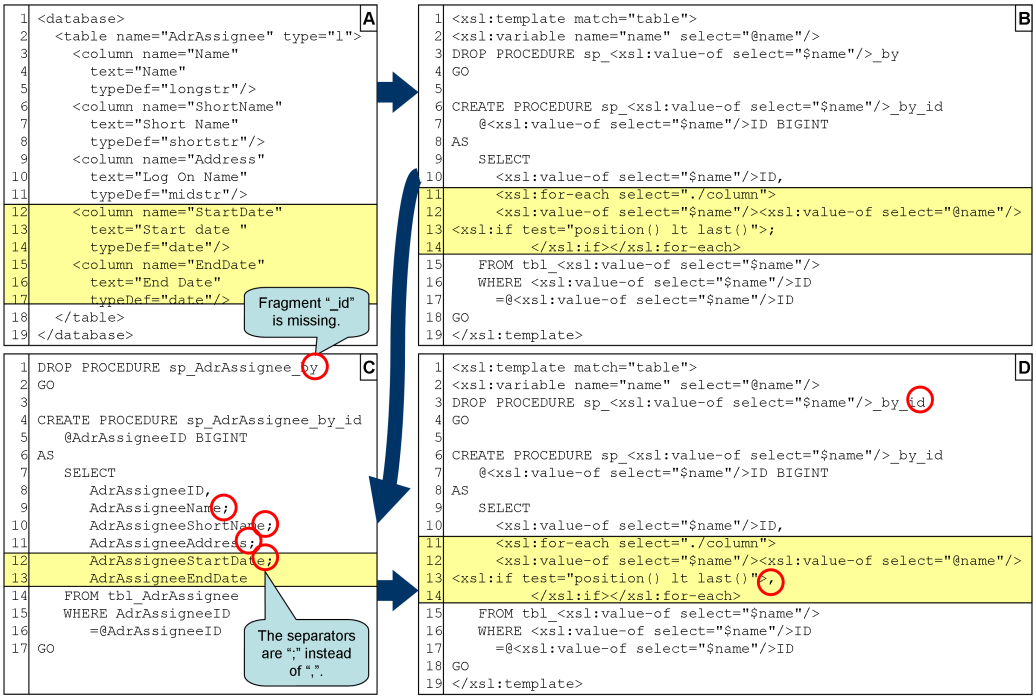


Figure 2: Example XSLT correction

To illustrate the basic idea we give as an example (Figure 2) an XML fragment of a database table description (A) which is transformed into a T-SQL code (B) by an XSLT transformation (C). This example is a simplified fragment of the code developed in an industrial project[28, 27]. The T-SQL code is a fragment from a database code of an enterprise application, which drops the old stored procedure and creates a new one. The stored procedure returns a single row of the database table which ID is equivalent to the input parameter of the stored procedure. We assume some minor errors in the transformation: the code emits a semicolon after the database column names instead of a comma, and the "_id" suffix is omitted from the procedure name (C). We correct these errors in the example T-SQL code and our goal is to derive the corrected XSLT template (D).

Our goal is to investigate the theoretical background of the inference of M2T transformation modifications. M2T transformations are modeled with simple finite-state transducers over strings. We present a general framework to describe inference of transducer modifications. We describe and evaluate an inference algorithm.

We aim to develop specialized algorithm to infer transformation modifications. The main difference of our approach from other grammatical inference

methods is that we require extra information in form of an existing approximate solution. According to the theoretical results of the grammatical inference field we know the inefficiency of algorithmic learning of even relatively simple grammars, e.g. inferring regular grammars over strings. We expect that the provided approximate solution helps to reasonably reduce the solution space of inference process. In the frame of our work we carry through the following tasks:

- develop algorithms to infer transformation modification by example;
- develop methods (define metrics) to objectively evaluate such inference algorithms;
- evaluate the developed algorithms with the developed methods and demonstrate their practical usability.

We start with the investigation of finite state transducers over strings. This simple transformation model makes easy to develop algorithms, metrics and evaluate algorithms with the proposed metrics.

In Section 2 we provide a formal description of the problem. In Section 3 we discuss the related work. In Section 4 we give a demonstrative example and according to this example we describe each step in detail. In Section 5 we define metrics and analyze the results. Finally, in Section 6 we conclude and summarize further work.

2 Formal Description of the Problem

In a model transformation system we usually generate code g from a specification s by a given transformation t. The generated code is determined by the specification and the transformation $g = t(s)$. We may define an expected result of the transformation g', which is produced from g by modification m, i.e. $g' = m(g)$. Our goal is to investigate whether it is possible to modify the approximate transformation t to the expected transformation t' automatically to generate the modified code g' from the original specification, i.e. $g' = t'(s)$. With fixed g and no constraints on the t' the task is minor, e.g. the algorithm may just learn to give the right answer to the specified example. In order to get a non-trivial solution, constraints (metrics of the generalization properties of the algorithm) need to be introduced on t'.

Therefore we reformulate this problem as depicted in Figure 3. S is a set of possible specifications. G is a set of generated codes where each element g of G has a corresponding element s of S such that $g = t(s)$. Take

a small subset of S and call it S_k. Call G_k the subset of G, containing the elements of G which correspond to the elements of S_k. Then we may introduce modification m over the elements of G_k, the result of which will be the members of the set G'_k. Thus the reformulated question is, whether it is possible to infer algorithmically an optimal transformation t', which transforms the elements of S_k to the corresponding G'_k elements in such a way that a certain "optimality" criterion is satisfied.

To express the optimality of the transformation we have to introduce metrics. These metrics cannot be computed by t alone, but are also influenced by other characteristics related to the transformation, e.g. similarity of t to t'. We also allow to our algorithm to deliver no or multiple results, because the examples may be contradicting or ambiguous.

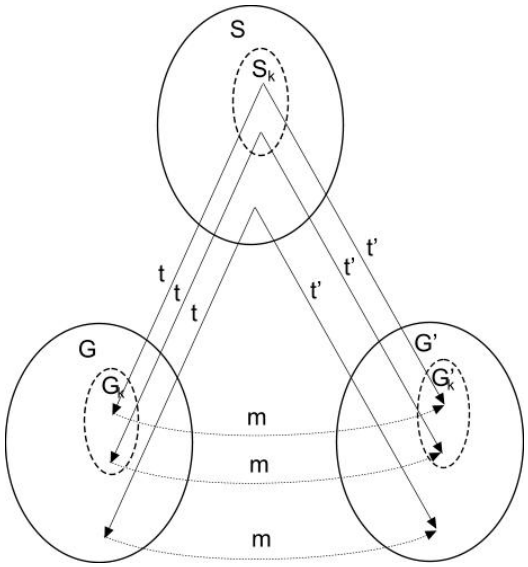


Figure 3: Transformation Modification Problem

3 State of the Art

The idea of model transformation by example (MTBE) has been recently recognized by the model transformation community [4, 44, 40, 21, 19]. In MTBE the transformation is inferred from examples by various methods, but they are still considered experimental. These methods are not capable to handle complex attribute transformations and the degree of automatization needs improvement [4]. The main difference in our approach is that we require an approximate solution from the developer which is automatically

or semi-automatically modified to achieve the required result. XML transformation by example is technically the same problem, but operates over the domain of tree structures instead of graphs [18]. Transformation by example is implemented by an inference process which produces transformations from the source schema, target schema and transformation examples. In the case that no transformation examples are specified the problem is called schema equivalence which was studied in [32]. The schema equivalence problem can be extended by specifying partial mappings between elements of the source schema and target schema as e.g. in IBM's Clio System [35]. The idea of programming by examples dates back as early as 1975 [39], but discussed by others since that time [38].

Research results can be found on the automatic correction of the data structures [12, 16]. The closest work related to our approach is [13], in which a dynamic tainting technique is described to locate errors in the input T2M transformation. This technique adds taints marks to the input which are traced to localize the input data triggers the incorrect output.

In case of unknown structures of the examples, the schemas can be inferred as well a set of examples is given and a schema (grammar) inferred from the example space[6, 5, 24]. A decent overview of inferring automata and transducers can be found in [11]. Two big classes of algorithms exist: generic search methods (e.g. GA, Tabu search, etc.) and specialized algorithms (e.g. RPNI, L-star, etc.). The RPNI algorithm builds an automaton which accepts only the given positive examples. Then the algorithm merges states if the resulting automaton does not accept any of the specified negative examples. By this way the algorithm delivers the most general solution respect to the given examples. [34] The L-Star algorithm starts building a table representation of a grammar by querying acceptance of strings appears in the table as unknown till the table representation becomes complete and consistent. If it is so, the algorithm asks the oracle whether the table contains the required grammar. If the answer is no, the oracle returns a counter example. The counterexample is inserted to the table and the algorithm starts to build again a consistent and complete table. This cycle repeats till the oracle answer yes to the proposed solution. [3] Application of such grammatical inference range from schema learning of XML documents [2, 22, 23, 41] to grammar learning of Domain Specific Languages [43].

The inverse problem of the structural inference is the example or test-case generation, i.e. examples are generated to represent the structure with respect to some structural coverage consideration. Work related to test case generation can be found in [7, 14, 37].

Although, we have not found any theoretical work on the inference of grammar modification by example, applications can be found which use

heuristics to solve such problems. Typical applications are learning dialects of a programming language [15] or error tolerant parsing [9, 10, 29, 42].

Automata can be extended to form an abstraction of general purpose programming language e.g. in [7] in which an Extended Finite State Machine described to model VHDL and C languages. Various classes of automata and transducers are defined to deal with trees and graphs e.g. tree grammars [31], summary of tree automatas and realted thories [8], Tree-to-Graph transduction [17]. A special area of the grammatical inference is when small changes are inferred on an existing structure according to examples. Typical applications are learning dialects of a programming language [15] or error tolerant parsing [9, 10, 29, 42].

4 Inferring Modification of Finite State Transducers

In this section we describe the algorithm which was developed to infer finite state string transducer modification. The algorithm infers a modified transducer from a set of example pairs and an original transducer. An example pair contains an input and the corresponding output which is the expected result of the execution of the modified transducer.

Technically, the only correctness requirement is that the algorithm must produce the expected output by executing on the example input and the behavior respect to the rest of the inputs is undefined. Informally we can formulate two further requirements: introduce the smallest possible modification on the transducer; and keep the behavior of the transducer with respect to the specified input as much as possible.

We can modify this transducer in infinitely many ways to produce the new output. One extreme solution is that we construct a transducer from the scratch to accept the specified input and to produce the expected output. The informal requirements provide us with some guidelines. We may formalize the informal requirements in two ways: we can define metrics and we expect the sum of these metrics minimal; or we may create the expected modified transducers and measure the properties of the input by which the algorithm produce the expected modification of the transducer (by forming a transducer for each example pair and create an union of them).

In Section 4.1 we define the concepts used to describe the algorithm. In Section 4.2 we describe the algorithm and explain the intuition behind it. The algorithm description starts with the brief explanation of the algorithm, which is followed by the detailed explanation through an example. In Sec-

tion 4.3 we discuss the properties of the algorithm informally and outline the possible way of its improvements.

4.1 Definitions

We provide here definitions of the concepts used in this work. We deviate from the traditional notation in order to make the definitions reusable in the algorithm as data type definitions. The main difference in our definition comparing to the classical notation is that we use records instead of tuples. *Input*, *Output*, and *State* are types of un-interpreted symbols from which the input alphabet, output alphabet, states formed, respectively.

- A *Transition Key* (or *Guard*) is a record of a source state and an input symbol.

  TransitionKey := (source: State, input: Input)

- A *Transition Data* element is a record of a target state and an output symbol.

  TransitionData := (target: State, output: Output)

- *Transition Rules* (or *Rules* or *Transitions*) are represented by a map of *Transition Keys* to *Transition Data*.

  Rules := TransitionKey $\rightarrow$ TransitionData

- A *Transducer* is a record of input symbols, output symbols, states, initial state, transition rules.

  Transducer := (input: set(Input), output: set(Output), state: set(State), init: State, rules: Rules)

- A *Trace Step* is a state transition record.

  TraceStep := (source: State, input: Input, target: State, output: Output)

- A *Trace* is a sequence of state transition records.

  Trace := TraceStep*

- A *Difference Sequence* is a sequence of records each of which consists of an output symbol and its relation to the source string and target string. The relation can be represented by " ", "-", or "+", which means that the output appears in both the source and the target, appears in the source, or appears in the target, respectively.

  Diff := (output: Output, mod: {" ", "-", "+"})*

- An *Usage Counter* is a map of the *Transitions Keys* to the number of their occurrence in the trace.

  UsageCounter := $TransitionKey \rightarrow \mathbb{N}$

- An *Inference Algorithm* is an algorithm which takes an *Transducer* and a set of input and output pairs as parameter and returns a transducer.

  InferenceAlgorithm := (Transducer, Input*, Output*)) → Transducer

The algorithms are represented in a Python like syntax. The main differences in our pseudo language are that we declare types to make the data passed between the algorithms easier to follow. Comments are starts with #. The notation has three kinds of data structures: record, sequence (list) and function (dictionary). Operators used over the data structures are defined as follows:

- Records are used to represent heterogeneous data structures.
  - They can be constructed by writing the name of the type and enumerating the values in the order of their field definitions enclosed in parentheses.
  - Values of the record can be accessed by "dot and the name of the field" notation.
- Lists are used to enumerate ordered values with equivalent type. They can be constructed by enumerating their elements. The signature of the access functions as follows:
  - list(_): name of T → T* (empty list)
  - head(_): T* → T (first element of the list)
  - tail(_): T* → T* (a list contains all but first element)
  - last(_): T* → T (last element of the list)

- allButLast(_): $T^* \to T^*$ (a list contains all, but first element)
- length(_): $T^* \to \mathbb{N}$ (the length)
- _ + _ : (T^*, T^*) $\to T^*$ (concatenation of lists)
- [_]: $T \to T^*$ (one element list)

- Functions are used to represent mappings.
  - func(_ , _): (name of T_1, name of T_2) $\to$ ($T_1 \to T_2$) (empty function)
  - _ [_]= _ : ($T_1 \to T_2$, T_1, T_2) $\to$ ($T_1 \to T_2$) (define function value for a key)
  - _ [_]: (($T_1 \to T_2$, T_1) $\to T_2$ (the value belonging to a key)
  - keys(_): ($T_1 \to T_2$) $\to T_1^*$ (sequence of keys on which the function is interpreted)

4.2 Trace Modification Based Inference Algorithm

In this section we describe our first inference algorithm. The algorithm starts by recording the trace of the transducer execution on a given input. The output is extracted from the trace and is compared. According to the result of the comparison, the algorithm modifies the trace (the algorithm is named Trace Modification Based Inference, because of this behavior). If an additional element is expected in the output, a fictional transition is inserted into the trace which leaves the internal state and the position of the input string unmodified and produces the expected output. If the expected change is the deletion of a trace element, the output of the trace step is changed to an empty sequence. To make the transducer capable to reproduce the modified trace, new transition rules are added or existing transition rules are modified. During the addition of new transition rules the algorithm keeps the transducer deterministic.

Example An example transducer is depicted on Figure 4. The nodes and the edges represent the states and the transitions, respectively. The arrow labeled "a/x" between the node "A" and "B" means that if "a" exists on the input and the transducer is in state "A", an "x" is produced on the output and the new state is "B". Empty (or λ) input means that the system changes its state without reading anything from the input. This transducer produces "xyywzxywz" from input "aabaaba". We can specify the requested modification by providing a single example pair: the input "aabaaba" and

```
1 inferTransducer(Transducer trans, Input* input,
2       Output* output'): Transducer
3   Trace tr=execute(trans, input)
4   Output* output=getOutput(tr)
5   Diff diff=calculateDiff(output, output')
6   Trace tr'=modifyTrace(tr,diff, trans.init)
7   Transducer trans'=modifyTransducer(trans, tr')
8   return trans'
```

Listing 1: The Main Algorithm

the expected output "xyyvzxyvz". The algorithm is expected to infer a new transducer from the specified transducer and from the example pair.

The Main Algorithm The main structure of the algorithm can be seen in the Listing 1. It takes the following inputs: a transducer *trans*, an example input *input* and an expected output *output'* and returns the modified transducer *trans'*. The algorithm starts with the execution of the input on the transducer, which produces the trace *tr* (line 3). The trace is sequence of trace steps corresponding to executed transition rules. A trace step records the source state, processed input symbol, target state and produced output symbol of the corresponding transition. The output is the sequence of the output elements of the trace sequence (line 4). Then difference sequence *diff* is calculated between the output and the expected output by a commonly used comparison algorithm, described in [36] (line 5). Then the modified trace *tr'* is calculated from the original trace *tr* and the difference sequence

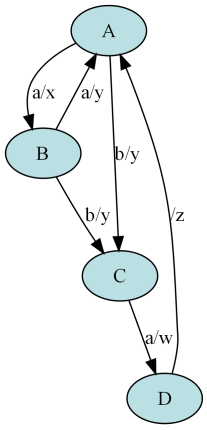


Figure 4: An Example Transducer

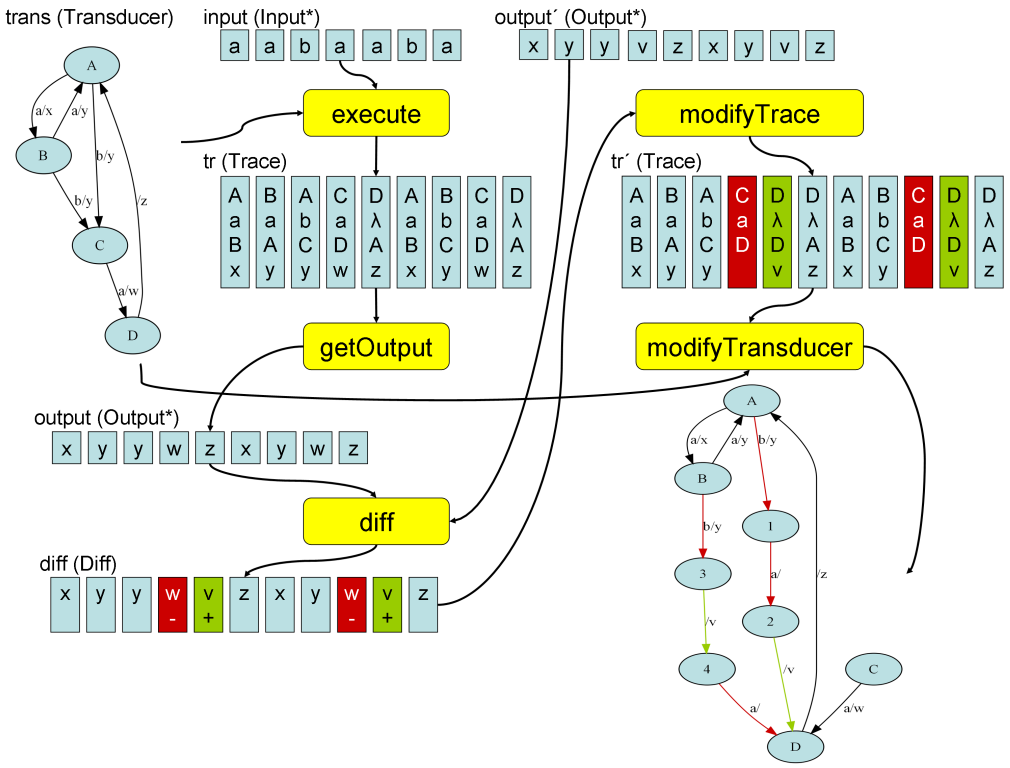
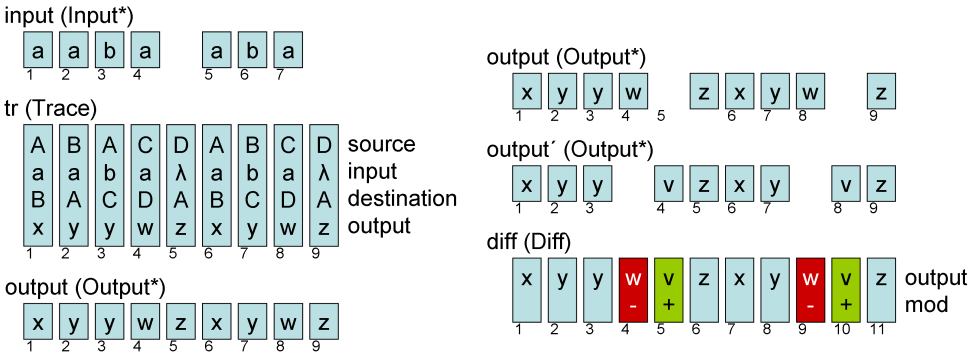


Figure 5: The Example of the Main Algorithm

diff (line 6). Finally, the new transducer *tr'* is built from the modified trace (line 7).

The function *execute(trans, input)* initialize its internal state variable according to the *init* element of the *trans* parameter. Then the function executes the rules in the *trans* parameter according to the *input* parameter. After each transition a new trace step is appended to the trace sequence. The function *getOutput(tr)* returns the sequence of the output elements of the trace sequence *tr*.

By considering the example described in the beginning of Section 4 we can follow the variables passed between the intermediate steps of the algorithm. We present the values here to provide an impression to the reader and the details of how the values are computed will be explained in parallel with the description of the functions. The graphical representation of the transducer *trans* can be seen in Figure 5. The value of the *input* parameter is "aabaaba" and the *output* parameter is "xyyvzxyvz". The *input*, the *tr*, and the *output* can be seen in Figure 6a. The result of the *diff* and the *modifiyTrace* functions can be seen on the Figure 6b and Figure 7, respectively. The actual result



(a) The Trace

(b) The Difference Sequence

Figure 6: Processing Steps of the Example

of the algorithm is the rightmost graph of the figure.

The modifyTrace Function We can see the *modifyTrace* function in Listing 2. If the input difference sequence (*diff*) is not empty (lines 2-12), the function processes the first element of the difference sequence and recursively calls itself to process the rest of the sequence. The result of processing of the rest is a modified trace which is concatenated to result of processing an element of the sequence and returned. If the sequence is empty (lines 13-15), the function returns the trace without modification (it is expected to return an empty sequence, because the trace must be empty in such case). The processing of the element of the difference sequence can happen in three ways depending on the modification represented by the element:

1. If there is no difference (lines 3-5), the element is copied without modification.

2. If the element is added (lines 6-8), a new trace step is inserted into the trace. The value of the source and the target states of the new trace step is the value the target state of the previous trace step. The new trace step reads no input and writes the same value which is defined in the processed element of the difference sequence.

3. If the element is deleted (lines 9-11), the output part of the trace step is modified to empty.

In Figure 7 the modifications of the trace can be seen. The elements marked with blue (light gray) ones are the unchanged, the green (gray) ones are the newly added ones and the red (dark gray) ones are the modified ones.

```
1  modifyTrace(Trace tr, Diff diff, State initS): Trace
2    if length(diff) > 0:
3      if head(diff).mod== " ":
4        Trace tr'=(head(tr))
5         +modifyTrace(tail(tr), tail(diff), head(tr).target)
6      elif head(diff).mod== "+":
7        Trace tr'=(initS,Empty, initS, head(diff).output)
8          +modifyTrace(tr,tail(diff),initS)
9      elif head(diff).mod== "-":
10       Trace tr'=(head(tr).source, head(tr).input,
11           head(tr).target," ")
12         +modifyTrace(tail(tr),tail(diff), head(tr).target)
13   else:
14     Trace tr'=tr
15   return tr'
```

Listing 2: The modifyTrace Function

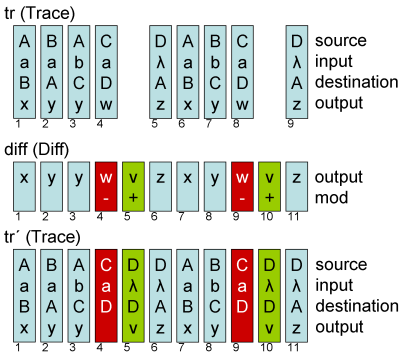


Figure 7: Example of the modifyTrace Function,

The function which infers the transducer modification from the modified trace uses three auxiliary functions:

- The function *isSimilarKey(Rules, TransitionKey): boolean* checks, whether the addition of the specified guard introduce non-determinism i.e. a rule exists with similar guard as the parameter. A guard of a transition rule is similar to the parameter if they expect the same input and in the source states are equal or any of them reads nothing and in the source states are equal.

- The function *getDistinctState(Rules): State* returns a new state symbol, which does not occur among the source or target state of the existing transition.

- The function *attachNewState(State, Transition, Trace, UsageCounter): (State, Transition, Trace, UsageCounter)* modifies the transition rule occurring in the previous trace element to provide the transition to the newly introduce state. This function is going to be explained later after the description of main algorithm.

The modifyTransducer Function The function *modifyTransducer* is shown in Listing 3. This function is actually a wrapper function which contains data structure initializations and a call of the *modifyTransducerImpl* function which contains the actual algorithm. This function takes a step of the trace and checks whether a transition rule exists for this execution step. If yes, no modification is necessary; if no, it inserts the step as a transition rule or modifies an existing one.

The wrapper function initializes the usage counter. This mapping assigns to each rule the number of its applications (lines 3-5). Then the function initializes the *trace'* which will contain the execution trace of the modified transducer with respect to the input symbols (lines 6-7). Finally, it executes the *modifyTransducerImpl* on each step of the trace.

The *modifyTransducerImpl* processes the trace step parameter. After the processing it returns a corrected trace to which the new trace step is appended. The transducer and the usage counter parameters are modified according to the processing of the trace step. The algorithm distinguishes between three cases:

- A transition rule exists for the trace step (lines 20-21). The usage counter corresponds to the rule is increased. The function returns the modified usage counter and inserts the trace to the end of the existing trace.

```
modifyTransducer(Transducer trans, Trace trace): Transducer
  if length(trace)>0:
    UsageCounter u=func(TransitionKey, nat)
    for guard in keys(trans.rules):
      u[guard]=0
    Transducer trans'=trans
    UsageCounter u'=u
    Trace trace'=list(TraceStep)
    for trStep in trace:
      (trans', u', trace')
          =modifyTransducerImpl(trStep, trace', trans', u')
    return trans'
  return trans

modifyTransducerImpl(TraceStep trStep, Trace trace,
    Transducer trans, UsageCounter u): (State, Transition,
    Trace, UsageCounter)
  TransitionKey guard
      =TransitionKey(trStep.source, trStep.input)
  if isSimilarKey(trans.rules, guard):
    if guard in trans.rules and trans.rules[guard]
        == TransitionData(trStep.target, trStep.output):
      u[guard]=u[guard]+1
      return (trans, u,  trace+[trStep])
    else:
      State newS=getDistinctState(trans.rules)
      TransitionKey newGuard
          TransitionKey(newS, guard.input)
      trans.rules[newGuard]
          =TransitionData(trStep.target, trStep.output)
      u[newGuard]=1
      (newS',trans',tracePrev',u')=attachNewState(newS,
          trans, trace, u)
      TraceStep trStep'=TraceStep(newS, trStep.input,
          trStep.target, trStep.output)
      return (trans', u', trace+[trStep'])
  else:
    trans.rules[guard]
        =TransitionData(trStep.target ,trStep.output)
    u[guard]=1
    return (trans,u, trace'+[trStep])
```

Listing 3: The modifyTransducer Function

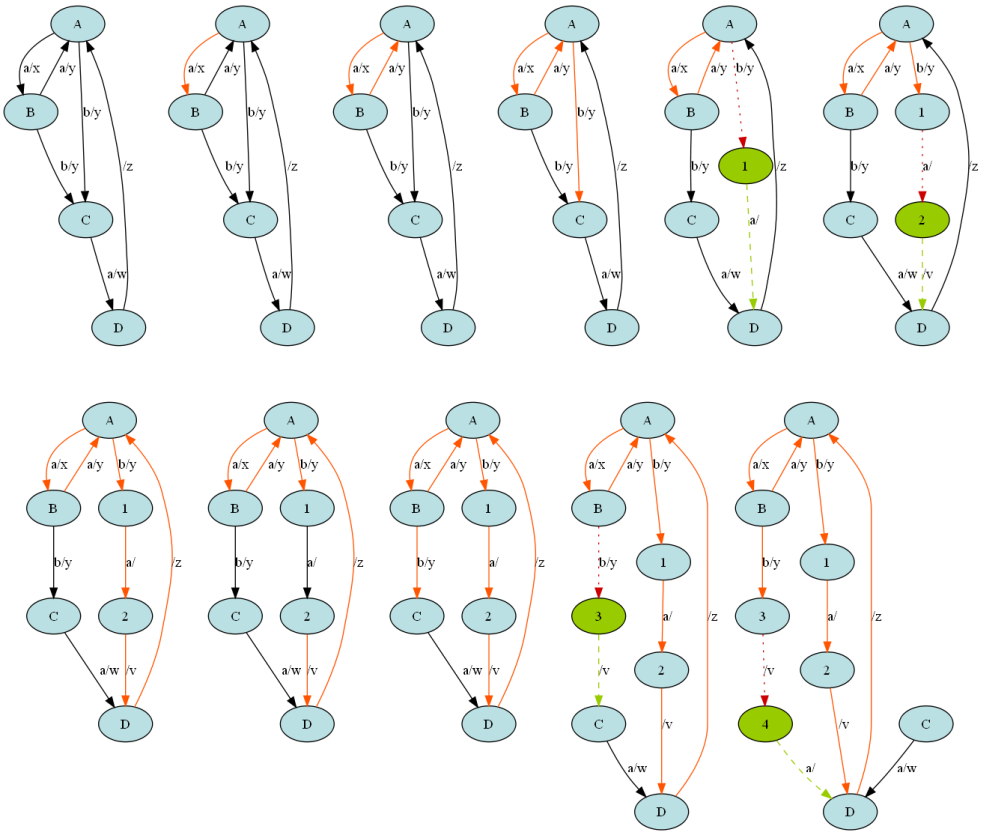


Figure 8: States of the Modifications of the Transducers

- A similar rule exists (lines 23-31). A new state is created and a transition rule which source state is this new state is added. Then the function calls the *attachNewState* function which modifies the existing trace and the transducer in a way that the target state of the last trace step be equal with the newly introduced state.

- No rule exists (lines 33-35). A new rule and the corresponding trace step is created.

We can follow the modifications of the transducer in Figure 8 according to the *tr'* (Trace) which can be seen in Figure 7. Each graph is the state of the transducer after the call of *modifyTransducerInfer* function. The solid black arrows mean unprocessed transitions; solid gray (orange) ones are the processed; the dotted gray (red) ones are the modified and the dashed gray (green) ones are the newly inserted. For example on the fourth graph from the left the state after processing of the third trace step can be seen. The

rule (A, b) $\rightarrow$ (C, y) is processed and the usage counter corresponding to this transitions (A, b) is set to 1. The forth step of the trace is a newly inserted element (C, a, D, λ). The corresponding transition rule is the (C, a) $\rightarrow$ (D, λ). Only a similar rule exists, namely (C, a) $\rightarrow$ (D, w). In this case the algorithm introduces a new rule (1, a) $\rightarrow$ (D, λ) and calls the *attachNewState* function to modify the previous transition to (A, b) $\rightarrow$ (1, y).

The attachNewState Function The *attachNewState* function can be seen in Listing 4. The function modifies the transition rule referenced by the previous trace element to provide the transition to the newly introduce state. If there are preceding transitions and the transition referenced in the previous trace step was applied once (lines 5-13), the algorithm modifies the detestation state to the new state. If the transition referenced by the previous trace step is applied multiple times, it copies the transition with a modified source ad target state and decreases the corresponding usage counter (lines 14-18). The target state is modified to the new state provided by the parameter *newS* and the source state is the incremented *newS* (line 17). Then the algorithm calls itself recursively to change the target state of the preceding transition to the increased *newS* (lines 20-24). This process repeats till the algorithm finds a transitions applied once or no more preceding trace step exists. If no more preceding trace step exists, the algorithm modifies the initial state (line 26). In the worst case the transducer contains as many states as elements as the length of the example. The example contains only situations which exercise the first branch. The modifications caused by this function are marked dotted gray (red) in Figure 8.

4.3 Remarks and Possible Improvements

In this section we analyze our algorithm and the inferred solution for the described example. In the analysis of the algorithm we describe aspects of the behavior of the algorithm. In the analysis of the described example we comment on the quality of the inferred solution and discuss possible ways to improve the algorithm.

There are four main aspects:

- Each example pair is interpreted as a set of example pairs: as the example pair itself and as the set of prefixes of the input part of the example and the corresponding outputs. The connection between the original and the expected output can be described with a modification function *mod: Output** $\rightarrow$ *Output**. More precisely to each prefix i

```
attachNewState(State newS, Transducer trans, Trace* tr,
    UsageCounter u): (State, Transition, Trace,
    UsageCounter)
  Transducer trans'=trans
  if length(tr) > 0:
    Trace trPrev=allButLast(tr)
    TransitionKey guard
        =TransitionKey(last(tr).source, last(tr).input)
    if u[guard]==1:
      trans'.rules[guard]
          =TransitionData(newState, last(tr).output)
      TraceStep newTrStep=TraceStep(last(tr).source,
          last(tr).input, newS, last(tr).output)
      Trace tr'=trPrev+[newTrStep]
      return (newS, trans', tr', u)
    else: #u[guard]>1
      UsageCounter u'=u
      u'[guard]=u'[guard]-1
      TransitionKey newTransGuard=(newS+1, last(tr).input)
      u'[newTransGuard]=1
      trans'.rules[newTransGuard]=(newS, last(tr).output)
      TraceStep newTrStep=TraceStep(newS+1, last(tr).input,
          newS, last(tr).output)
      (State, Transducer, Trace, UsageCounter) (newS',
          trans'', trPrev', u'')=attachNewState(newS+1,
          trans', trPrev,u')
      return (newS', trans'', trPrev'+[newTrStep], u'')
  else:
    trans'.initState=newS
  return (newS, trans', tr, u)
```

Listing 4: The attachNewState Function

of input I, belongs a prefix o' of the modified output O' which is the modified output of the transducer t produced from i.

$$\forall I, O, i': O=\text{mod}(\text{getOutput}(\text{execute}(t, I))) \wedge \text{prefix}(i', I) \Rightarrow \exists o': o'=\text{mod}(\text{getOutput}(\text{execute}(t, i))) \wedge \text{prefix}(o', O)$$

By an example pair "aabaaba" and "xyyvzxyvz" the set of pairs are "a", "x"; "aa", "xy"; "aab", "xyy"; etc.

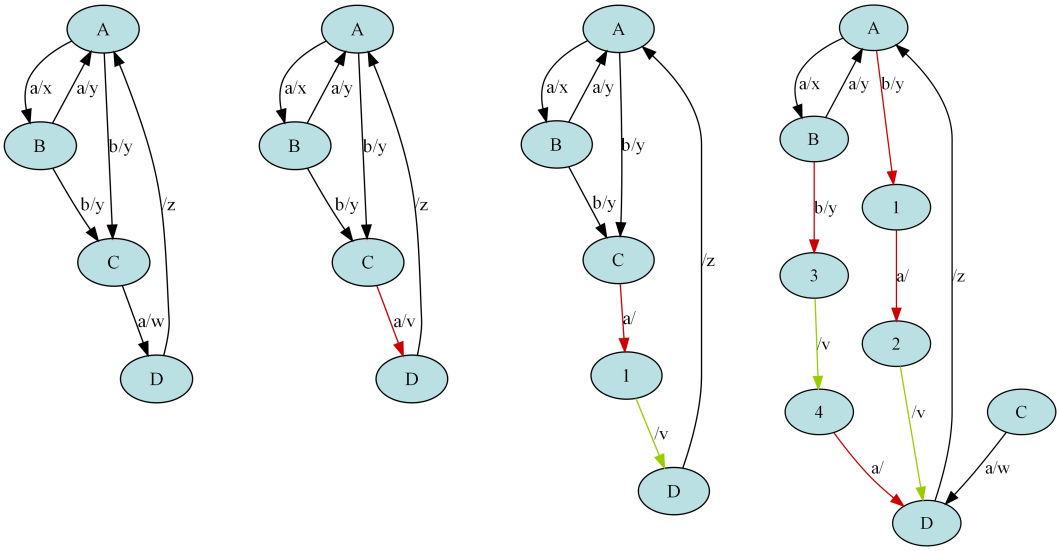
- The algorithm modifies the transducer immediately to a locally optimal form after each new trace step is seen. This means that the modified transducer will not change its behavior with respect to the already seen example. This also means that examples presented earlier affect the modification more significantly.

- The algorithm assumes that the state changes are bound to the output operation. Let us take a part of a trace as an example: (A, a, B, x), (B, a, C, y). The output "x, y" is modified to "x, z, y". This modification is interpreted as (A, a, B, x), (B, λ, B_1, z), (B_1, a, C, y) by the current algorithm. Alternatively it can be interpreted as (A, a, B, x), (B, a, B_1, z), (B_1, λ, C, y).

- The algorithm infers different solutions to the different results of the comparison algorithm even if the difference sequences are semantically equivalent. For example the difference sequence of modifying w to v can be either (w, -), (v, +) or (v, +), (w, -).

In Figure 9 four transducers can be seen. The original transducer, the solution which we intuitively expect (*Expected A*), the solution which we could still accept from an automated inference process (*Expected B*), and the inferred solution which the algorithm delivered can be seen in Figure 9a, Figure 9b, Figure 9c, and Figure 9d, respectively. The inferred solution produce the same result as the *Expected A* and *Expected B* for every input. The edges (3, λ) $\rightarrow$ (4, v), (4, a) $\rightarrow$ (D,) and (1, a) $\rightarrow$ (2,), (2, λ) $\rightarrow$ (D, w) are the result of the missing capability of the comparison algorithm to interpret the concept of the modification and not only deletion and addition. The unreachable transition (C, a) $\rightarrow$ (D, w) and the duplication of the modified (C, a) $\rightarrow$ (D, v) transition can be explained by the locally optimal behavior of the modification algorithm. This problem can be worked around by a clean up procedure which can normalize the solution by deleting unreachable transitions and merging paths with equivalent behavior.

A possible way to improve the algorithm is to delay the immediate modifications on the transducer after each step. The algorithm can track the



(a) The Original (b) The Expected A (c) The Expected B (d) The Inferred

Figure 9: Different Modifications of the Transducers

modification and consolidate the changes before the modification of the transducer. This may deliver a result that is closer to our expected solution. This remedy will be discussed in a subsequent paper. Before visiting the algorithm, we will in next section analyze features according to which the result of algorithm can be evaluated.

5 Evaluation of the Measurement Methods

We can analyze the quality of the inference algorithms by evaluating the results manually as we did in the previous section. In this section we focus on methods which can help to evaluate the quality of the solution automatically. First, we examine the basic formal properties of the solution, e.g. the number of new states. Then we introduce a new view-point to evaluate the quality of the solution, which is closer to our manual evaluation. This view-point is the effectiness of the inference with respect to the intention of the change. The metrics which are built according to this view point are explaining the quality of the solution much better and more intuitively.

Metrics	Expected A	Expected B	Inferred
Number of new states	0	1	4
Number of deleted states	0	0	0
Ratio of number of new states and number of original states	0	1/4	1
Ratio of number of deleted states and number of original states	0	0	0
Number of new transitions	0	1	2
Number of deleted transitions	0	0	0
Number of modified transitions	1	1	4
Ratio of number of new transitions and number of original transitions	0	0	1/3
Ratio of number of deleted transitions and number of original transitions	0	0	0
Ratio of number of modified transitions and number of original transitions	1/6	1/6	2/3

Figure 10: Metrics of the Example

5.1 Structural Metrics of the Transducer Modification

A modification of a transducer can be expressed by metrics as follows:

- the number of new states,
- the number of deleted states,
- the ratio of the number of new states and the number of original states,
- the ratio of the number of deleted states and original states,
- the number of new transitions,
- the number of deleted transitions,
- the number of modified transitions,
- the ratio of the number of new transitions and the number of original transitions,
- the ratio of the number of deleted transitions and the number of original transitions,
- the ratio of the number of modified transitions and the number of original transitions.

Input	Output	Output'	Equals
'b'	'y'	'y'	True
'a'	'x'	'x'	True
'ba'	'ywz'	'yvz'	False
'aa'	'xy'	'xy'	True
'ab'	'xy'	'xy'	True
'bab'	'ywzy'	'yvzy'	False
'baa'	'ywzx'	'yvzx'	False
'aab'	'xyy'	'xyy'	True
'aaa'	'xyx'	'xyx'	True
'aba'	'xywz'	'xyvz'	False
'baba'	'ywzywz'	'yvzyvz'	False
'baaa'	'ywzxy'	'yvzxy'	False
'baab'	'ywzxy'	'yvzxy'	False
'aaba'	'xyywz'	'xyyvz'	False
'aaaa'	'xyxy'	'xyxy'	True
'aaab'	'xyxy'	'xyxy'	True
'abab'	'xywzy'	'xyvzy'	False
'abaa'	'xywzx'	'xyvzx'	False

Figure 11: Example Input and Output Pairs with Maximum Length Four

N	All/Different	Avg. length till the first difference	Avg. num. of changed characters
1	2/0	0	0
2	3/1	2	1
3	5/3	7/3	1
4	8/6	16/6	7/6

Figure 12: Behavioral Metrics with Maximum Length Four

In Figure 10 we can see the metrics of the different modifications shown in Figure 9. The metrics for the second, the third and the fourth graph are labeled by "Expected A", Expected B" and "Inferred", respectively. One can clearly see that the more intuitive modifications have lower numbers in the table.

5.2 Behavioral Metrics of the Transducer Modification

The deviation in the behavior of a specific transducer caused by the modification can be characterized by comparing the output of the original and of the modified transducer for the same set of input strings. The set of all

Coverage Type	Coverage Amount
Transition coverage	100%
State coverage	100%
Path coverage	100%

Figure 13: Coverage metrics of the example

possible input strings with maximum length n can be generated easily. The simplest metric which can be defined is the ratio of size of this set and the number of differences in the output with respect to this set of inputs. We may measure also:

- the average length of the output until the first difference;
- the average number of changed characters in the modified outputs.

All the three modifications behave in the same way, that is why we have listed the results only once. In Figure 11 we can see rows of an input, an original and a modified output. Only rows corresponding to a valid input are listed (an input is valid if it is accepted by our example transducer). One can see that the number of the rows grows rapidly by increasing the limit of the input length. In Figure 12 we can see the behavioral metrics of these inputs up to length four with respect to the example we have studied.

5.3 Coverage Metrics of the Transducer Modification

In case of white box testing of software, the quality of the test cases is measured by test coverage [33]. Coverage metrics for transducers can be defined similarly to traditional code coverage metrics. If the transducers and the example are known, the coverage metrics can be calculated. We define the following coverage metrics for transducers:

- *Transition coverage* shows the percentage of the applied rules out of all rules (similar to the concept of statement coverage).
- *State coverage* shows the percentage of the reached states out of all states.
- *Path coverage* shows the percentage of the exercised paths out of all possible paths.

The coverage results of the example can be seen in Figure 13. The input achieves 100% coverage of the original transducer in our example. In

traditional software testing the aim is to reach approximately 80% program coverage. This provides the best cost/bug detection ratio. However the example needs complete coverage just around the modifications, i.e. a high coverage of the unmodified parts is not necessary.

5.4 Goal, Questions, Metrics Paradigm

The metrics defined in Section 5.1 can be correlated with the quality of the inferred solution, but they generally do not provide sufficient information to judge the quality of a solution. The reason for this phenomenon is that the inferred solution depends on the properties of the expected modifications and on the quality of the examples. In this section we use the term of *examples* to denote the sequence of input and output pairs. If a single modification of a non-cyclic path is expected, it can be presented by a simple example and low values of the transducer modification metrics can be expected. Intuitively, we have to define metrics to evaluate the quality of the example with respect to the transducer needs to be modified. To investigate this idea, we "borrow" the GQM (goal, quality, metrics) paradigm of software engineering [20].

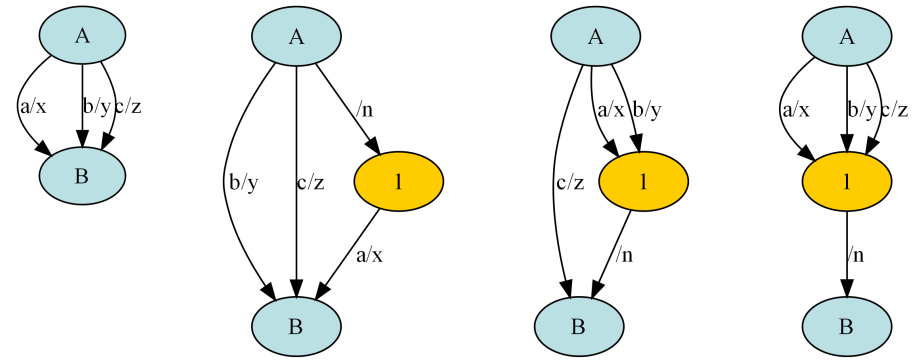
The GQM paradigm says that the goal of the measurement has to be formulated clearly, before any metric is defined. Then the defined goal has to be broken down into questions; the definition of the metrics is just the final step. In our case the goal is *to evaluate how effectively the examples describe the modification intention of the specifier.*

This goal can be formulated by the following questions:

- How many possible interpretation of the examples exist? (How clear is the intention of the specifier?) This question is related to the number of possible interpretations of the examples. An interpretation is a possible way to modify the transducer to make it capable to reproduce the examples. This question will be investigated in more detail in Section 5.5.

- Are the examples minimal? The question whether the examples are minimal can be decided easily: an example is minimal if there is no shorter input/output example pair which can be interpreted exactly in the same way.

- Are the examples consistent? Each example defines a set of the possible interpretations. We call two examples consistent if the intersection of the sets of the possible interpretations is not empty. The question can be determined by producing these sets.



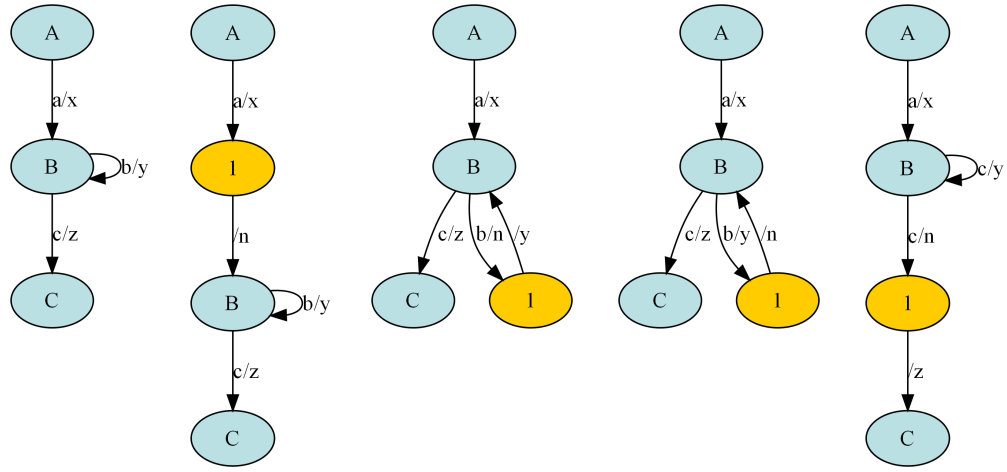
(a) The Original

(b) A single path is modified

(c) A set of paths are modified

(d) All paths are modified

Figure 14: Single Point Modifications of Branches



(a) The Original

(b) Modified Before the Cycle

(c) Modified at the Beginning of the Cycle

(d) Modified at the End of the Cycle

(e) Modified After the Cycle

Figure 15: Single Point Modifications of a Cycle

5.5 Metrics of the Intention

Rationale If we are talking about algorithms which are learning the modification from examples, it is an important to examine whether it is theoretically possible to do so. In this section we are going to investigate whether the modification intention of the specifier can be identified from examples (*examples* are sequences of input and output pairs). This investigated property has two aspects: a combinatorial one and a psychological one. The combinatorial aspect means that we determine how many interpretations (possible modifications) exist by a given transducer and set of examples. The psychological aspect means that how examples would be selected by a person to express his or her modification intention. In this chapter we focus on the combinatorial aspect and describe the psychological aspect in short to demonstrate the difference between the two aspects (we explicitly state that we mean intuition in sense of the author's intuition and not in sense of results of psychological experiments).

First we examine the examples of inference settings: we take example transducers and their possible modifications. We list all possible example traces which may be able to demonstrate the modification. Then we investigate which other modification may be represented by the same example trace. Finally, we try to generalize the metrics calculated for each examples to make it possible to determine metrics of the intention for any arbitrary pair of a transducer and a sequence of examples.

Examples We start our investigation by constructing two sets of modified transducers. The example transducers and their possible modifications are selected in a way to represent a class of modifications. The first set (Figure 14b, 14c, 14d) contains single point modifications of a transducer containing two states and three edges between these two states (Figure 14a). We represent the branching constructs in transducers by this set of examples. We consider three edges enough to represent the general concepts of modifying a single path (one edge), a set of paths (two edges), or all paths (three paths). The second set (Figure 15b, 15c, 15d, 15e) contains single point modifications around a loop in a transducer (Figure 15a). We can study the behavior of cycles in transducers with this set of examples.

We select each modified transducer and generate all sensible examples to represent the modifications. We do not consider an example sensible if it does not include at least one modified edge or if it differs only from the existing examples in the way in which the unmodified paths are selected. Then we check, which modified transducer from the set can be meant by the example. We summarize the result of this process in Figure 16 and Figure 17.

Modification	Example	Interpretation	Comb.	Int.
A single path	A modified path (a/nx)	A single path	1/3	Yes
		A set of paths	1/3	No
		All paths	1/3	Maybe
	A modified path + an unmodified one (a/nx;c/z)	A single path	1/2	
		A set of path	1/2	No
	All paths (a/nx;b/y;c/z)	A single path	1	No
A set of paths	A modified path (a/nx)	A single path	1/3	No
		A set of paths	1/3	Yes
		All paths	1/3	No
	A modified path + an unmodified one (a/nx;c/z)	A single path	1/2	No
		A set of paths	1/2	No
	A set of modified path (a/nx;b/ny)	A set of paths	1/2	Yes
		All paths	1/2	No
	All paths (a/nx;b/y;c/z)	A set of paths	1	Yes
All paths	A modified path (a/nx)	Single path	1/3	No
		A set of paths	1/3	No
		All paths	1/3	No
	A set of modified path (a/nx;b/ny)	A set of paths	1/2	No
		All paths	1/2	No
	All paths (a/nx;b/ny;c/nz)	All paths	1	Yes

Figure 16: The Interpretations of the Examples by the Single Point Modifications of Branches

Modification	Example	Interpretation	Comb.	Int.
Before the cycle	No iteration (ac/xnz)	Before the cycle	1/2	Maybe
		After the cycle	1/2	No
	One iteration (abc/xnyz)	Before the cycle	1/2	No
		In the beginning of the cycle	1/2	Maybe
	Two iterations (abbc/xnyyz)	Before the cycle	1	Yes
In the beginning of the cycle	No iteration (ac/xz)	In the beginning of the cycle	1/2	No
		In the end of the cycle	1/2	Maybe
	One iteration (abc/xnyz)	Before the cycle	1/2	Maybe
		In the beginning of the cycle	1/2	No
	Two iterations (abbc/xnynyz)	In the beginning of the cycle	1	Yes
In the end of the cycle	No iteration (ac/xz)	In the beginning of the cycle	1/2	No
		In the end of the cycle	1/2	No
	One iteration (abc/xynz)	After the cycle	1/2	No
		In the end of the cycle	1/2	Maybe
	Two iterations (abbc/xynynz)	In the end of the cycle	1	Yes
After the cycle	No iteration (ac/xnz)	Before the cycle	1/2	Maybe
		After the cycle	1/2	No
	One iteration (abc/xynz)	After the cycle	1/2	No
		In the end of the cycle	1/2	Maybe
	Two iterations (abbc/xyynz)	After the cycle	1	Yes

Figure 17: The Interpretations of the Examples by the Single Point Modifications of a Cycle

Example	Interpretation	Int.
A modified path	A single path	Yes
	A set of paths	No
	All paths	Maybe
A modified path + an unmodified one	A single path	No
	All paths except the unmodified one	Yes
A set of modified paths	A set of paths	Yes
	All paths	Maybe
All paths	All paths are modified as they are shown	Yes

Figure 18: Summary of the Interpretations

The first column contains the label of modified transducer. The second column describes the considered examples, demonstrating the example (we not list the irrelevant examples like paths only covering the unmodified part of the transducer). The third column the possible interpretations of the example (we also assume that the modification is one of the listed transducers). The last two columns of the table contain two values: combinatorial unambiguousness (comb.) and intuitive unambiguousness (int.). The combinatorial unambiguousness is a value between 0 and 1 showing the probability of choosing the same interpretation as the selected modified transducer. The intuitive unambiguousness is a value out of the following three possibilities: yes, no, or maybe which can be interpreted numerically as e.g., 0.8, 0, and 0.2, respectively. This value reflects whether we find the selection of the interpretation intuitive. In Figure 18 we can see the summary of the examples and the possible interpretation without redundancy. We have to notice that if the example is described as "a set of paths is modified", the example can be interpreted as "all except the shown unmodified path" for the general case (more than three paths).

In Figure 19 two graphs are shown as an example of two transducers behaving equivalently, i.e. they generate the same output for any valid input. According to their behaviors we can form equivalence classes of such graphs. If this kind of variations is possible, we always prefer the graph which has the smallest number of states.

Generalization To define metrics of the quality of the transducer and example pairs we have to generalize the concepts of combinatorial unam-

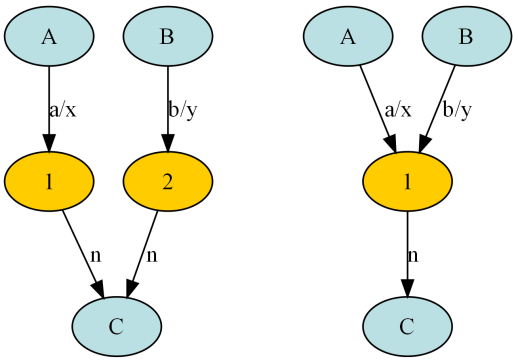


Figure 19: Two Transducers with Identical Behavior

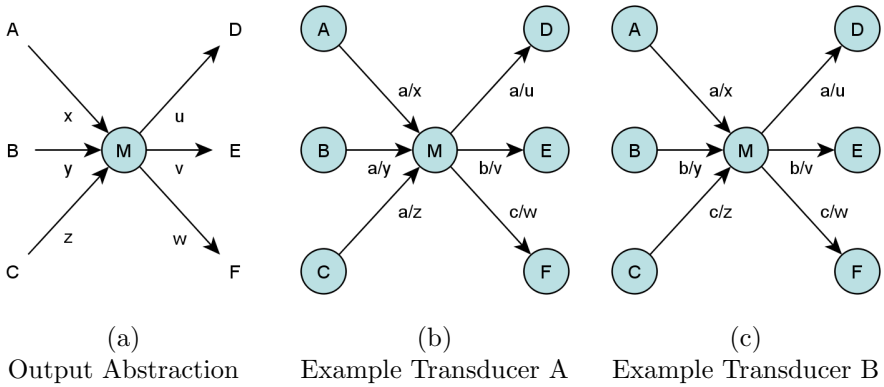
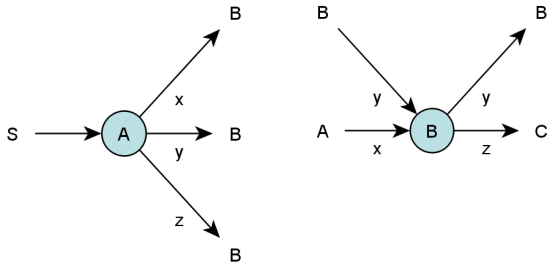


Figure 20: The Output Abstraction of the Transducers

biguousness and the intuitive unambiguousness for any arbitrary transducer and example. We studied how the values corresponding to single point modifications are calculated. We want to identify the building blocks similar the examples we studied and combine the values calculated for each building blocks. Intuitively, we expect that the value respect to multiple modifications is the product of the values of the single point modifications. (The values calculated for single point modifications are essentially probabilities and the combination of independent events can be calculated by multiplying the probability of each event.) We consider as the basic building block of such metrics the set of states around which the insertion must occur.

In Figure 20 one can see three graphs. In Figure 20a one can see a graph which is the output abstraction of the transducers in Figure 20b and



(a) Abstraction of the Branches

(b) Abstraction of the Cycle

Figure 21: The States to which the Insertions Belonging

Figure 20c. The edges and the nodes are labeled with output symbols and states, respectively. The graph in Figure 20a can be the abstraction of any deterministic transducer having the same states and transitions producing the same outputs symbols. In Figure 21a and in Figure 21b one can see the output abstractions of Figure 14a and Figure 15a, respectively.

Consider an arbitrary but fixed transducer t of which the Figure 20a is an output abstraction. Let i be an input for which t produces a trace between the states A, M, and F. In this trace the output symbols of two trace steps are x and w. If we specify a modification by x, n and w in which n is a new output symbol of a newly inserted edge, the location of the insertion must be around M (before or after this state). If the node has only a single in and out edge there are only two possible ways to insert the new edge which prints n (the decision is irrelevant with respect to the behavior of the modified transducer). In our case we may interpret the modification intention of t in several ways:

- a new state has to be inserted with label N, a new transition has to be inserted from state N to M reading nothing and writing n, the target state of the transition printing x has to be modified from M to N;
- a new state has to be inserted with label N, a new transition has to be inserted from state N to M reading nothing and writing n, the target state of the transition printing x has to be modified from M to N, the target state of the transition printing y has to be modified from M to N;
- a new state has to be inserted with label N, a new transition has to be

inserted from state M to N reading the same symbol as the transition from M to F and writing n, the source state and the input symbol of the transition printing w have to be modified to N and to empty, respectively;

- etc.

The amount of the possible variations can be reduced by showing further paths through the node M in the example. We can summarize the facts assuming that the output abstraction graph has n in edges and m out edges:

- on both sides 2^n and 2^m permutations of inserting single point modifications are possible;
- by constraining the number of modifications to one, then $2^n - 1 + 2^m - 1$ permutations are possible (e.g.: consider Figure 20a to which we want to insert a single point modification before (or after) the node M i.e. insert a new edge m between x, y, z, (or u, v, w) and M; this is possible by connecting a new node N to the M by edge m; then we can connect x, y, z to the node N or to the node M; we can do this in $2^3 - 1$ different ways; we can insert the edge n between the node M and the out-going edges (u, v, w) in a similar way, which leads to additional $2^3 - 1$ permutations; so, the number of possible ways to insert a single point modification into this example is $2^3 - 1 + 2^3 - 1$ i.e. 14);
- the number of all possible paths are n*m.

The metrics of the quality of the example with respect to expressing the intention of the specifier can be imagined as the following. According to the above description we can calculate the possible number of interpretations in the combinatorial sense, we call this metrics the combinatorial unambiguousness of the example. We can assign a weight to the selected interpretation according to the likeliness of the interpretation in sense of human intuition (to determine the correct weight we need to further study the psychological aspect of such preferences). This metric can be called intuitive unambiguousness.

The combinatorial unambiguousness metrics can be used to evaluate the quality of the examples with respect to a given transducer. If the example is ambiguous i.e. the value of the combinatorial unambiguousness is less than one, the algorithm may ask questions to clarify the intention or require the specifier to annotate the example by information that indicates which modification is expected to be generalized (the annotation can be interpreted as a selection function for the surrounding edges). Calculating the intuitive

unambiguousness of the interpretations can help to select the more intuitive solution from the many possibilities if no interaction with the specifier is allowed. These metrics may also help to define the expected behavior of the algorithms.

6 Conclusions

In this paper we investigated the theoretical background of model transformation modification by example. The paper presents our motivation to develop such an algorithm and the idea of modeling transformations with transducers. We formulated the theoretical problem in a formal way. The main contributions of this paper are the detailed description of an inference algorithm and the proposed metrics to evaluate the effectiveness of similar algorithms.

An M2T transformation language (XSLT, JET, etc.) can be mapped to transducers. Such a mapping needs the identification of the control flow structure and the output operations. The details of mapping XSLT to transducers can be found in [26] (in the paper we call the transducer abstract control-flow graph).

In a subsequent paper we will discuss a new algorithm constructed according to the lessons learned by constructing this algorithm. We are also going to revisit the question of the metrics and to compare the two proposed algorithms.

References

[1] Code Generation Information for the Pragmatic Engineer. http://www.codegeneration.net/.

[2] Helena Ahonen. Generating Grammars for Structured Documents Using Grammatical Inference methods. *Ph.D. thesis, University of Helsinki*, 2004.

[3] Dana Angluin. Negative results for equivalence queries. *Mach. Learn.*, 5(2):121–150, 1990.

[4] Zoltán Balogh and Dániel Varró. Model Transformation by example using inductive logic programming. *Software & Systems Modeling*, 8(3):347–364, 2008.

[5] Marc Bezem, Christian Sloper, and Tore Langholm. Black Box and White Box Identification of Formal Languages Using Test Sets. *Grammars*, 7:111–123, 2004.

[6] S.P. Caskey, E. Story, and R. Pieraccini. Interactive grammar inference with finite state transducers. In *Proc. Automatic speech recognition and understanding (ASRU-03), IEEE Workshop*, pages 572–576, 2003.

[7] Kwang Ting Cheng and A. S. Krishnakumar. Automatic functional test generation using the extended finite state machine model. In *Proc. 30th International Conference on Design Automation*, pages 86–91, New York, USA, 1993. ACM Press.

[8] Hubert Comon, M. Dauchet, R. Gilleron, Florent Jacquemard, Denis Lugiez, Sophie Tison, and Marc Tommasi. Tree automata techniques and applications. *Available on: http://www. grappa. univ-lille3. fr/tata*, 1997.

[9] Rafael Corchuelo, José a. Pérez, Antonio Ruiz, and Miguel Toro. Repairing syntax errors in LR parsers. *ACM Transactions on Programming Languages and Systems*, 24(6):698–710, November 2002.

[10] J.R. Curran and R.K. Wong. Transformation-based learning for automatic translation from HTML to XML. In *Proc. 4th Australasian Document Computing Symposium (ADCS99)*, 1999.

[11] Colin de la Higuera. *Grammatical Inference: Learning Automata and Grammars*. Cambridge University Press, 2010.

[12] Brian Demsky and Martin C. Rinard. Automatic Detection and Repair of Errors in Data Structures. In *Proc. ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 78–95. ACM, 2003.

[13] Pankaj Dhoolia, Senthil Mani, Vibha Singhal Sinha, and Saurabh Sinha. Debugging Model-Transformation Failures Using Dynamic Tainting. In *Proc. 23rd European Conference on Object-Oriented Programming*. Springer, 2010.

[14] D. Drusinsky. Model checking of statecharts using automatic white box test generation. In *48th Midwest Symposium on Circuits and Systems, 2005.*, pages 327–332. IEEE, 2005.

[15] Alpana Dubey, Pankaj Jalote, and S.K. Aggarwal. Inferring Grammar Rules of Programming Language Dialects. In *Proc. 8th International Colloquium on Grammatical Inference. LNCS*, volume 4201, page 201. Springer, 2006.

[16] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-based repair of complex data structures. In *Proc. 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 64–73. ACM, 2007.

[17] J Engelfriet. The Equivalence of Bottom-Up and Top-Down Tree-to-Graph Transducers. *Journal of Computer and System Sciences*, 56(3):332–356, Juni 1998.

[18] M. Erwig. Toward the automatic derivation of XML Transformations. In *Conceptual Modeling for Novel Application Domains, ER 2003 Workshops ECOMO, IWCMQ, AOIS, and XSDM. LNCS*, volume 2814, pages 342–354. Springer, 2003.

[19] Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, and Clémentine Nebut. Metamodel matching for automatic model transformation generation. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*, volume 5301 of *Lecture Notes in Computer Science*, pages 326–340. Springer, 2008.

[20] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach.* PWS Publishing Co., Boston, MA, USA, 1998.

[21] Iván García-Magariño, Jorge J. Gómez-Sanz, and Rubén Fuentes-Fernández. Model Transformation By-Example: An Algorithm for Generating Many-to-Many Transformation Rules in Several Model Transformation Languages. In *ICMT '09: Proceedings of the 2nd International Conference on Theory and Practice of Model Transformations*, pages 52–66, Berlin, Heidelberg, 2009. Springer-Verlag.

[22] M. Garofalakis, A. Gionis, R. Rastogi, S. Seshadri, and K. Shim. XTRACT: Learning Document Type Descriptors from XML Document Collections. *Data mining and knowledge discovery*, 7(1):23–56, 2003.

[23] Minos Garofalakis, Aristides Gionis, R. Rastogi, S. Seshadri, S. Genomics, and K. Shim. DTD inference from XML documents: the XTRACT approach. *IEEE Data Engineering Bulletin*, 26(3):18–24, 2003.

[24] Jonathan Graehl, Kevin Knight, and Jonathan May. Training Tree Transducers. *Computational Linguistics*, 34(3):391–427, September 2008.

[25] Jack Greenfield, Keith Short, Steve Cook, and Stuart Kent. *Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools*. Wiley, 2004.

[26] Gábor Guta, András Pataricza, Wolfgang Schreiner, and Dániel Varró. Semi-Automated Correction of Model-to-Text Transformations, 2010. Submitted to *ME2010*.

[27] Gabor Guta, Wolfgang Schreiner, and Dirk Draheim. A Lightweight MDSD Process Applied in Small Projects. *35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 255–258, 2009.

[28] Gabor Guta, Barnabas Szasz, and Wolfgang Schreiner. A Lightweight Model Driven Development Process based on XML Technology. Technical Report 08-01, RISC Report Series, University of Linz, Austria, March 2008.

[29] Bertin Klein and Peter Fankhauser. Error Tolerant Document Structure Analysis. *International Journal on Digital Libraries*, 1(4):344–357, 1998.

[30] Anneke G. Kleppe, Jos Warmer, and Wim Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley, 2003.

[31] D. Lee, M. Mani, and M. Murata. Reasoning about XML Schema Languages using Formal Language Theory. Technical Report RJ 95071, IBM Almaden Research Center, November 2000.

[32] RJ Miller, YE Ioannidis, and R. Ramakrishnan. Schema Equivalence in Heterogeneous Systems. *Information Systems*, 19(1):3–31, 1994.

[33] Glenford J. Myers. *The Art of Software Testing, Second Edition*. Wiley, 2 edition, June 2004.

[34] P. Oncina, J.; Garca. *Identifying regular languages in polynomial time*, chapter -. World Scientific Publishing, 1992.

[35] L. Popa, M.a. Hernandez, Y. Velegrakis, R.J. Miller, F. Naumann, and H. Ho. Mapping XML and relational schemas with Clio. *Proc. 18th International Conference on Data Engineering*, pages 498–499, 2002.

[36] John W. Ratcliff and David Metzener. Pattern Matching: The Gestalt Approach. *Dr. Dobb's Journal*, (July):46, 1988.

[37] Jonathan Riehl. Grammar Based Unit Testing for Parsers. *Master's thesis, University of Chicago*, pages 1–22, 2004.

[38] S. Schrödl and Stefan Edelkamp. Inferring Flow of Control in Program Synthesis by Example. volume 1701, pages 171–182. Springer, 1999.

[39] David Smith C. Pygmalion: A Creative Programming Environment. Technical Report ADA016811, Department of Computer Science, Stanford University, 1975.

[40] Yu Sun, Jules White, and Jeff Gray. Model transformation by demonstration. In *MODELS '09: Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 712–726. Springer, 2009.

[41] Nobutaka Suzuki and Yuji Fukushima. An XML document Transformation algorithm inferred from an edit script between DTDs. In *Proc. 19th Conference on Australasian Database*, volume 75, pages 175–184. Australian Computer Society, Inc., 2008.

[42] S.D. Swierstra and P.R.A. Alcocer. Fast, Error Correcting Parser Combinators: A short Tutorial. In *SOFSEM99: Theory and Practice of Informatics. LNCS*, volume 1725, page 763. Springer, 1999.

[43] Matej Črepinšek, Marjan Mernik, Barrett R. Bryant, Faizan Javed, and Alan Sprague. Inferring Context-Free Grammars for Domain-Specific Languages. *Electronic Notes in Theoretical Computer Science*, 141(4):99–116, 2005.

[44] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. Towards Model Transformation Generation By-Example. In *Hawaii International Conference on System Sciences*, volume 40, pages 47–70, 2007.