JOHANNES KEPLER
UNIVERSITÄT LINZ | JKU

Technisch-Naturwissenschaftliche
Fakultät

Model-to-Text Transformation Modification by Examples

DISSERTATION

zur Erlangung des akademischen Grades

Doktor

im Doktoratsstudium der

Technischen Wissenschaften

Eingereicht von:
Gábor Guta

Angefertigt am:
Institut für Symbolisches Rechnen

Beurteilung:
A. Univ.Prof. Dipl.-Ing. Dr. Wolfgang Schreiner (Betreuung)
Dr. Dániel Varró, PhD, Habil. (Associate Professor)

Linz, April, 2012

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Dissertation selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Die vorliegende Dissertation ist mit dem elektronisch übermittelten Textdokument identisch.

Abstract

In software industry model transformations are used in a more and more extensive manner. *Model Transformation* is about transforming a high level model into a different or more detailed representation. These transformations form the backbone of the *Model Driven Software Engineering* approach in which the development process is about to transform a high level model in several steps into executable software. *Model-to-Text Transformations* are a type of model transformations in which the result of the transformation is source code or configuration text. It is an active area of research on how the transformations can be created and modified effectively.

We propose a new approach to model-to-text transformation modification to ease the enhancement and modification of the existing transformation programs. In our approach the transformation can be modified by providing example input/output pairs of the expected new behavior. The initial transformation is typically derived from the source code of a hand crafted prototype, which is an implementation of the example model. Several minor modifications are made on the transformation before it can generate the expected source code from the example models. This thesis is motivated by a project in which we have developed transformations iteratively and we concluded that the most part of the transformation development can be automated.

To eliminate the complexity of a full featured transformation language we map the model-to-text transformation problem to a representation based on a special kind of automata called *transducers*. Furthermore, we aid the inference of the transducers from positive examples with an initial solution provided by the developer. We propose two algorithms to solve this inference problem. The first algorithm is insufficient for practical applicability. The lesson learned from construction of the first algorithm helped us to develop the improved second algorithm. To assess the quality of inference algorithms, we also define measurement methods which show that the second algorithm indeed performs better than the first one. Finally we apply our approach to building a complete system which is capable to infer modifications of XSLT programs.

Zusammenfassung

In der Software-Industrie werden Modell-Transformationen in einer immer umfangreicheren Weise verwendet. Eine *Modell-Transformation* ist eine Umwandlung eines auf einer hohen Ebene befindlichen Modells in eine andere oder detailliertere Darstellung. Solche Transformationen bilden die Basis des Ansatzes der *Modell-getriebenen Software-Entwicklung*, bei der der Entwicklungsprozess darin besteht, ein Modell von einer hohen Ebene in mehreren Schritten in ausführbare Software zu verwandeln. *Modell-zu-Text-Transformationen* sind eine Art von Modell-Transformationen, bei der das Ergebnis der Transformation Quellcode oder Konfigurationstext ist. Die Frage, wie solche Transformationen effektiv erstellt und bearbeitet werden können, stellt ein aktives Feld der Forschung dar.

Wir schlagen einen neuen Ansatz für die Modifikation von Modell-zu-Text-Transformationen vor, um die Erweiterung und Änderung bestehender Transformations-Programme zu erleichtern. In unserem Ansatz kann eine Transformation durch die Bereitstellung von Paaren von Eingaben und Ausgaben des erwarteten neuen Verhaltens verändert werden. Die anfängliche Transformation wird in der Regel aus dem Quellcode eines handgearbeiteten Prototyps hergeleitet, der das Beispiel-Modell implementiert. Diese Transformation wird in kleinen Schritten verändert, bevor sie den erwarteten Quellcode aus den Beispiel-Modellen generieren kann. Diese Dissertation wird durch ein Projekt motiviert, in dem wir Transformationen iterativ entwickelten, und in dem wir zu dem Schluss kamen, dass die meisten Teile der Entwicklung der Transformation automatisiert werden können.

Um die Komplexität einer voll ausgestatteten Transformations-Sprache zu vermeiden, bilden wir das Problem der Modell-zu-Text-Transformation in eine Darstellung auf der Basis von speziellen Automaten, so genannten *Transducern*, ab. Dann unterstützen wir die Herleitung von Transducern aus positiven Beispielen durch eine anfängliche Lösung, die durch den Entwickler zur Verfügung gestellt wird. Wir schlagen zwei Algorithmen vor, um dieses Herleitungsproblem zu lösen. Der erste Algorithmus reicht für die praktische Anwendbarkeit nicht aus. Die Lehren aus der Konstruktion des ersten Algorithmus helfen uns aber, einen verbesserten zweiten Algorithmus zu entwickeln. Um die Qualität der Herleitungsalgorithmen zu beurteilen, definieren wir anschließend Messmethoden, die zeigen, dass der zweite Algorithmus in der Tat eine bessere Leistung als der erste aufweist. Schließlich wenden wir unseren Ansatz auf den Aufbau eines kompletten Systems an, das in der Lage ist, Änderungen von XSLT-Programmen herzuleiten.

Acknowledgments

I would like to say thank you to my supervisor Wolfgang Schreiner. I learned from him how to explain and write down ideas in a form which can be understood by others. I think being my supervisor was a real challenge for him, because I was the person who has never chosen from the offered topics, but proposed a completely different one. He always managed my "green horn" behavior very well and provided me the freedom to make my own decisions even if it meant much more work for him.

I thank my co-supervisor Dániel Varró for teaching me how to approach problems from an engineering viewpoint. I also thank András Patarica for providig me a unique opportunity to be a guest at his group.

I am thankful to János Szilágyi at DLK Informatik for organizing challenging projects. I thank my colleagues at SCCH, namely: Bernhard Moser, Holger Schöner, Klaus Wolfmaier and Klaus Pirklbauer; for providing me the possibility to finish my dissertation. I am grateful to Mátyás Arató, Gábor Fazekas, Magda Várterész and István Juhász at University of Debrecen for helping me in the steps towards to post-graduate studies.

Finally, I thank to my parents for their support and to my girlfriend Anikó Tordai for her patience.

This work is supported by the Research Institute for Symbolic Computation and by the Upper Austrian Government.

Contents

Chapter 1

Introduction

1.1 Motivations and Goals

Software industry has always had an interest of producing software in a more effective way. One of the approaches towards this goal is the generation of the software from a model. The advantage of this concept is that the model can be represented close to the problem domain while the final program is generated by a transformation program. The concept leads us to *Model Driven Development (MDD)*.

MDD aims to develop software by describing the system by models and transforming them into executable code, rather than writing code directly [17]. By the term "model" we understand a high level (formal) representation of all the important aspects of a system. By separating the model from the implementation (code) the modification and maintenance of the model is more straightforward, because implementation details and platform-specific optimizations are kept separated from the business logic embedded in the model. The model in higher abstraction level can be analyzed and verified easier than the detailed implementation. MDD can also result in a higher level of optimization, because the model is at the right abstraction level and contains no early implementation decisions which can conflict with the automatic optimization during the transformation process [108]. *Model-to-Text (M2T)* transformations are used to transform models to text like artifacts e.g. source code, report, configuration descriptions.

Back in 2006, prior to the current line of research, we attempted to use MDD in a medium size project in an industrial context at the company DLK Informatik where the project team used a custom iterative software development process. We did not find any off-the-shelf tool which matched to our problem domain and implementation platform. Thus, we decided to develop our own model transformations. We had to integrate both the development of the model transformations and their usage. This setup also necessitated to carefully select an appropriate development process which supported the simultaneous development and use of MDD artifacts. We concluded that our situation was not unique and deserved the effort to develop a generic solution; this became a practical motivation for starting the scientific investigations in the current thesis.

Our first goal was to construct a software development process for small and medium sized development teams which integrates MDD with an iterative process. The advantage of the iterative development process is that the project team always plans for short iterations with well understood goals and clear requirements of what must be delivered. Most of the MDD approaches expose two major problems: fisrst, a steep learning curve prevents the team to deliver products in short iterations especially at the beginning; second, tools do not have robust round-trip engineering support i.e. it is hard to synchronize the changes of the models the transformations and the generated source code.

We recognized that the way our model transformations were developed might be carried out semi-automatically if we developed an appropriate tool. The development of a model transformation can be started with the construction of a domain model from the key elements of the problem domain and with the development of a program prototype which realizes one's domain model on the implementation platform. The development of the model transformations is primarily guided by the constructed small domain model and by the expected output i.e. our program prototype. The model transformations often need corrections, because their results may contain minor differences (typos, inconsistent changes, etc.) compared to the expected output. During our industrial projects, we often felt that most of the corrections were simple manual work which might be carried out automatically.

We found that the iterative extensions of the model transformations might be improved by tool support in a way that is similar to the corrective activities during the development of the initial transformations. The modifications which affect only the implementations (e.g. improvement of the user interface layout) are first implemented in the prototype program then the transformations are modified to generate code which is equivalent with the program. These modifications hardly ever changed the structure of the transformation but mainly affected the generated text.

Our second goal is to prove that it is possible to construct a tool which supports the modification of M2T transformations according to example input model and expected output text pairs.

1.2 Originality of the Thesis

In this section we present the innovations described in this thesis. In the following, we summarize each innovation and describe the applied research methods, our results and their potential applications.

A Lightweight MDD Process MDD was promoted around 2003 as the next big paradigm shift in software engineering [108, 117]. Since that time MDD has not been adapted widely and most of the success reported from enterprise and embedded context. Most of these solutions suffer from the high development effort of the initial creation and maintenance of the model to source code transformation [83, 117]. We investigated whether the MDD or its parts can be applied in a lightweight manner i.e. the project team must be able to perform with MDD at least as efficiently as they were using their regular development tools even in the first project. We evaluated the capability of the available tools and also studied the literature carefully. We came up with a process and selected an implementation tool chain, which we tested in an industrial project.

The author developed and tested a novel lightweight development process to implement MDD in small-medium sized industrial projects. The practical relevance of our results is that we adapted MDD to make it applicable for an average small project team without high initial investments. The XML based realization was used in two projects: one project was a larger project which was described in detail later; the other project was a small project in which we tested how efficiently we can reuse existing templates in a small project. The results were published as a technical report [66]; a short conference paper was written that focuses on the process aspect [65]. The developed process can be applied in similar projects in the future.

Modeling M2T Transformation by Finite State Transducer In order to be able to study how the model-to-text transformations can be modified automatically according to examples we needed a simple way to represent the transformations. The most popular dedicated transformation languages are full featured programming languages [4]. To focus on the core problem we had to eliminate the necessary complexity of the specific transformation language which we used. To represent the core problem in a formal (mathematically precise) way we looked for some sort of finite state automata, because this notation has a strong and extensive theoretical foundation. We finally selected transducers to model M2T transformations (transducer is automaton capable to produce output). We re-formulated the original problem as a finite-state transducer modification by example.

The author describes a mapping of the original problem to the finite state transducer inference from input/expected output pairs. The formulation of the problem is important for two reasons: with the help of formulating the problem with transducers we can use the accumulated knowledge in field of formal languages; and by redefining inference problems as *modification inference* problems new algorithms can be developed in which the inference can be guided by an approximative solution. This representation can be also used to other analyses of the transformations.

Algorithms to Infer Finite State Transducer Modifications We aimed to adapt or develop algorithms which can infer finite state transducer modifications from example input/expected output pairs. One potential research field in which we looked for solution was Grammatical Inference (GI). GI is about inferring grammars from the set of examples (grammars are often called schemas or meta-models by software engineers). Unfortunately, we have not found such algorithms as the research focuses on the initial creation of the grammars from examples. The other interesting research field is Model Transformation by Example (MTBE) which focuses on the inference of transformations on model transformation level. The results of this research are systems which are capable to infer simple mappings between models [52]. As we did not find an appropriate algorithm which properly handles M2T transformations we developed our own algorithms. We evaluated the algorithm by executing them on small number of hand-crafted examples. Experiments with this algorithm exhibited some deficits; thus, we developed a second algorithm to overcome these. To verify the capability of the second algorithm we evaluated over the same examples as we did with the first one. Then we tested both of them on hundreds of automatically generated examples.

The author developed two algorithms which can infer transducer modifications. By comparing the results it was demonstrated that the algorithm which used lazy evaluation of the modifications clearly outperformed the algorithm which modified the transducer immediately during the processing of the modifications. We use the one with lazy evaluation in our transformation correction tool support. The algorithms can be applied in similar systems for other transformation languages. Our algorithms may also serve as inspiration for implementing modification inference algorithms for other classes of automata. The two algorithms were published in chronological order as technical reports [62, 63].

Metrics to Express the Quality of Transducer Modifications We needed some objective measure to evaluate the quality of the algorithms. There were no prior results available on evaluation of the properties of modification inference. We defined some metrics to quantitatively express the effectiveness of such algorithms.

The author presents a set of metrics to measure the quality of transducer modification by example. These metrics are used to verify the performance of our algorithms. The

presented measurement method makes our algorithms comparable with the algorithms to be proposed in the future. Our measurement method is also adaptable to measure similar modification inference problems. The metrics were published in [62] and the method of creating measurements with the proposed metrics is presented in [63].

Tool for Semi-Automated Correction of M2T Transformations We developed a semi-automatic tool for M2T transformation to help the correction and extension of the transformations. Existing experimental solutions support only the initial creation of the transformation from the examples and the mappings are usually defined as model to model transformations [52]. We used this system to evaluate the practical usability of the transducer based algorithms.

The author presents a proof-of-concept tool which use one of our algorithm. The importance of this result is to prove the practical applicability of our theoretical results to a real world transformation language. We expect that our system can be extended for use in industrial environments. The results were presented in the Models and Evolution 2010 workshop of MODELS Conference [64].

1.3 Structure of the Thesis

In this section we present the structure of the thesis. The chapters present the innovation described in the previous section in logical order.

Chapter 2 (State of the Art) provides an overview of the state of the art in the related fields. This section starts with the summary of *Model Driven Development (MDD)* and model transformation (a key technology of MDD). We describe the available results of *Model Transformation by Example (MTBE)*, which is the closest research area to our results. Finally, we give an overview of *Grammatical Inference (GI)* as the research field to which our theoretical results can be connected.

Chapter 3 (A Lightweight MDD Process) presents the industrial context which motivated the author. We propose a development process to implement MDD in an iterative software development process. We also give a detailed description of the actual implementation based on XML technology. Finally, we describe how the explained process and technology was applied in a real-world project.

Chapter 4 (Finite State Transducer Modification by Example) describes the main idea of the thesis, which is the mapping of the "Model Transformation By Example" problem into the grammatical inference problem. We document two algorithms which can infer transducer modifications.

Chapter 5 (Evaluating the Inference Algorithms) presents an inference algorithm which infers modification of an existing transducer according to the examples of desired input/output pairs. We propose a basic set of metrics to evaluate the effectiveness of the two algorithms. We evaluate the two algorithms by executing them on examples and by calculating the described metrics on the results.

Chapter 6 (Semi-Automated Correction of M2T Transformation) demonstrates the practical applicability of the algorithms. It repeats the motivational example and

presents the technical details of how our tool inferences the modification semi-automatically.

Chapter 7 (Conclusion) summarizes our results presented in this thesis. It also provides an outlook on future research directions.

Chapter 2

State of the Art

In this chapter we overview the related work in three fields of research. In Section 2.1, we start with *Model Driven Development (MDD)* to understand the current trends and challenges of this field. As the transformation of models is a key concept in MDD, we examine in Section 2.2 a special way of defining a model transformation, namely *Model Transformation by Examples (MTBE)*. Finally, we give in Section 2.3 an overview of the field of *Grammatical Inference (GI)* which describes algorithms that generalize rules (grammars) from examples.

2.1 Model Driven Development

In this section we start with describing some aspects of MDD which can be used for comparison. Then, we describe the most well known paradigms and approaches. We also describe the development process necessary to use these approaches in a real life projects. Finally, we give a short summary of the model transformation.

2.1.1 Overview of Approaches

The idea of developing a model and transforming it into a more detailed model or an executable code appears in several other approaches [108]. One of these approaches is *generative programming* which is closely related to MDD. The main difference is that generative programming focuses on a family of systems, while MDD aims to develop an universal technology [31]. To focus on a family of systems, extensive domain modeling has to be done. Methodologies are proposed to guide the domain modeling. The domain models can be implemented with different programming techniques (Aspect-Oriented Programming, Generators, Static Meta Programming, etc.) [32]. Other similar approaches to MDD with different emphases exist, e.g.: Model-Integrated Computing [17], Software Factories [60], code generation [68].

These approaches can be compared based on how well they support the reuse of the created artifacts. To compare their reusability, the generative approaches are categorized into software schemas, application generators, very high level languages and transformational systems [82]. These categories (plus additional forms of reuse, e.g. component libraries) are compared according to the abstraction, selection, specialization, and integration mechanism. The conclusion is that generative approaches generally lower the cognitive distance between the problem domain and the model representation at the cost of requiring extra effort of turning this model to a runnable implementation (e.g. application generators must be developed by someone). A good summary on the representation of domain models can be found in [49]. This book introduces the term *external*

Domain Specific Language (DSL) as a model representation to distinguish from abstraction techniques which are provided generally by third generation programming languages (*internal DSL*).

Other advantage of MDD that the quality of the models can be verified early during the development. The verification in model level is typically a consistency check between the models e.g. "name of message must match an operation in receiver's class" [40] or "the events produced in a sequence diagram should not produce inconsistent states in the state diagrams of the objects which participate in the interaction" [89].

Model Driven Architecture Most of the literature referencing to MDD is connected to the Object Management Group's (OMG) *Model Driven Architecture (MDA)* [109, 117]. In OMG's MDA, the domain model is represented in the Unified Modeling Language (UML). Typically domain model representation is not pure UML, but extended with some extra notations called UML Profiles. MDA calls the initial model Computation-Independent Model (CIM) and defines two intermediate models: the Platform Independent Model (PIM) and the Platform Specific Model (PSM). In MDA the PSM is at the same abstraction level as the source code [50, 76]. MDA is driven by the challenge that complex system need to be developed with lower cost and higher quality [108, 109]. The generated complex systems must be capable to communicate with each other [50].

MDA paradigm is described mainly as a vision extended with some key technology. One of the key technology is the Meta Object Facility (MOF), which defines four levels of abstraction: M0 is the objects and data; M1 describes the model; M2 describes the metamodel (schema of the model); M3 is the meta-metamodel which defines the schema of the metamodel [50, 76]. Other key technologies are OMG's own transformation language (QVT) and interchange format (XMI) [8].

The literature referenced above focuses on the technology aspects. It does not give detailed descriptions on how the transformations are created or customized. It also does not provide detailed explanation about how the quality assurance activities (validation and verification) of the models, the transformations, and the resulting code can be carried on.

Software Factories Microsoft's Software Factory [59] initiative has a different approach. The models and transformations are described as critical but not as the only innovations to improve software reuse (other form of reuse are the software components, patterns, etc.). A product line contains similar software products for the same application domain. This initiative focuses on the details of forming product lines, i.e. building a domain specific abstraction and tools for transforming these abstraction into software. The steps of the process are product line analysis, design and implementation [60].

Model Integrated Computing In [17, 106] the Model-Integrated Computing (MIC) variant of MDD is described, which uses a Domain Specific Model Language (DSML) in contrast with MDA's UML. These DSMLs can be defined visually with the help of Generic Modeling Environment (GME). The authors developed several systems successfully with their MDD variant.

2.1.2 Supporting Methodology

The MDD process is described as a simplified iterative process (analysis, design, coding, testing, and deployment steps) with the focus on the required and delivered artifacts [93]. In MDA the delivered artifacts of the analysis and design activities must be the PIM and

PSM, respectively [76]. UML models (PSM) can be annotated in a way that they can be executed, such models can be turned into code by model compilers (model transformation) [93]. If the PSM can be transformed automatically to the program source code, no coding is required. Round trip engineering can be realized with some technical limitation [76]. Several tools exist which support the MDA or some key steps of it [116].

The key parts of the MDA paradigm are compatible with Agile Software Development, XP (eXtreme Programming) and Rational Unified Process (RUP) [50]. In [94] the authors introduce the term of Agile MDA. They claim that MDA is actually an Agile Software Development approach, because it is focused on executable models. These models are both close to view of the customers and considered as valuable as the source code due to their executability (according to the view expressed in the Agile Manifesto, the working software has higher business value than comprehensive documentation [19]). A code-oriented MDD [79] propose an Agile way to execute an MDD project: develope a prototype software by hand for evaluation; then create a model and transformations to reproduce the prototype; finally, construct the model and generate the required software.

As we have seen that most of the resources available for practitioners focus on the technological aspects and give no or little hint about how MDD can be integrated into the software development process. The process, if mentioned, is explained in the context of technological steps. The process must deal with the details of the construction of the meta-models and transformations. It must specify when and how the artifacts (models, code, meta-models and transformations) can be modified. In [26] we get a good overview of development methods for MDD namely: MODA-TEL [55], MASTER [86], MIDAS [22], C^3 [69], ODAC [56], and the authors' own methodology. Some of the methodologies explicitly divide the process into two main parts: first a preparation phase in which the meta-models and transformations are created, then an execution phase in which the model is created, the code is generated, integrated and tested (MODA-TEL, C^3, methodology in [26]). Other methodologies assume the prior existence of the meta-models and the transformations. The methodologies mentioned in this section target enterprise environments with sophisticated, customizable methodologies [92]. The case study [83] even explicitly states that the MDA technologies are appropriate only for large organizations. This opinion is also agreed by certain practitioners [117]. Other case studies also provide no comparison of MDD development effort vs. traditional development effort as they measure the success in the ratio of the generated and hand-written code size [51, 90, 121].

2.1.3 Model Transformation

Many software developers simply transform models and generate code with the help of dynamic or transformation languages without meta-models and sophisticated theoretical background. They generate source code of an application or configuration information from some compact domain specific model representation [3, 67]. This is necessary, because the widely accepted enterprise software architectures such as JEE and .NET require huge amount of code and configuration, which is in most of the cases error prone and boring to write by hand [11, 44, 70, 102].

Scripting Languages One can build a complete application generator in Ruby which is capable to generate DB schema, data access layer, business logic, UI, and RPC interface [68]. The author also provides practical advices e.g. how the generated and the hand crafted source code can be mixed.

XML Transformation Languages XSLT, an XML transformation language, is used also successfully for model transformation and for code generation [115]. A detailed comparison of XSLT with 3GL programming languages can be found in [105]. XSLT is also powerful enough to be used in a high-end MDA tool, namely in Borland's Together 2007 DSL Toolkit which uses a special extension of XSLT to generate source code from models [2].

Model Transformation Languages As model transformation is the "the heart and soul" of MDD [110] several specialized transformation languages developed. The model transformation languages can be categorized according to several categories [33, 95]. It is important what kind of transformation we would like to carry on: type of the source and target of the transformation (e.g.: model-to-model, model-to-text, etc.); whether the type is vertical (e.g.: model refinement, code generation, etc.) or horizontal (e.g.: refactoring, optimization, etc.); quality attributes of the transformation tools (e.g.: level of automation, testability, usability, etc.); representation of the transformation program (declarative vs. operational).

Eclipse Modeling project collects the most popular open source model transformation tools for MDD [4]. Its core is the Eclipse Modeling Framework which provides tools for build modeling applications [112]. The project contains tools for Model to Model transformations e.g. ATL [72], VIATRA2 [119]; and for Model to Text transformations e.g. JET [7], Xpand [9]. Also research languages are available in the Eclipse Modeling projet: Epsilon [78] is a language family which contains several task-specific languages on top of Epsilon Object Language, Henshin [16] which is a high performance transformation language of which notation base on attributed graph transformations.

2.2 Model Transformation by Example

In this section we start with a short introduction to the roots of the *Model Transformation by Example (MTBE)* approaches. The general form of this approach is programming by examples which dates back to as early as 1975 [111]. Then, we give a detailed description of the main MTBE approaches. Finally, we mention some loosely related applications of the MTBE approach.

Program Synthesis Program synthesis aims to generate programs from input/output examples. Two main approaches are known: functional program inference form program traces and logic based approaches e.g.: *Inductive Logic Programming (ILP)* [47]. The synthesis of algorithms typically contains two main steps: recording the trace (intermediate steps) of turning the input into the output and the generalization of the trace. In [107] an algorithm is described to infer a program from manipulating the content of variables in a graphical user interface. Similar methods can be applied to develop user interfaces or text editing macros [30].

Overview of the Approaches The idea of program synthesis can be extended to a special class of programs namely transformation programs. MTBE is about synthesizing programs from source and target model pairs to which additional information can be added e.g. prototype mapping or meta-models of the source and target models. The main motivation of MTBE is to hide the complexity of the meta-models and the transformation languages, because users are typically unfamiliar both with the concepts and with their notations [122].

Typically the user defines mappings between source and target models. The MTBE defines transformations by example mappings [18, 122]. The system described in [18] uses an ILP engine which handles negative examples to prevent overgeneralization. Transformation rules can be inferred with the help of *Rough Set Theory* [88] or *Relational Concept Analysis* [38] It can be also handled as an optimization problem [74], but this approach does not infer rules as the other approaches. In [52] an algorithm is described which is capable to convert N:M mapping examples into transformation rules. MTBE approach is used for data transformation in IBM's Clio tool [100].

A slightly modified approach to the problem called *Model Transformation by Demonstration* is closer to the trace based approach used in program synthesis. The modification steps which are done by the developer to convert the source model into the target model are recorded. These recorded steps can be turned to generic transformation rules [113].

Model transformations can be generated without prototype mapping assuming that nodes (and edges) are labeled with the same name in the source and the target model. In [45] models are interpreted as labeled graphs and the mapping of the nodes is inferred according to the labels of the edges. The same approach is applied to information schema mapping [96], XML *Document Type Definition (DTD)* mapping [43], and HTML to DTD mapping problem [114]. If one restricts the set of the elementary transformations to the ones which preserve the information content of the original model, the search space of the possible sequence of elementary transformations can be reduced. In this case the transformation can be learned by searching the right sequence of elementary transformations [29].

The above mentioned methods are still considered experimental, the authors suggest to review the inferred transformation manually. These methods are also not capable to handle complex attribute transformations and the degree of automatization needs improvement [18].

Alternative Applications Similar algorithms can be applied to the automatic correction of data structures [36, 41]. In such cases algorithm must find the transformation from the illegal state to the closest legal state of the data structure. In [37] a dynamic tainting technique is described to locate errors in the input *Text to Model (T2M) transformation*. This technique adds taints marks to the input which are traced to localize the input data which belongs to the incorrect output.

2.3 Grammatical Inference

In this section we give a detailed overview of the GI algorithms. Then, we give a short overview of the possible extensions of *Deterministic Finite State Automaton (DFA)*. Finally, we overview applications of GI algorithms.

Overview of the Algorithms An overview of techniques for inferring grammars can be found in [34]. Most of the knowledge is accumulated from the inference of regular languages over strings. Regular languages are recognized by DFA. Two big classes of algorithms exist: *generic search methods* (e.g. Genetic Algorithm [104], Tabu search [57], etc.) and *specialized algorithms* (e.g. RPNI [99], L-star [15], etc.).

The RPNI algorithm [99] 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 with respect to the given examples.

The L-Star algorithm [15] starts by 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. Then the algorithm asks an oracle whether the table contains the required grammar. If the answer is no, the oracle returns a counterexample. 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 . A more complex inference problem is posed by class of *Context-Free Language (CFL)*. One major problem is that one can only infer a *Context-Free Grammar (CFG)* instead of a CFL, because it is generally undecidable whether two CFGs are equivalent i.e. they define the same CFL. Learning CFGs is still not possible effectively [20]. Only special subclasses of CFGs can be learned in polynomial time e.g.: deterministic linear grammar. Another option is that one may use some generic search technique (e.g. Genetic Algorithms) to get some statistically approximate solution [34].

Extensions of the DFA In practice several extensions of DFA over Strings are interesting for researchers. DFA can be extended to form an abstraction of a general purpose programming language; e.g. in [25] an Extended Finite State Machine is described to model the languages VHDL and C. Probabilistic extensions (i.e. transitions are associated with probabilities) are used in natural language processing [91].

Another interesting way to extend an automaton is to define it over trees [27] or over graphs [103] instead of strings. Tree grammars are especially interesting because of the popularity of XML technologies. The Document Type Definition and the XML Schema are actually tree grammars [87].

Finally, an automaton can produce output. This kind of automaton is called *transducer* [34].

Several real life application need the combination of these extensions. Probabilistic tree transducers are defined and their properties are summarized in [58]. In [42] tree-to-graph transducers are defined and it is proved that the bottom-up and top-down tree-to-graph transducers define the same class of tree-to-tree translation.

Applications Grammatical inference can be used for schema learning of XML documents. One possible approach is to use the *Minimum Description Length (MDL)* principle to select the best schema from candidates produced by some heuristics [53, 54]. Another possible approach is to build the prefix tree acceptor i.e. an automaton which has tree shape; each example input is represented by a distinct path from the root of the tree; the paths share the same states and transitions as they have common prefix. The prefix tree acceptor can be generalized by merging states to accept strings which are not included in the example set[14]. The third possible approach is using *Genetic Algorithms* which are experimentally applied to learn CFGs of Domain Specific Languages [120].

A special area of GI is when small changes are inferred on an existing structure according to examples. One such application is the learning of the dialect of a programming language [39]. It is possible to infer small extensions of a large (200-300 rules) LR grammar with the help of the MDL principle. In [23] an inference algorithm is proposed to help the generalization of a grammar by improving its coverage by the examples.

Another possible application can be the automatic repair of parsing errors. In [28], the authors describe a heuristic which can correct most of the parsing errors by adding missing tokens or deleting unnecessary ones. In [75] a novel parsing strategy is described which can efficiently handle ambiguous parsing paths of error tolerant document structure analysis. Eclipse's Java parser is generated by LPG (formerly known as JikesPG) which also supports fully-automatic error recovery [24].

Chapter 3

A Lightweight MDD Process

In this chapter we present a lightweight, iterative, model driven development process which was implemented and tested in industrial projects. By the term lightweight we mean that the process makes it possible to use MDD only in specific parts of the development and it lets one start its usage incrementally. Besides the process itself, we also give a description of an example tool chain and architecture which we used with our process. Finally, we share the details of how the described process, the tool chain and the architecture were applied in a real-world project. The structure of this chapter is described at the end of the introductory section.

3.1 Introduction

MDD and DSL attract the attention of the industrial practitioners. Recently also more and more tools have become available to support these paradigms. Unfortunately, these paradigms are typically described in a water-flow style development process, which does not fit for small and medium sized agile teams.

Our goal is to fill this gap by proposing an MDD process which can be fitted to an iterative development. Most of the available MDD tools are complex and hard to customize. The proposed methodologies supporting MDD also target enterprise environments. Certain MDD experts even explicitly state that the MDA technologies are appropriate only for large organizations [117].

We give here a short overview about the constraints in small and medium sized development projects. The learning curve of the technology, the maturity of the tools, or the adaptability of the transformations to special requirements are probably the most crucial properties.

A lightweight approach for model driven development has to deal with the following issues:

- Most of the risks should be mitigated: The technology should not be based on immature tools and the introduction of the technology should always leave open the possibility to fall back to the traditional methodology without MDA tools.
- The approach should not delay the schedule in general. It should have immediate business advantage.
- The approach should be cost sensitive. Purchasing a high cost MDA tool set and extensive training of experts is typically not possible.
- The domain-model should be kept as simple as possible and the generated code should be human-readable and modifiable.

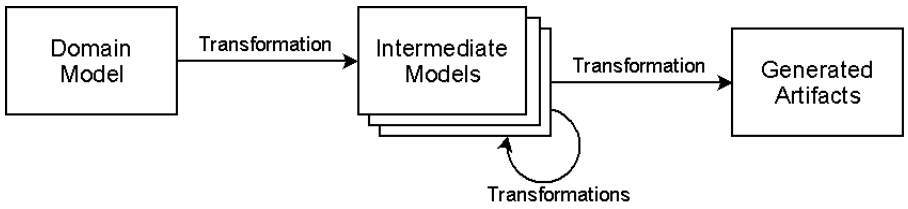


Figure 3.1: Model Driven Development

- The results should be reusable in other projects.

Two key properties make our process lightweight i.e. help to address the mentioned issues:

- The process explicitly stats that the only parts of the software generated with MDD where it brings enough advantages.
- The process encourages continuous integration of the MDD artifacts i.e. the development can be started with a minimalistic model and source code generation from the intermediate versions of transformations is required.

Both the process and the tool chain were developed in the end of 2006 and have been in use since that time. We present them here in a form that is the results of several improvements according to the project experiences.

The remainder of this chapter is structured as follows: In Section 3.2 we start with an overview of MDD. In Section 3.3 we describe our lightweight MDD process. In Section 3.4 we show how this process can be implemented by an XML-based tool chain. Finally, in Section 3.5 we describe how the process and the tool were applied in one of the projects. We also share our best practices which we consider necessary to carry through a successful MDD project.

3.2 MDD Overview

In this section we provide a deeper overview of MDD than we did in Section 1.1. MDD can generally be represented as shown in Figure 3.1. It starts with an abstract *domain model* from which different *generated artifacts*, like source code, are created by *transformation*. Transformations are also called *templates*, if the results are generated artifacts and not models. The MDD process can contain intermediate models with corresponding transformations. Models are typically represented in different abstraction levels. *Meta-models* can be assigned for the domain model and the intermediate models, which describe their meaning and the allowed constructs. The *domain model* can model a real life domain like a business process or can be an implementation model like a website navigation flow. Generally speaking, the different MDD approaches have to deal with the following questions:

- What is the domain model (real life/implementation)?
- How can the domain meta-model be represented?
- What kind of notation do we use to communicate the domain model?
- How are the transformations defined?

- Do we allow to edit the intermediate models or the generated artifacts?
- How does MDD affect the development process?
- Are we able to reverse the transformation (propagate changes of the generated artifacts back to the original model)?
- How can intermediate models be merged if the original generated model was modified and a different one was generated after the domain model was changed?
- What are the quality implications of the MDD? (How does it modify the test and review procedures? What kind of additional problem can be expected?)

In practice, the answers to these questions influence how one can integrate MDD technology to existing software processes. For small or medium size project teams with limited resources this issue is even more critical. To successfully apply a new technology in one's project there are several other factors that are independent from the theoretical soundness of the technology.

3.3 The Process Aspect

In this section we describe the process that we developed to help fulfill the requirements described in Section 3.1. Our main contribution is a process, which is described with the focus on the model driven aspects and presented in its abstract form.

One of the key ideas behind our approach is that we explicitly support the partial usage of MDD. Our process lets the developers to consider in which parts of the project the use of MDD will pay off and they employ the process just for these parts. The advantage of this approach is that the domain model can be easily kept focused and abstract. On the other side, keeping track of which artifacts are generated and which are not requires only minimal administrative overhead.

The process is defined as an extension of iterative methodologies in general [84]. It fits well in the existing methodologies like eXtreme Programming [5] or the Rational Unified Process [81].

3.3.1 Process Definition

In our development process we define two phases: the *initial phase* and the *development phase*. The *development phase* can be repeated arbitrarily many times. We introduce our own terms for the phases because different methodologies define varying numbers of phases and different terms and we do not want to stick to a particular methodology [85, 77]. For the same reason, we name our set of roles, artifacts, and activities distinct from those of other methodologies. We mark those definitions with a star (*) which are supposed to defined by an other particular methodology. On first reading, one may directly proceed to Section "The Process" and return to Sections "Roles", "Artifacts", "Activities" for definitions of the terms.

Roles

We define a role called *model driven development expert* (MDD expert). The responsibility of this role is to deal with the activities related to model driven technologies and support the managers, architects and developers. Certainly, in case of larger teams this role can be decomposed into more specific roles.

Artifacts

We use the following artifacts in the process description as it was defined in Section 3.2: *domain meta-model*, *domain model*, *generated artifacts*, and *templates*. We define some further artifacts as the following:

- *A MDD vision* is a document containing the strategic design decisions related to the MDD. It specifies which aspects of the requirement are implemented with MDD technology. It also prescribes the abstraction level of the domain model and its scope.
- *A code generation tool* is a collection of programs and scripts which runs the templates on the models.
- *A domain test model* is a special model which serves as a test for the generation process. It is an artificially created small model with good meta-model coverage. It is used by the developers and it does not represent any real life model.
- *A MDD environment* is a collection of a domain meta-model, a domain test model, and a Code Generation tool.
- *Hand crafted software* (*) is software or a collection of software fragments created with some traditional software development methodology. It becomes available short before the end of a phase. It can be a design prototype, a part of an internal release, or a part of a beta release.
- *Released software* is the software resulted by the integration of *hand crafted software* and *generated artifacts*.
- *A software requirement specification* (or *requirements in short*) (*) is an evolving document containing the requirements. In practice, other methodologies define other documents as well.

Activities

The following terms for the activities are used in the process:

- *Domain model extraction* is the activity which extracts the domain model from the requirement specification and form other documents. During this process the *MDD vision* and the initial *domain model* are created.
- *The MDD environment setup* is the activity when the *code generation tool* is set up according to the MDD vision and the initial domain model. As part of this activity, the initial version of the *domain meta-model* and the *domain test model* are created.
- *Domain Modeling* is an activity by which the *domain model* is extended and refined.
- *The MDD environment development* is the activity by which the *MDD environment* is further developed. This development activity is typically extensive and not corrective.
- *Template development* is the activity by which the templates are extended and refined. This activity usually happens in three steps: the required functionality is implemented in a software prototype; the implementation is extracted and to edited into the *templates*; finally, they are tested with the help of the *domain test model*.

- *Code generation* is the activity by which the *generated artifacts* are created with the *templates* from the *domain model*. The consistency of the domain model is always verified prior to the code generation.
- *Integration* is the activity by which the generated code and the hand-crafted software are merged. After the merge, the integration test is carried out.
- *Development* (*) is the activity when new functions are designed, implemented and tested. This activity is defined in more detail by particular methodologies.
- *Requirements Refinement* (*) is the activity when requirements are refined and extended.
- *Architectural prototyping* (*) is the activity by which the architecture of the software elaborated. In the end of this activity the architecture of the software is designed and a prototype which contains the initial implementation of that architecture is created.
- *Design* (*) is the activity when all kinds of design activates are carried through.
- *Development* (*) is the activity when all development related activates (class design, coding, and testing) are carried through.

The Process

Figure 3.2 gives an overview of the process. The artifacts are represented by boxes and the activities are represented by ellipses. Shading of the shapes means that they can be replaced by other activates or artifacts that are specific to a particular methodology and have the same purpose in the process as we prescribed (they are just included to help the understanding of the process). The flow chart is divided by dashed lines into three "swim-lanes". The "swim-lanes" represent three different internal teams, namely: *agile development team*, *business analyst team*, and *model driven development team*. The horizontal position of the activities expresses which internal teams they belong to. If an activity or an artifact is positioned between two swim-lanes, then it is a joint effort of the two teams.

- The *agile development team* can include roles like *developer*, *designer*, or *tester*. This team is responsible for the development of all non-generated software (functionality which is not represented in the domain model). The non-generated parts of the software can be the following: larger special components with custom functionality that are added only in a single situation or components that are used globally in the application like utility classes or authentication mechanisms.
- The *business analyst team* can include role like *business analyst* or *developer* who are responsible for business analysis activities. This team is responsible to keep in touch with the customer and is also responsible for the requirements and the domain models.
- The *model driven development team* includes *MDD experts* and developers. This team is responsible for the development of the MDD environment and the templates.

The vertical axis in Figure 3.2 denotes the time and the horizontal lines represent the borders of the phases. The figure shows the initial phase and two iterations such that the connections between phases are depicted. The vertical positions of the objects express

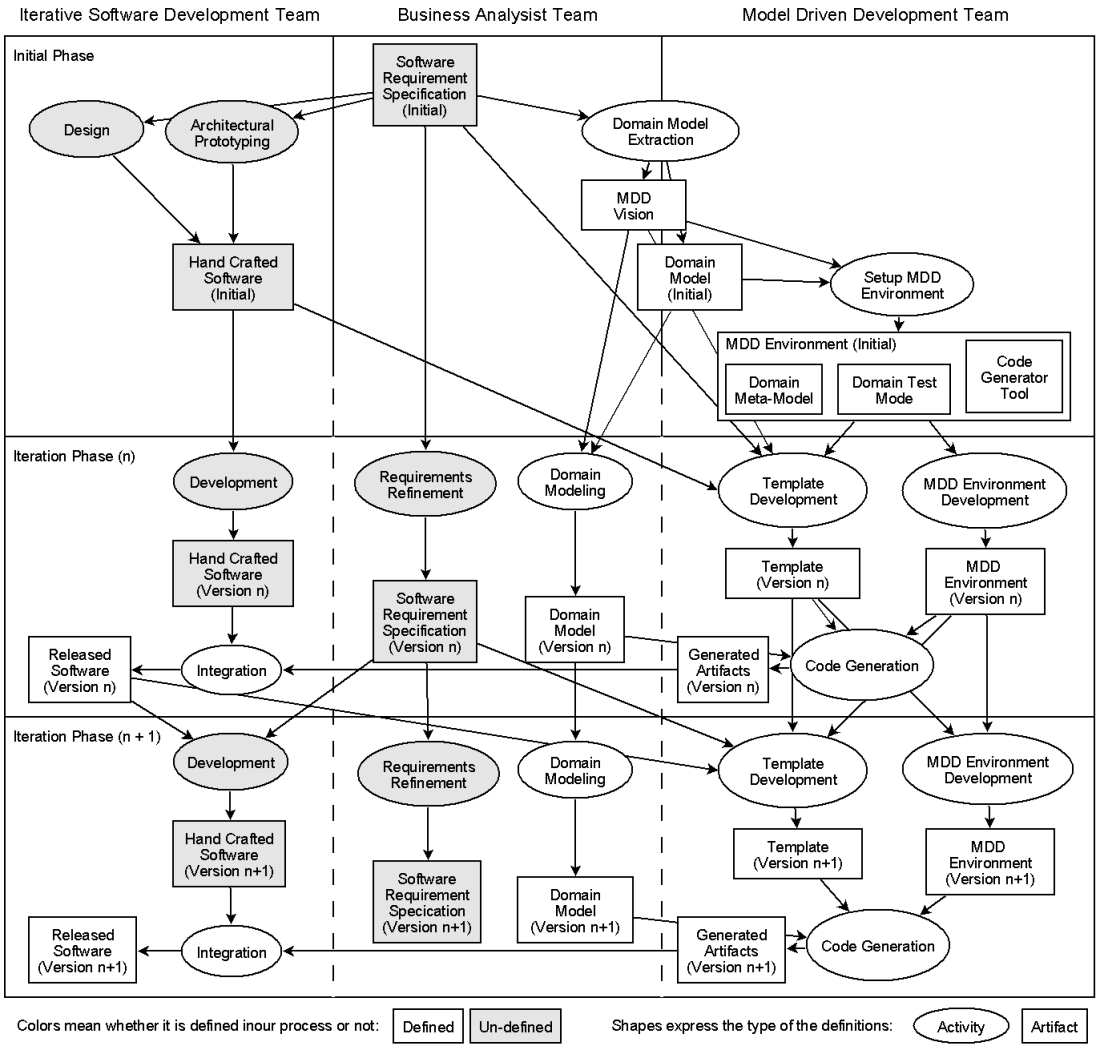


Figure 3.2: The Process

how they are related in time. The arrows also provide information about the dependency of two actions. If two objects are connected and they have the same vertical position, they follow each other immediately. In detail, the process proceeds as follows:

- The process starts with an initial phase (which can be preceded by phases defined by the methodology that our process extends). There are three groups of activities relevant in this phase:
  - The *business analysts team* and the *MDD team* extract the domain model by a joint effort. Domain experts from the customers can also contribute.
  - The *agile development team* elaborates the architecture of the software. By the end of the initial phase they ship a sufficiently stable prototype to the *MDD team* which starts to develop the transformation.
  - The *MDD team* sets up the initial version of the *code generator core*. This team also has to be familiar with the technology environment used by the *agile software team* in order to be capable to take over the architecture from the prototypes.
- During the iteration, the following activities are carried through in parallel:
  - The *business analyst team* refines and extends the domain model. The changes in the domain model do not affect the developers, thus,the model can be changed until the *code generation* activity starts. This team keeps in touch with the customer; its responsibility is to handle the change-requests of the requirements.
  - The *agile development team* develops the non-generated part of the software. The customization of the generated parts and the integration of the custom components are also the responsibility of this team. At the end of the iteration this team receives the generated artifacts and carries through the integration:
    - the generated artifacts are collected together with the hand crafted software;
    - in case an artifact generated in the previous iteration is modified by the agile development team, this modification is applied to the newly generated artifact too;
    - the whole software is built and finally tested.

    There are always cases when the newly generated artifacts need manual modifications or interface mismatches have to be corrected. These cases can be resolved in three different ways:
    - The hand crafted software is modified by *agile software team.*
    - The *MDD team* solves the problem in the next iteration; a temporary workaround is provided by the *agile software team.* If it is not possible to solve the problem, the templates are fixed by the *MDD team* and the process is repeated starting with the code generation activity.
    - The problem can not be solved; consequently the automatic merge does not become possible again. In that case, the affected artifacts have to be merged in every iteration manually, if the corresponding part of the domain model is changed. The business analyst team keeps track of these artifacts and tries to avoid any changes to them. It is crucial to keep these special cases minimal and freeze the corresponding parts of the domain model as early as possible.

To ensure the success of the integration, integration tests and regression tests are also carried through. Critical bugs are corrected during the integration and the whole test activity is repeated.

- The *MDD development team* is busy with the *template development* and the *MDD environment development*. In the end of the iteration this team produces the generated artifacts from the domain model with the help of the latest MDD environment and templates. The enhancement of the MDD environment is usually orthogonal to the template development.

  During the iteration two different versions of the MDD environment are used. The latest version from the stable branch is available at the beginning of the iteration used for the template development and a separate development branch is used for the MDD environment development. As the iterations make progress, the MDD environment needs less and less modification; then the *template development*, in which new features are added to the templates, becomes the main activity. This will be explained in detail in Section "The Template Development Activity".

The Template Development Activity

The *template development* is the most complex and critical activity of the process. Typically for an iteration of the process several features are planned to be implemented iteratively in the frame of this activity. The implementation of a feature starts with its design, then the feature is coded and tested. During the implementation the generated development artifacts are extended.

Generated development artifacts are parts of the runnable application which is generated by the latest version of the templates, of the code generation tool, and of the domain test model. The implemented feature is extracted from the source code and inserted into the templates with the necessary modifications. The generated development artifacts are re-created by the modified templates. The result typically contains small errors, which can be detected when the result is built and tested. The test is carried through with the test cases of the original feature implementation. The templates are modified as long as no more errors occur during the test of the generated development artifacts. Then the feature is considered complete and the team moves on to implement the next one.

If a problem is recognized with the code generation tool, then it can be corrected in the frame of the MDD environment development. The tool used in the template development can be updated if the fixes are ready or if a new version is released during the MDD environment development.

The first iteration differs slightly from the other iterations, because at that point of time there are no prior generated development artifacts available. Before the MDD team starts to implement the first feature, it has to extract the utility artifacts, which is necessary to build and run the generated development artifacts. These artifacts can be kept separate or they can be shared with the agile development team. In both cases these artifacts have to be stabilized and frozen as early as possible; if change is necessary, the update must carried through in a planed and controlled manner.

3.4 The Technology Aspect

In this section we show a particular XML-based code generation tool that we designed and implemented to realize the development process explained in Section 3.3. First, we

describe how the components of the XML technology fit to general MDD. Then we present the architecture of our tool. Dedicated model transformation tools are continuously improving and their user base increase continuously, too. The validity of the optimal tool selection claims may need revaluation by the reader in case of creating similar code generation architecture.

3.4.1 XML and MDD

XML technologies can be used efficiently to implement one's own code generation tool. In this subsection we go through the elements of MDD as mentioned in Section 3.1 and explain which XML technology is appropriate to implement each element.

The domain model is usually described by graphs. The graphs are represented as trees with pointers, which allows to store the graph in XML format (pointers are value nodes which reference to ancestor nodes having the same values). OMG's MDA uses this approach for its XML based interchange format named XMI [13]. Moreover, a tree based representation of the domain model and its transformation fits better to problems with small and medium complexity [73].

The syntax of the domain meta-model can be expressed in the XML Schema language [21]. XML Schema defines a grammar that expresses which XML structures are valid. However, XML Schema is not expressive enough to formulate all properties, e.g. it can not say "If an N type node is referenced by an attribute A of an M type node, then the M type node must have an attribute B whose value is V". This type of properties can be checked with the help of Schematron [12] or of XSLT (see below). More details on the expressive power of XML Schema and related notations can be found in [97].

XSLT is the most well known standardized transformation language for XML. It is capable to express all kinds of transformations [71] and is designed for the efficient description of templates [10]. The main features that make it efficient for this kind of task are: basically it is a declarative (functional) language, but it can also be used in *rule-matching* style (constructs `template` and `apply-templates`) and *imperative* style (constructs `call-template`, `for-each`, and `if`); it is capable to output the results into different files; it supports output in both plain text and XML format.

XML technologies are a good choice, because they are standardized and well supported by the industry. Consequently, it is easy to find trained developers and one can expect long-term support. Furthermore, there are various kinds of tools available, e.g.: editors for XML, XML Schema, and XSLT and engines and debuggers for XSLT. Most of these tools have been available for almost a decade, so they can be considered mature.

XSLT was already used successfully by others for model transformation or for code generation [115]. A detailed comparison of XSLT with 3GL programming languages can be found in [105]. Borland's *Together 2007 DSL Toolkit* uses a special extension of XSLT to generate source code from models [2].

In 2006 we have evaluated several alternative tools, before decided to build our XML-based solution. The two most promising ones were AndroMDA 3.2 [1] and Iron Speed Designer [6]. AndroMDA is an open source MDA tool supporting the Java platform. This tool uses the XMI output of an UML editor and its code generator is written in Java. Although the tool initially seemed promising, we estimated that it would need too much effort re-target it to our domain model and .NET platform. On the other hand, Iron Speed Designer is a commercial tool which first generates an ASP.NET web application from an existing database schema and then allows a GUI based customization of the generated application. Also, this tool did not fit to our process, because we required an editable domain model from which the application could be generated. The architecture

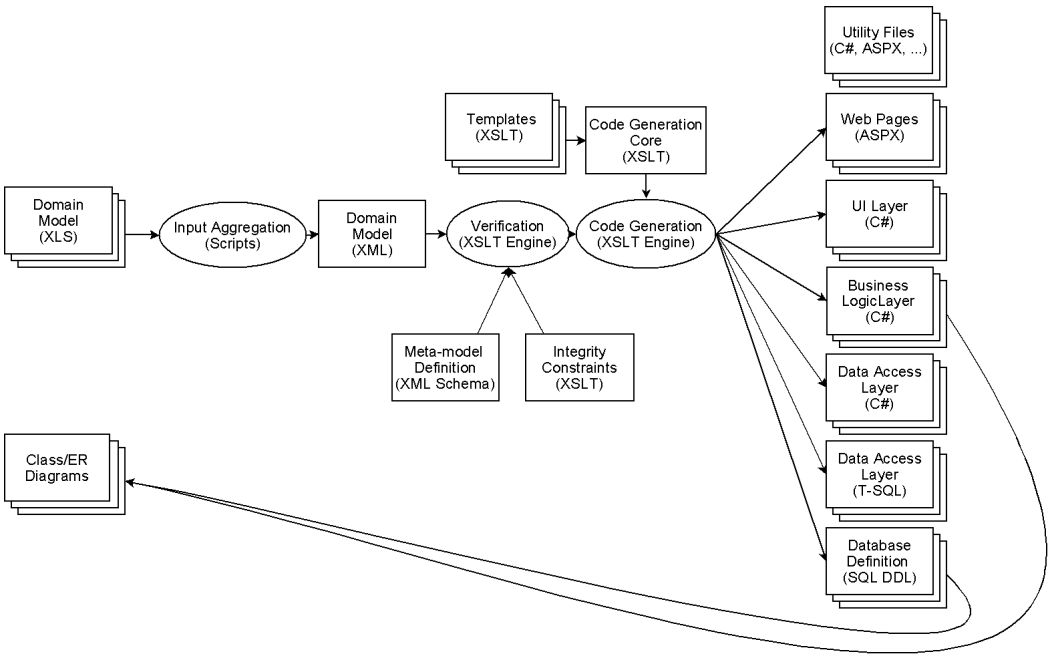


Figure 3.3: The Architecture of the Code Generation Tool

of the generated application did not meet our requirements either.

3.4.2 The Code Generation Tool

Figure 3.3 shows the architecture of our code generation tool. The ellipses indicate three processing phases and the rectangles represent the artifacts which are processed or produced. The purpose of the code generation tool is to generate a runnable ASP.NET 2.0 web application from a given domain model. Our code generation tool is specific for our domain model, but the architecture can be reused for enterprise applications which need user interface close to the database structure with simple business logic. The artifacts that are used to specify the domain model can be seen on the left side (the diagrams used to support the communication of the domain model are also indicated there). These artifacts, which serve as an input for the tool, were created and refined by the *business analyst team*. The generated artifacts are represented on the right side. These artifacts are the result of the *code generation* activity and are handed to the *agile development team*. The templates can be developed separately from the other parts of the tool.

The whole system is implemented with the help of approximately 5000 lines of XSLT templates and some hundred lines of shell scripts. The components of the tool must be as simple as possible and easy to learn, because the development process assumes short iterations (between two weeks and two months).

The Input

The domain model is represented in the form of Excel spreadsheets. These are used for communication between the customer and the *business analyst team*. Excel spreadsheets are a straight forward way to represent the domain model, because all stake-holders feel comfortable to work with them (according to our experience managers find editing spreadsheets more simpler than editing diagrams).

The Class/ER diagrams are only used to provide visualization. They augment the understanding of the big picture, but even customers trained to work with diagrams show

resistance to contribute to them (this behavior may have at least two explanations: they do not feel comfortable with the diagramming tool or they do not fully understand the details). The initial format of the Excel spreadsheets is created during the *domain model extraction*; it can be refined during the *domain modeling*.

Input Aggregation

Input aggregation is the first processing phase of the the code generation tool. The Excel spreadsheets are converted to XML files, then are aggregated into a single XML file with the help of scripts. The two representations of the domain model are semantically equivalent. The conversion step also adds additional flexibility: the representation format of the Excel spreadsheet can be changed without affecting the other phases of the code generation tool. The part of the tool responsible for the input aggregation is fine-tuned during the *MDD environment development* activity.

Verification

Verification is the second processing phase. It consists of two steps: an XML Schema validation and a constraint check:

- In the first step, the XML Schema validation, it is ensured that the XML representation of the domain model conforms to the XML Schema of its domain meta model. This serves as an extra safety check of the aggregated XML file.
- The second step, the constraint check, is the more important part of the verification. During this step several global constraints are checked (currently 24). The checks are implemented by some hundred lines of XSLT templates (currently 156). While it is possible to check the properties ensured during the first step by Excel macros, these global constraints can not, because this check needs information stored in separated Excel spreadsheets.

The required infrastructure for both steps can be quickly elaborated by the *MDD team* during the *MDD environment setup* and refined in the iterations during the *MDD environment development*. The code generation tool can be used by the *business analyst team* to verify the *domain model* by running only the first two processing phases. This usage is typical during the domain modeling.

Code Generation

The final processing phase is the code generation. The code generation tool and the templates are implemented in pure XSLT. The *code generator core* is a single XSLT file which is implemented in *rule-matching style* and calls the *templates* successively. The first template denormalizes the domain model before the templates are executed. The denormalization makes the templates simpler, because it eliminates a lot of the complicated referencing and repetitive computation of the values. The size of the XSLT templates varies between approximately 100 and 1500 lines. The bigger templates are already at the edge of maintainability.

The advantage of using XSLT to represent verification constraints, code generator core and templates is that the developers can be quickly reallocated and they can also understand each others' work with small extra effort. There were only two minor annoyances with the XSLT templates: the generation of ASPX and the formating of output.

- Microsoft's ASP.NET technology, called Active Server Pages, defines an extension for HTML to create web forms (ASPX). These files are neither proper XML nor HTML formats; thus, the output mode for them have to be plain text, which implies that all special characters have to be escaped.

- The problem of the output formating is more typical: to have the correct indentation and human readable formating, extra effort is needed. This problem is common with other template notations as well.

The Output

The result of the code generation is the source code of a web application. It is runnable with the necessary utility files, and can be tested immediately. The architecture of the generated application follows the current architecture guidelines [11, 44, 48, 102].

The *MDD team* has the task to keep the balance between the simplicity and size of the generated code. *Simplicity* means that the generated code must remain easy to understand and to extend. *Size* means that unnecessary duplication does not occur. Certain architectural guidelines imply a lot of generated code: stored procedures, data access objects, etc. On one hand, replications are not harmful if they are applied in a controlled fashion, because they can make the code faster, more readable, and easier to customize. On the other hand, they make certain type of changes much harder. Certainly, in a lot of cases it is a question of taste where and to which extent to apply replication.

The code generation approach helps also to eliminate certain types of errors, e.g. broken string references. The references are represented as strings that denote the stored procedures on the SQL-Server or information in the configuration files. The integrity of these string references can not be checked by the compilers.

3.5 The Practical Aspect

In this section we describe how the process and the technology explained in the previous sections have been applied in practice. We also describe minor details necessary to understand for carrying through a successful project. Both the process and the technology were successfully used in several projects. Here we describe that project in which they were applied for the first time. The project was carried through in the end of 2006 at the site of a customer who ships applications to its end customers.

3.5.1 Description of the Project

The aim of the project was to deliver an enterprise web application for a specific domain. The deliverable application had to build on top of Microsoft's ASP.NET 2.0 technology and an MS-SQL 2000 back-end. It also had to conform to architectural guidelines for the multi-layer enterprise ASP.NET applications at that time. Readable source code and complete developer documentation was also a must, because the end customer intended to extend and maintain the delivered application in house. The secondary goal was to introduce and evaluate the process and technology concepts which we have developed. The evaluation was done according to the criteria explained in Section 3.1.

At the beginning of the described project a detailed preliminary database plan and a detailed software requirement specification were available. These described approximately 100 database tables and over 300 screen masks. In these early documents it was

already easy to recognize repetitive functions and recurring simple Create, Read, Update, Delete (CRUD) dialogs. Both of the mentioned documents were accepted by the end customer, but they were too detailed and did not reflected its real expectation. Additionally, the end customer also had a detailed intranet application style guide. ASP.NET 1.1 software prototypes were available from our customer to demonstrate certain features of the software it planned to build. Also a technology demonstration prototype of the code generation tool was available which generated HTML files out of a small domain model which was similar to the target domain.

As a part of our work to develop the "Process and Technology Concepts" guide we reviewed the available literatures and tools. Our customer already knew both the domain and the end customer well. This was an important factor, because, if one is not confident with the the domain and the requirements of the end customer, a longer initial phase is needed.

Initial Phase

The initial phase was started with the domain model extraction and took approximately one and a half months. In the first two weeks we fixed the initial version of the domain model. After that, we started to create the domain meta-model and its XML Schema syntax. In parallel with that we started to set up the code generation tool. Our aim was that the initial version of this tool would include some simple templates which generate from the XML files an ASP.NET 2.0 application containing empty screen masks with navigation elements.

The first problem we faced was that the editing of the XML files was more challenging for the end customer than we thought. Initially, we planed to use XML files as a temporary solution, until we would build our own domain language and corresponding diagramming tool in a later iteration, but the time was not sufficient for this. Thus, we adapted Excel spreadsheets as a solution.

By the end of the initial phase we had the first version of the domain model represented in Excel spreadsheets, an informal description of the domain meta-model, a working version of the code generation tool and a part of the domain model as a domain test model. Our customer successfully used the first generated code to demonstrate to the end customer the navigation structure of the application.

First Iteration

The main goal of the first iteration was to generate in two weeks simple artifacts for all layers of the application. During that time also the initial architecture was fixed according to the architecture prototype developed by the agile development team. By the end of the iteration, the templates were capable to generate the following artifacts:

- the DDL scripts to create the database;
- simple stored procedures in T-SQL with SELECT statements;
- the methods in C# of the data access layer to call the stored procedures;
- the skeleton of the business objects in C#;
- and the ASPX forms to display data from the database.

The different generating phases in the prototype of the code generation tool had to be run manually. That issue was solved with the help of simple shell scripts.

At that point the internal team structure allocation stabilized. Our method completely fitted to the situation: the concept of business analyst team was not only a theoretical concept, but there was a real team who regularly met with the end customer. That team collected the requirements and communicated them to the developers. The division of the development work between the agile development team and the MDD team was also proved to be efficient. In summary, we did not have any serious issue at the first iteration.

Second Iteration

This iteration was planned to take one month; its aim was to complete the functionality of the data access layer together with the necessary stored procedures and the business objects. The other enhancement was that a special domain test model was developed. This was necessary because the original domain test model, which was a part of the real domain model, did not cover a lot of possible cases in the domain meta-model and was already too huge to get generated and compiled quickly.

Third Iteration

During this iteration, user interface features were implemented and enhanced. Extra functionality was also added to the data access layer according to the request of the agile development team. The iteration was completed in two weeks. Here we had the first serious integration problem: the iteration resulted in a lot of files that needed to be merged and fixed manually. This issue emerged from the fact that the teams did not take it serious enough to keep track of the artifacts modified after the generation. This issue was solved by moving features to separate methods in C# partial classes, rather than code the features directly in the generated artifacts. The files in the version tracking system were also cleaned up.

Further Iterations

The fourth iteration was a one week long bug fix iteration. An error was found in the framework; this error was triggered by certain combination of the properties in the domain model. The error was not reproducible with the domain test model. We closed this issue by developing a workaround and extending the integration test. The fifth iteration contained the last set of the features and the integration was gone smoothly.

Maintenance

During the maintenance most of the modification was carried out by the agile development team. If the domain was modified, new artifacts were generated. The integration of these new artifacts was carried through in a piece by piece manner: instead of collecting the generated artifacts and the hand crafted artifacts together and integrating them, the MDD team delivered just the requested new or modified files to the agile development team, which merged these artifacts manually. This ensured that the modification had as little effect on the other parts of the software as possible.

Metrics

Our project was considered by our customer successful in comparison with its previous agile projects. However, it is not clear how to evaluate the project objectively. It may be judged according to basic project metrics, but there are no such data available from similar

projects. In the final release the total amount of source code developed was approximately 300 thousand lines of code (KLOC). This contained 200 KLOC of generated code, which was produced from 900 lines of XML specification. The project team had 8 members and the net effort to complete the project was around 25 man-months.

3.5.2 Best practices

In this section we describe the best practices collected during our projects. Although they are simple, they can save a lot of unnecessary effort.

GUI Prototyping

During the initial setup of the code generation tool, it is an important question what kind of artifacts should be generated first. The generation of a rapid GUI prototype is a reasonable choice. This task is a realistic aim and during its implementation the team can face with the typical implementation challenges. It also results in valuable artifacts for the project stake-holders: a large scale GUI prototype is capable to demonstrate that the planned navigational or menu structure is not only effective with a small demonstration example, but can be used in the final application.

Skip Integration

Integration is an important activity of the described MDD process. During this activity the generated artifacts and the hand crafted software are merged. Even if it proceeds smoothly, it takes time. If the changes do not affect the interfaces and the customer can be satisfied by demonstrating the improvement of the deliverable system separately, the integration can be safely skipped. If the interfaces are changed, then it is strongly recommended to do the integration. The more integration problems are postponed, the more problems have to be fixed later, which can be much more costly.

Minimize Manual Merge

By the integration, two different versions of the artifacts may emerge: one version of the artifact is generated during the previous integration and is modified by the agile development team, the other version of the artifact is newly generated by the MDD team from the new templates and domain model. These versions have to be merged manually by human interaction. If too many artifacts need manual merge during the integration, it must be considered as a serious warning. The more manual effort the integration needs, the more inefficient the MDD approach is. If one does not start to solve this issue early, it can quickly become the bottleneck of the process. This issue can be resolved by the following two options:

- split either the generated artifacts and handcrafted software into separate files (e.g.: use partial classes in C#);
- or generate artifacts at the beginning and then, if the artifacts are customized, freeze the specification and those templates which can change the result of the generation.

It is important to document these decisions in the MDD vision document and evaluate which parts of the technological environment and the domain are more likely to change. This documentation also helps the business analysts to understand how much effort a different type of modification costs.

Variable Naming

The importance of selecting good variable names is well known by the software development community. There are a lot of guidance and coding standards to help developers in this task. In the case of handcrafted software the variable names chosen by the developers, in the case of generated artifacts it is derived from the specification. Hence, business analysts have to be educated about the importance of name selection. The developers of the templates also have to pay extra attention to the variable names, even if it does not affect their work directly. They should avoid adding long or senseless pre- or postfixes or loop counters to the identifiers.

Testing Issues

Testing is an important part of the current software quality assurance practice. In our process we prescribe several point where testing should be done. If the domain test model is carefully chosen, a high quality of the generated artifacts can be ensured by testing a relatively small amount of code. The quality of the domain test model can be expressed by the coverage of the templates and the coverage of the domain meta-model. This kind of test does not rule out the traditional integration, regression and stress tests.

3.6 Summary

In this chapter we described a development process which combines artifacts of traditional software engineering and model driven development. It also supports continuous integration in short iterations which allows to be used by small and medium development teams. We also described a tool-chain that was implemented and applied in a project which executed with the help of our process. Finally, we documented our project experiences and proposed some best practices.

We noticed that the refinement steps of the template development consist of simple repetitive activities. Next, we investigate whether these activities can be automated to reduce the effort needed to develop the templates.

Chapter 4

Finite State Transducer Modification by Examples

In this chapter we investigate whether an effective inference procedure can be developed to derive a modified transducer from examples of desired input/output pairs. We propose two algorithms to infer modifications of the transducers and we discuss their possible extensions. The structure of this chapter is described at the end of the introductory section.

4.1 Introduction

In model driven software development the creation of transformation is not well supported. Model Transformation By Example (MTBE) is a new approach to automatically create transformations according to examples [18, 122]. As we have seen in the previous chapter 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.

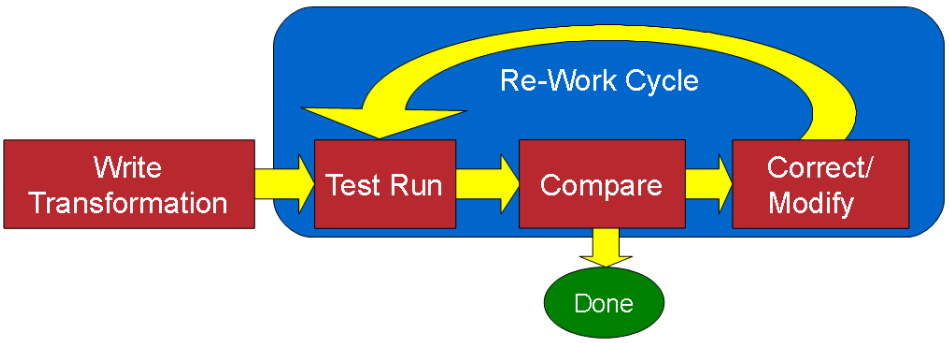


Figure 4.1: MDD Process

The template development of a traditional model transformation process can be seen in Figure 4.1. The development of the template (a program describes the transformation) is usually driven by examples. Examples are pairs of source models and expected results. The template is crafted from expected result by inserting template statements into the output document e.g. generalizing the example to get values from the source model or make the emitting of certain parts dependent from the source model. After the creation of the initial template, it is executed on the source model of the examples. We compare

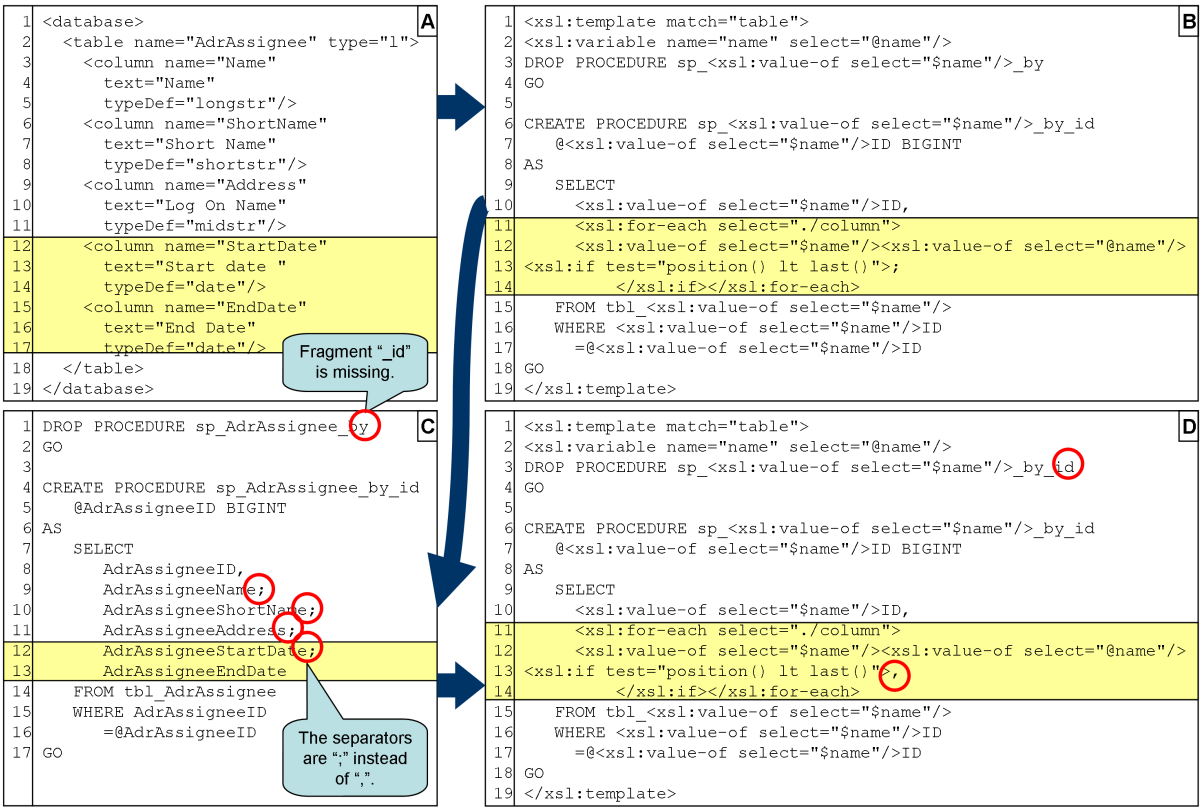


Figure 4.2: Example XSLT correction

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 template and repeat the process from the test run. During the creation of the templates developers often introduce minor errors e.g. forget 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. Locating the part of the transformation which contains the corresponding erroneous output string and correcting the errors automatically may save the effort of several iterations of the re-work cycle. In case of smaller refactorings or extension of the outputted source code, the structure of the templates remains the same. In such cases the text generated by the transformation can be updated automatically too.

To illustrate the basic idea we give (Figure 6.2) an XML fragment of a database table description (A) as an example 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[66, 65].

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 that some minor errors exist 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) automatically.

Our goal is to develop specialized algorithm to infer M2T transformation modifications. The existing MTBE approaches can infer only simple transformations. The main difference of our approach from the existing MTBE approaches is that we require ex-

tra information in form of an existing approximate solution. This enable us to work with complex transformations and update simple modifications which does not require human insight. M2T transformations are modeled with simple finite-state transducers over strings. This simple transformation model helps us to develop algorithms, metrics and evaluate algorithms. We present algorithms to infer transducer modifications from input/expected output pairs.

The main contribution of the current chapter are the following:

- we reformulate the MTBE approach to M2T Transformation Modification by Example
- we model the M2T transformation with finite-state transducers over strings;
- we develop algorithms to infer transducer modification by example;
- we describe our preliminary evaluation and possibility of their improvements.

The remainder of this chapter is structured as follows: In Section 4.2 we provide a formal description of the problem. In Section 4.3 we give a description of the first algorithms: the description starts with the definition of the concepts which are needed by both algorithm; then we give a demonstrative example and according to this example we describe each step in detail. In Section 4.4 we describe the second algorithm in a similar way. Finally, in Section 4.5 we describe the possible extensions of the algorithms.

4.2 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 automatically the approximate transformation t to the expected transformation t' to generate the modified code g' from the original specification, i.e. $g' = t'(s)$. In order to get a non-trivial solution, additional requirements (optimality of metrics of the generalization properties of the algorithm) need to be introduced on t'. With fixed g and no constraints on the t' the task is trivial solutions are exists, e.g. the algorithm may just learn to give the right answer to the specified example.

Therefore we reformulate this problem as depicted in Figure 4.3. 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" requirement 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 multiple results or no results at all, because the examples may be contradicting or ambiguous.

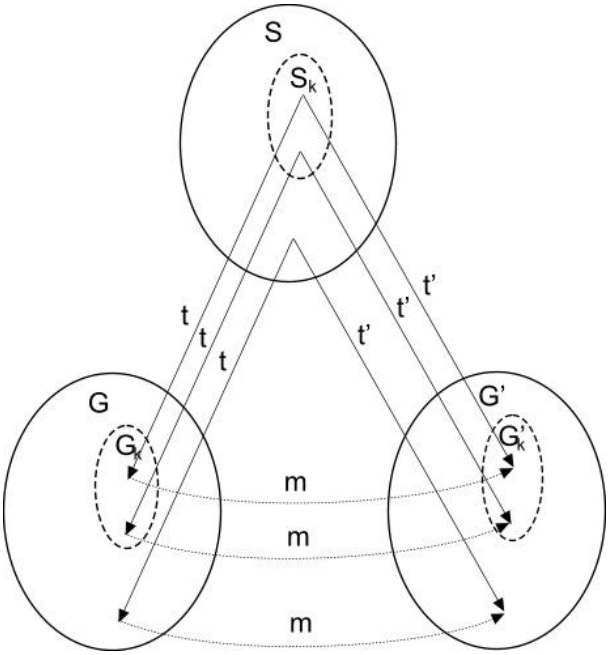


Figure 4.3: Transformation Modification Problem

4.3 Inferring Modification of the Transducers

In this section we describe the algorithm which we have developed to infer finite state string transducer modifications. The algorithm infers a modified transducer from a set of example pairs and the original transducer. An example pair contains an input string and the corresponding output string which is the expected result of the execution of the modified transducer. (The input which is expected by a M2T transformation is diffrent from the expected input of a transducers, and they both produce string as output.)

Technically, the only correctness requirement is that

- the algorithm must produce the expected output by executing on the example input and
- the behavior with 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 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 to be minimal; or we may create the expected modified transducers and measure the properties of the input by which the algorithm produces the expected modification of the transducer (by forming a transducer for each example pair and create a union of them).

In Section 4.3.1 we define the concepts used to describe the algorithm. In Section 4.3.2 we describe the algorithm and explain the intuition behind it. The algorithm description

starts with a brief explanation of the algorithm, which is followed by the detailed explanation through an example. In Section 4.3.3 we discuss the properties of the algorithm informally and outline the possible ways of its improvements.

4.3.1 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 a *Transducer* and a set of input and output pairs as parameter and returns a transducer.

  InferenceAlgorithm := (Transducer, Input*, Output*)) $\rightarrow$ 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 start with #. The notation has three kinds of data structures: record, sequence (list) and map (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 the name of the field prefixed by a dot.
- Lists are used to enumerate ordered values with equal type. They can be constructed by enumerating their elements. The signature of the access functions are as follows:
  - list(_): name of T $\rightarrow$ T^* (empty list)
  - head(_): $T^* \rightarrow$ T (first element of the list)
  - tail(_): $T^* \rightarrow T^*$ (a list contains all but first element)
  - last(_): $T^* \rightarrow$ T (last element of the list)
  - allButLast(_): $T^* \rightarrow T^*$ (a list contains all but last element)
  - length(_): $T^* \rightarrow \mathbb{N}$ (the length)
  - _ + _ : (T^*, T^*) $\rightarrow T^*$ (concatenation of lists)
  - [_]: T $\rightarrow T^*$ (one element list)
- Maps are used to represent mappings (functions).
  - map(_ , _): (name of T_1, name of T_2) $\rightarrow$ ($T_1 \rightarrow T_2$) (empty function)
  - _ [_]= _ : ($T_1 \rightarrow T_2$, T_1, T_2) $\rightarrow$ ($T_1 \rightarrow T_2$) (define function value for a key)
  - _ [_]: (($T_1 \rightarrow T_2$, T_1) $\rightarrow T_2$ (the value belonging to a key)
  - keys(_): ($T_1 \rightarrow T_2$) $\rightarrow T_1^*$ (sequence of keys on which the function is interpreted)

4.3.2 An Inference Algorithm Based on Direct Trace Modification

In this section we describe our first inference algorithm which we call *Direct Trace Modification Based Inference (DTMBI)*. 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 to the expected output. According to the result of the comparison, the algorithm modifies the trace. 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 be 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.

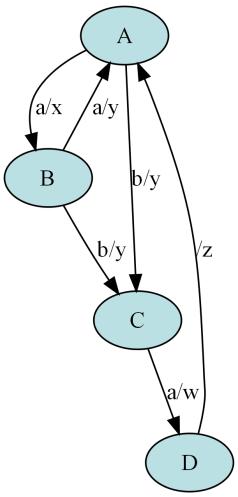


Figure 4.4: An Example Transducer

```
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 4.1: The Main Algorithm

Example An example transducer is depicted on Figure 4.4. 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 (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 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 4.1. 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 an input on the transducer, which produces the trace *tr* (line 3). The trace is a sequence of trace steps corresponding to executed transition rules. A trace step records the source state, the processed input symbol, the target state and the 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 [101] (line 5). Then the modified trace *tr'* is calculated from the original trace *tr* and the difference sequence *diff* (line 6). Finally, the new transducer *tr'* is built from the modified trace (line 7).

The function *execute(trans, input)* initializes its internal state variable according to

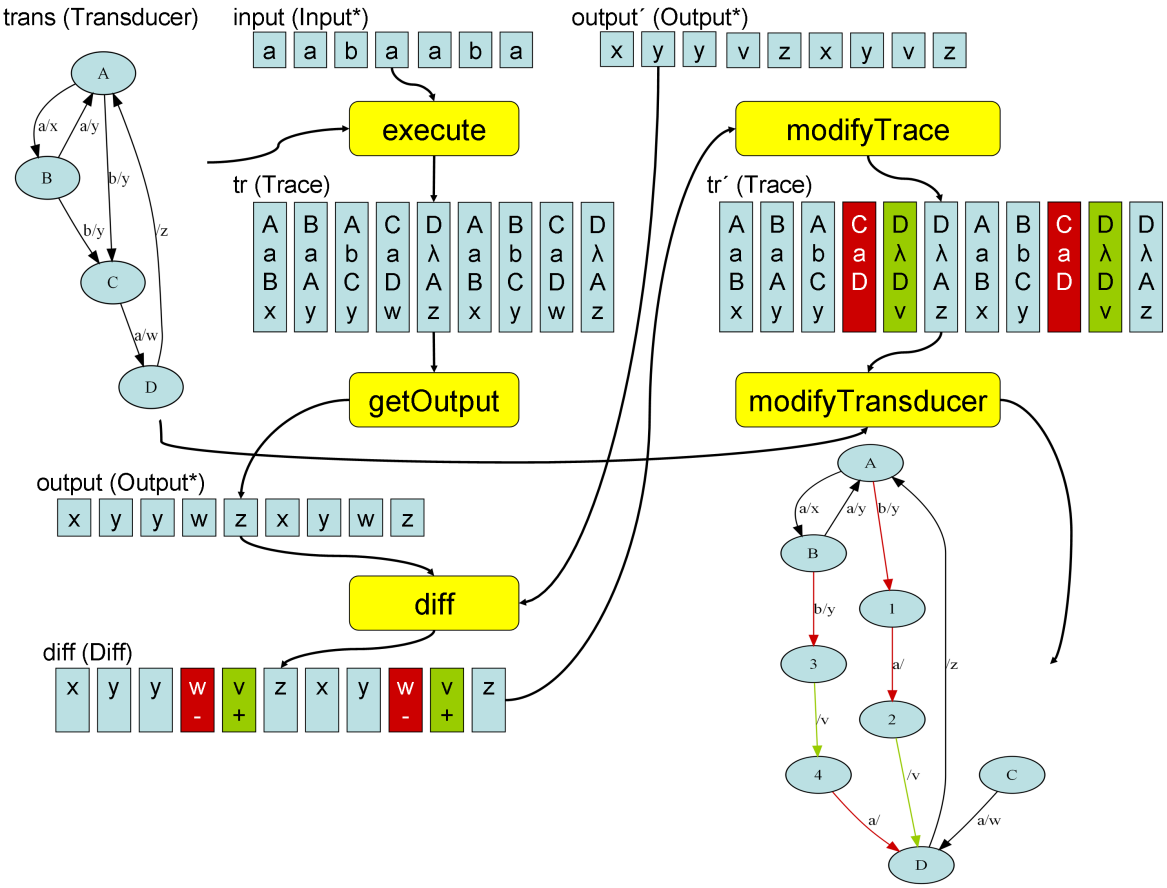
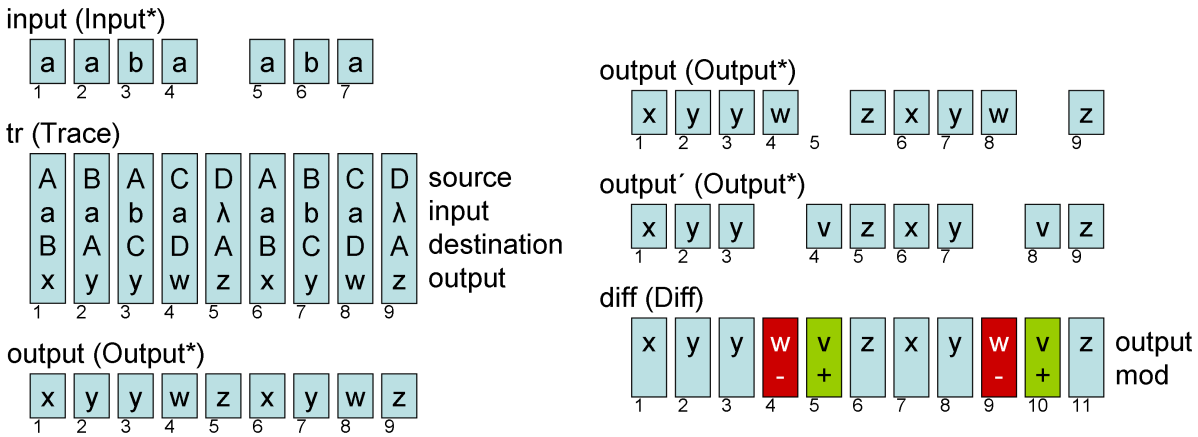


Figure 4.5: The Example of the Main Algorithm

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*.

Example: By considering the example described in the beginning of Section 4.3 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; 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 4.11. 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 4.6a. The result of the *diff* and the *modifiyTrace* functions can be seen in Figure 4.6b and Figure 4.7, respectively. The actual result of the algorithm is the rightmost graph of the figure.

The modifyTrace Function We can see the *modifyTrace* function in Listing 4.2. 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:



(a) The Trace

(b) The Difference Sequence

Figure 4.6: Processing Steps of the Example

```
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 4.2: The modifyTrace Function

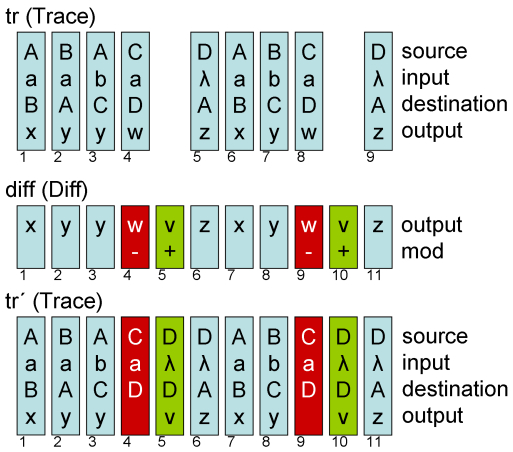


Figure 4.7: Example of the modifyTrace Function,

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 of 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.

Example: In Figure 4.7 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.

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 introduces 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 it expects the same input and its source states are the same; or it reads nothing and its source states the same.
- 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 introduced state. This function is going to be explained later after the description of the main algorithm.

The modifyTransducer Function The function *modifyTransducer* is shown in Listing 4.3. 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.

```
1  modifyTransducer(Transducer trans, Trace trace): Transducer
2    if length(trace)>0:
3      UsageCounter u=map(TransitionKey, Nat)
4      for guard in keys(trans.rules):
5        u[guard]=0
6      Transducer trans'=trans
7      UsageCounter u'=u
8      Trace trace'=list(TraceStep)
9      for trStep in trace:
10       (trans', u', trace')
11           =modifyTransducerImpl(trStep, trace', trans', u')
12     return trans'
13   return trans
14
15 modifyTransducerImpl(TraceStep trStep, Trace trace,
16     Transducer trans, UsageCounter u): (State, Transition,
17     Trace, UsageCounter)
18   TransitionKey guard
19       =TransitionKey(trStep.source, trStep.input)
20   if isSimilarKey(trans.rules, guard):
21     if guard in trans.rules and trans.rules[guard]
22         == TransitionData(trStep.target, trStep.output):
23       u[guard]=u[guard]+1
24       return (trans, u,  trace+[trStep])
25     else:
26       State newS=getDistinctState(trans.rules)
27       TransitionKey newGuard
28           TransitionKey(newS, guard.input)
29       trans.rules[newGuard]
30           =TransitionData(trStep.target, trStep.output)
31       u[newGuard]=1
32       (newS',trans',tracePrev',u')=attachNewState(newS,
33           trans, trace, u)
34       TraceStep trStep'=TraceStep(newS, trStep.input,
35           trStep.target, trStep.output)
36       return (trans', u', trace+[trStep'])
37   else:
38     trans.rules[guard]
39         =TransitionData(trStep.target ,trStep.output)
40     u[guard]=1
41     return (trans,u, trace'+[trStep])
```

Listing 4.3: The modifyTransducer Function

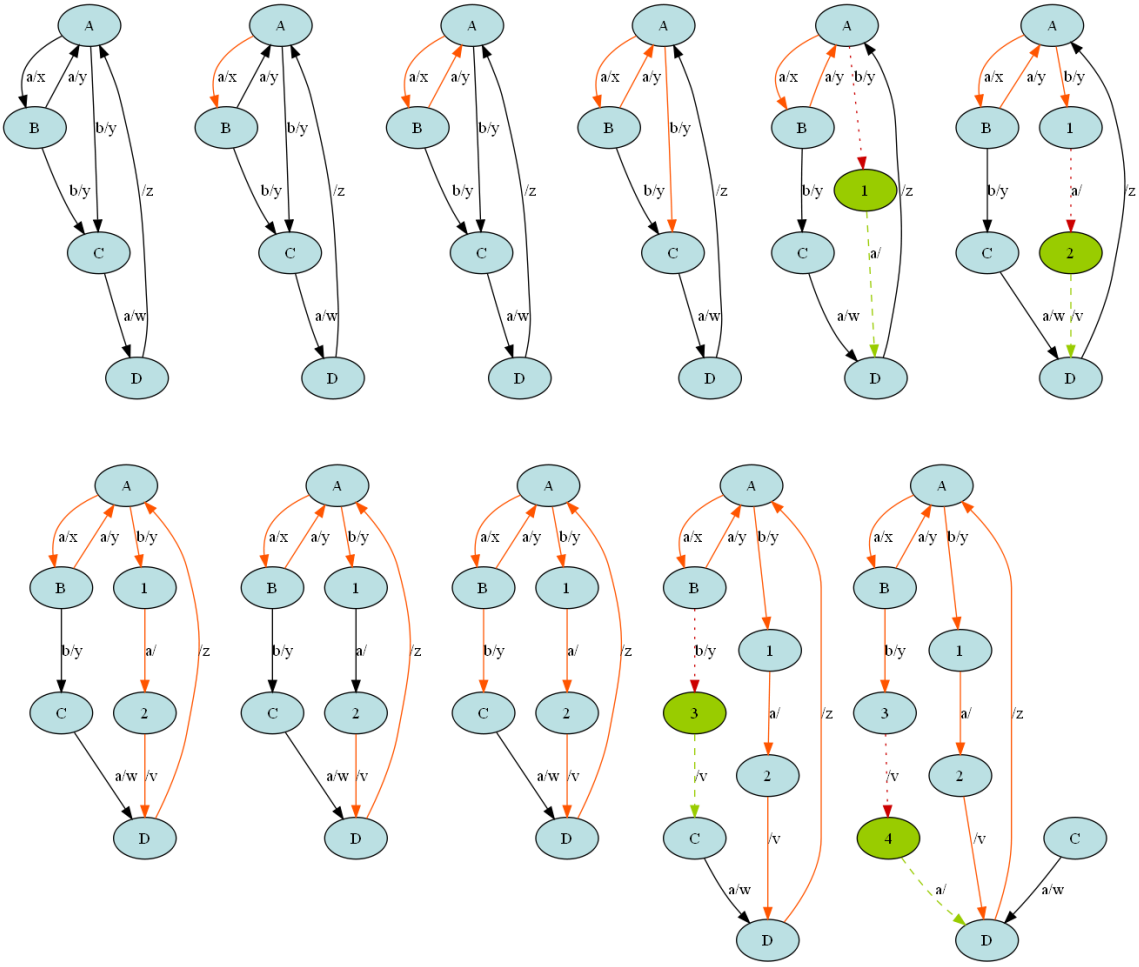


Figure 4.8: States of the Modifications of the Transducers

The wrapper function initializes the usage counter. This mapping assigns the number of rule applications to each rule (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 three cases:

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

Example: We can follow the modifications of the transducer in Figure 4.8 according to the *tr'* (Trace) which can be seen in Figure 4.7. Each graph is the state of the

```
1  attachNewState(State newS, Transducer trans, Trace* tr,
2      UsageCounter u): (State, Transition, Trace,
3      UsageCounter)
4    Transducer trans'=trans
5    if length(tr) > 0:
6      Trace trPrev=allButLast(tr)
7      TransitionKey guard
8          =TransitionKey(last(tr).source, last(tr).input)
9      if u[guard]==1:
10       trans'.rules[guard]
11           =TransitionData(newState, last(tr).output)
12       TraceStep newTrStep=TraceStep(last(tr).source,
13           last(tr).input, newS, last(tr).output)
14       Trace tr'=trPrev+[newTrStep]
15       return (newS, trans', tr', u)
16     else: #u[guard]>1
17       UsageCounter u'=u
18       u'[guard]=u'[guard]-1
19       TransitionKey newTransGuard=(newS+1, last(tr).input)
20       u'[newTransGuard]=1
21       trans'.rules[newTransGuard]=(newS, last(tr).output)
22       TraceStep newTrStep=TraceStep(newS+1, last(tr).input,
23           newS, last(tr).output)
24       (State, Transducer, Trace, UsageCounter) (newS',
25           trans'', trPrev', u'')=attachNewState(newS+1,
26           trans', trPrev,u')
27       return (newS', trans'', trPrev'+[newTrStep], u'')
28   else:
29     trans'.initState=newS
30   return (newS, trans', tr, u)
```

Listing 4.4: The attachNewState Function

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 in 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 fourth 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 one 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 4.4. The function modifies the transition rule referenced by the previous trace element to provide the transition to the newly introduced 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 target 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 and 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 until the algorithm finds a transition applied once or no more preceding trace step exists. In the later case, the algorithm modifies the initial state (line 26). In the worst case, the transducer contains as many states and elements as the length of the example.

Example: The example contains only situations which exercise the first branch. The modifications caused by this function are marked dotted gray (red) in Figure 4.8.

4.3.3 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 remarks in connection with the algorithms:

- *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* → Output**. More precisely to each prefix i of input I, belongs a prefix o' of the modified output O' which is the modified output of the transducer t produced from i.

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

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

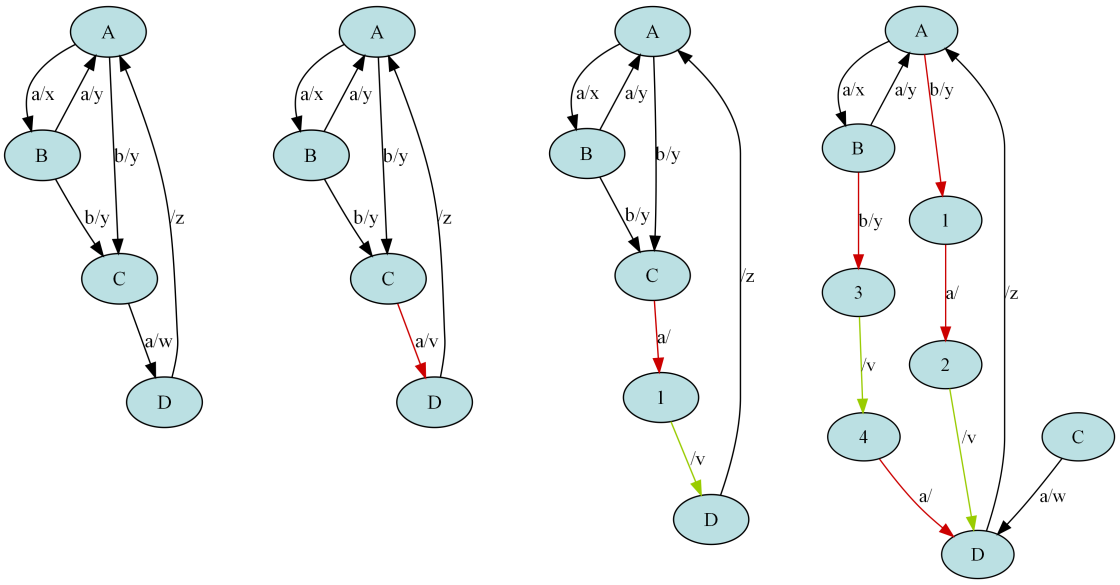
  By an example pair "aabaaba" and "xyyvzxyvz" the set of pairs are "a", "x"; "aa", "xy"; "aab", "xyy"; etc.

- *The algorithm modifies the transducer immediately to a locally optimal form* after each new trace step has been 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 4.9 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 4.9a, Figure 4.9b, Figure 4.9c and Figure 4.9d, respectively. The inferred solution produces the same result as the *Expected A* and *Expected B* for every



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

Figure 4.9: Different Modifications of the Transducers

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 fixed 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 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 Chapter 5.3.

4.4 An Inference Algorithm Based on Graph Annotation

In this section we describe an improved inference algorithm, which we call *Graph Annotation Based Inference (GABI)*. The algorithm starts by recording the trace of the transducer execution on a given input. The output is extracted from the trace and it is compared with the expected output. The algorithm builds a transition graph from the transition rules of the transducer. According to the result of the comparison, the algorithm annotates the transition graph. The annotations are assigned to the nodes containing the input edge, the new output and the output edge. The input and output edges are the preceding and following edges in the trace around the new output. The annotations assigned to the edges express that the edge must be deleted or must be changed to print an empty element. To make the transducer capable of reproducing the modified trace, new transition rules are added or existing transition rules are modified according to the annotations. During the processing of the annotation, the algorithm may ask for further information or may alert the user of contradictions.

Example We repeat here the same example which we described at the beginning of Section 4.3.2. An example transducer is depicted in Figure 4.10. 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 the expected output "xyyvzxyvz". The algorithm is expected to infer a new transducer from the specified transducer and from the example pair.

4.4.1 Overview of the Algorithm

Example: In Figure 4.11 we depict the steps of the algorithm. The rectangles with rounded corner (yellow) represent the functions. The objects between rectangles represent the data passed between the functions by considering the example described in the beginning of Section 4.4. In the top of the figure the input parameters can be seen: the graphical representation of the transducer *trans*, the *input* ("aabaaba") and the *output* ("xyyvzxyvz"). The algorithm computes the modified transducer *trans'* as the result.

1. The transducer (*trans*) executes the *input* to produce a trace (*tr*). From the trace the output and the graph representation (g) of the transducer can be computed. The computed output can be compared to the expected output, and the result of the comparison is represented in the difference sequence.

2. The modifications in the difference sequence are recorded as data elements of the nodes and as the edges of the graph by the annotateGraph function. The function also produces a list of the processed nodes and a list of the processed edges.

3. The functions inferNodeMod and the inferEdgeMod infer the solution from the annotated graph for each element of the list of the processed nodes and the list of processed edges, respectively.

4. The function inferEditScript creates an edit script (a list of modification steps) from the list of inferred solutions belonging to nodes and to edges. The transducer is finally modified by the function modifyTransducer resulting the transducer *trans'*.

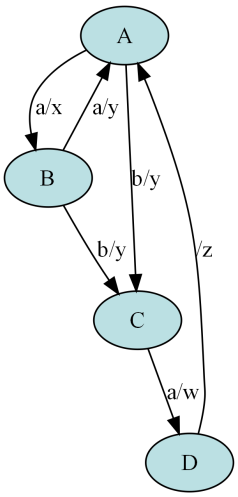


Figure 4.10: An Example Transducer

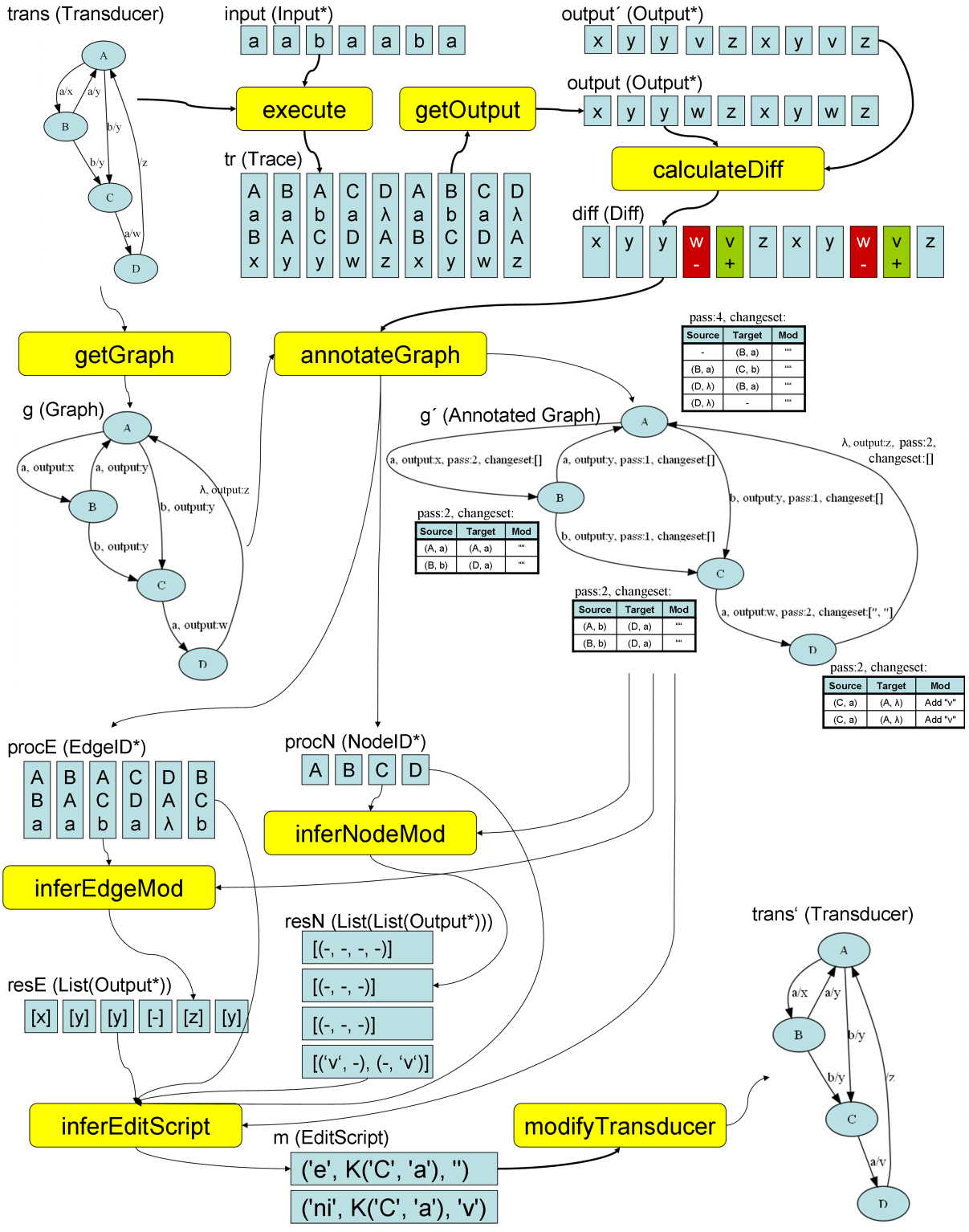


Figure 4.11: An Example of the Execution of the Algorithm

4.4.2 Extending the Definitions

For the new algorithm we slightly change the data structure definitions by allowing multiple output symbols in the transition rules. We also introduce data structures for intermediate representation of the modification in the graph. Except these changes listed above these definitions are used that can be found in Section 4.3.1. The four leading definitions are structurally the same as the ones in the earlier definitions, but these definitions allow multiple output symbols.

- 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 output symbols.

  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 *Trace Step* is a state transition record.

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

- A *Difference Sequence* is a sequence of records each of which consists of a string of output symbols and its relation to the source string and the 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: {" ", "-", "+"})*

- A *Node ID* is a State.

  NodeID := State

- A *Edge ID* is a tuples of two States and an Input symbol.

  EdgeID := (State, State, Input)

- A *List of the Processed Nodes* is a sequence of States.

  ProcNodes := NodeID*

- A *List of the Processed Edges* is a sequence of tuples of two States and an Input symbol.

  ProcEdges := EdgeID*

- An *Edge Label* is a label used to denote incoming or outgoing edges. In case of a multi graph we need the state and the input symbol to determine the edge uniquely.

  EdgeLabel := (State, Input)

- A *Vector Label* is a label used to denote incoming or outgoing edges. This is the same as the edge label extended with the direction information.

```
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    Graph g=getGraph(trans)
7    (g',procN, procE)=annotateGraph(tr, diff, g)
8    List(List(Output*)) resN=inferNodeMod(g', procN)
9    List(Output*) resE=inferEdgeMod(g', procE)
10   EditScript m=inferEditScript(g, resN, procN, resE, procE)
11   Transducer trans'=modifyTransducer(trans, m)
12   return trans'
```

Listing 4.5: The Main Algorithm

VectorLabel := (State, Input, {"in", "out"})

- A row of *Difference at a Node* is a record of the source edge label, target edge label and the modified output.

  NodeD := (source: EdgeLabel, target: EdgeLabel, output: Output*)

- A *Difference Table at a Node* is a sequence of rows of differences.

  NodeDT := NodeD*

- A *Difference Table at an Edge* is a sequence of rows of differences (modified outputs).

  EdgeDT := List(Output*)

- An *Edit Script* contains a sequence of modifications of a transducer.

  EditScript := (type: {"e", "ni", "no"}, nodeid: NodeID, edgeid: EdgeID, output: Output*)

- A *Data at a Node* is a record that corresponds to a node.

  NodeData := (pass: nat, diffTab: NodeDT)

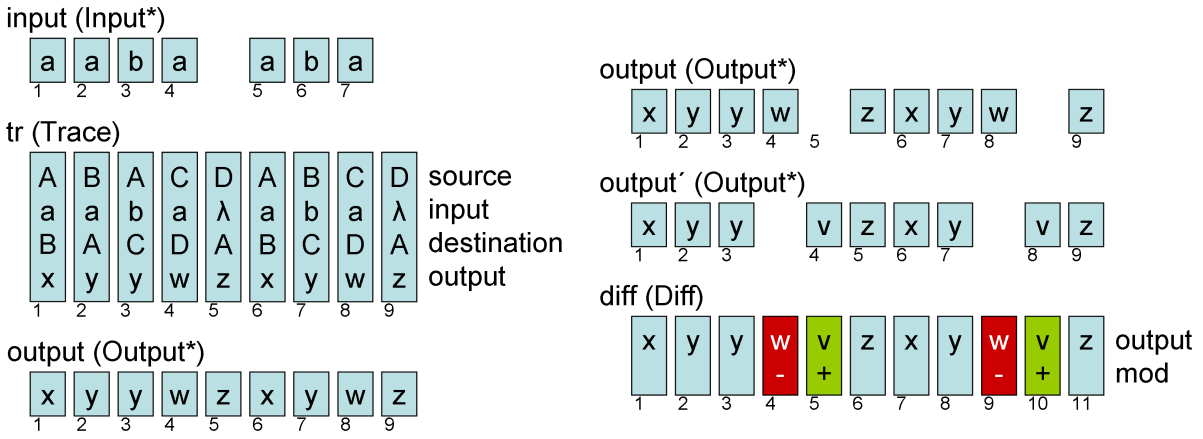
- A *Data at an Edge* is a record that corresponds to an edge.

  EdgeData := (output: Output*, pass: nat, diffTab: EdgeDT)

4.4.3 The Detailed Description of the Algorithm

The main function computes the modification of the transducer according to the supplied example by executing the part of the algorithm. It takes the following inputs: a transducer *trans*, an example input *input* and an expected output *output'* and returns the modified transducer *trans'*.

inferTransducer(Transducer trans, Input* input, Output* output'): Transducer



(a) The Trace

(b) The Difference Sequence

Figure 4.12: Processing Steps of the Example

The main function can be seen in Listing 4.5. The algorithm starts with the execution of the input on the transducer which produces the trace *tr* (line 3). The trace is a sequence of trace steps corresponding to the executed transition rules. A trace step records the source state, the processed input symbol, the target state and the produced output symbol of the corresponding transition. The output is the sequence of the output elements of the trace sequence (line 4). Then the sequence *diff* is calculated which describes the difference between the output and the expected output using a commonly used comparison algorithm described in [101] (line 5). Then the transducer *trans* is converted to a graph representation *g* (line 6). The annotated graph *g'* is calculated from the original trace *tr* and the difference sequence *diff* (line 7). The lists of the processed edges *procN* and nodes *procE* are created along with this calculation. The inference function infers the lists of modifications of the nodes *resN* and edges *resE* (line 8, 9). The edit script *m* is assembled from the lists of modifications and the list of processed edges and nodes (line 10). Finally, the new transducer *tr'* is built from the modified trace (line 11).

The function *execute(trans Transducer, input Input): Trace* initializes its internal state variable according to the *init* element of the *trans* parameter. Then the function executes the rules in the *trans* parameter. After each transition a new trace step is appended to the trace sequence. The *getOutput(tr Trace): Output* function returns the sequence of the output elements of the trace sequence. The *getGraph(trans Transducer): Graph* functions builds a directed multi graph from the transducer. The nodes of the graph are the states of the transducer. The edges are composed of the source and target states of the rules, the keys of the edges are the input symbols and the data elements contain the output symbols and the annotations.

Annotating the Transition Graph with the Modification This function converts the information from the difference sequence into the difference tables of the nodes and the edges with the help of the trace. The function takes the graph, the difference and the trace as input; and returns the annotated graph, the list of the annotated nodes and the list of the annotated edges.

annotateGraph(Graph graph, Trace trace, Diff diff): (Graph, NodeID*, EdgeID*)

The annotated graph is a copy of the original graph containing difference tables at the nodes and the edges. The function marks if a certain node or edge is represented in the

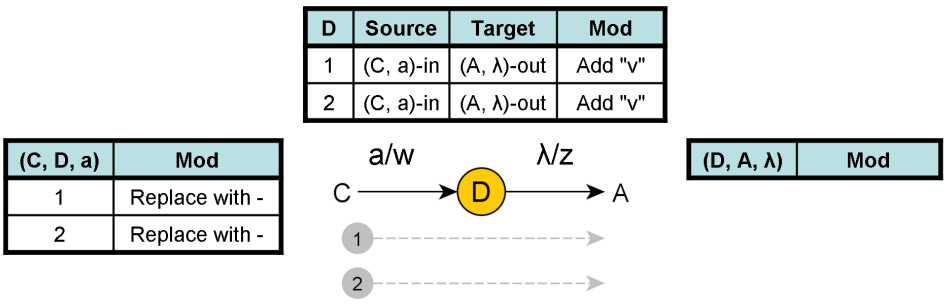


Figure 4.13: A Part of the Annotated Graph with Traces and Difference Tables

trace. If it is represented, then we also mark whether it is left unmodified, or new tokens are inserted, or existing ones are removed. The transition which produced a specific list of the output symbols can be identified by simultaneously processing the diff sequence and the trace itself. As soon as the transition which produced (or removed) the output symbols is known, the position of the output symbols can be located in the annotated graph.

Example: In Figure 4.13 the difference tables for Node D are shown, which are generated from the trace of Figure 4.12a. The node and the edges are labeled in the same way as it can be seen in Figure 4.10; the edges are additionally marked with the tuples of the source state, target state and input symbol. The trace elements which traverse Node D are represented by (numbered) gray arrows. The connection between the figures can be understood as follows (each line starts with the number of the trace in Figure 4.13, followed by the corresponding element numbers in Figure 4.12b, and is closed by the informal description of the trace elements):

- 1 (elements 4-6): Node D is passed for the first time, when a "v" is printed (C) and then a "z" is printed (A);
- 2 (tokens 9-11): Node D is passed for the first time, when a "v" is printed (C) and then a "z" is printed (A);

The insertion is stored in the difference table of the Node D and the deletion is stored in the difference table of Edge (C, D, a) as a replacement of the original output symbols.

Details: More precisley, the tables in the nodes and in the edges of the trace are generated as follows. Each row in the tables represents one continuous sequence of difference elements corresponding to the trace passing the edge or the node. In case the tables are located at the nodes, the source edge and the target edge are recorded besides the elements of the difference sequence. The difference tables are constructed by the function *annotateGraph*. This function calls two sub-functions: the function *procPrefix* which processes possible insertion at the beginning of the difference sequence (line 3); and the function *annotateGraphImpl* which actually does the processing of the trace steps in a recursive manner (line 4-5).

The function *procPrefix* searches additions at the beginning of the difference sequence (lines 6-8). Then it modifies the difference table at the node according to the result of the search (line 9-12). The target node and the source node of two successive trace steps are the same. If we pre-process the first source node, then we can uniformly process all the trace steps by processing the modifications belonging to the edge and to the target node, because we can assume that the source node was already processed.

The function *annotateGraphImpl* processes the trace steps if the trace sequence is not empty (lines 3-25). The modifications belonging to the first step of the trace are identified

```
1  annotateGraph(Graph graph, Trace trace, Diff diff):
2      (Graph, NodeID*, EdgeID*)
3    (graph', diff')=processPrefix(graph, diff, head(trace))
4    (graph', proceN, procE)=annotateGraphImpl(graph', trace,
5        diff', [head(trace).source], [])
6    return (graph', procN, procE)
```

Listing 4.6: The annotateGraph Algorithm

```
1  procPrefix(Graph graph, DiffSeq diff, TraceStep startT):
2      (Graph, Diff)
3    State startN=startT.source
4    Nat i=0
5    Output* outputN=""
6    while diff[i].mod=="+":
7      outputN+=diff[i].output
8      i+=1
9    getNode(graph, startN).pass=1
10   if i!=0:
11     getNode(graph, startN).diffTab+=((?, ?),
12         (startnode.target, startnode.input), nodeOutput)
13   return (graph,diff[i:])
```

Listing 4.7: The processPrefix Function

by calling *procTraceStep* which returns the modified output belonging to the edge (if it is modified), the modified output belonging to the target node and the rest of the difference sequence (lines 4-5). If a further trace step exists (one after the currently processed), it is used as the target state of the edge, otherwise, the unknown label (?, ?) is used (lines 6-8). The target node of the trace step and the edge represented by the trace step are selected for annotation (line 9-11). The counters which count how many times the transitions and the states referenced by the trace are increased (line 12-13). The output corresponding to the node is recorded even if they have not been modified (lines 14-15). The output corresponds to the edge is recorded only if modification occurred (line 16-17). The processed node and edge are added to the processing sequence if they are not already processed (line 18-23). The rest of the trace and diff sequence are processed recursively by calling the function itself (24-25).

Inferring the Possible Modifications These functions infer the possible transformation modifications from the difference tables in the annotated graph. The tables in the nodes and in the edges are processed in different ways. To process the annotated graph the *inferEdgeMod* and *inferNodeMod* functions take the list of the processed edges and the list of the processed nodes, respectively. They produce the lists of the inferred solutions.

inferNodeMod(Graph graph, ProcNodes procN): List(List(Output*))

inferEdgeMod(Graph graph, ProcEdges procE): List(Output*)

- **Processing node tables:** Inferring modifications in case of the nodes can deliver three kinds of results: one solution is possible (the ideal case), several modifications

```
1  annotateGraphImpl(Graph graph, Trace trace, Diff diff,
2      NodeID procN, EdgeID procE): (Graph, NodeID*, EdgeID*)
3    List(NodeID) procN
4    List(EdgeID) procE
5    if length(trace) > 0:
6      (outputE, outputN, diff')=procTraceStep(head(trace),
7          diff)
8      targetLabel=(?, ?)
9      if length(trace)>1:
10       TargetLabel=(trace[1].target, trace[1].input)
11     nodeData=getNode(graph, head(trace).target)
12     edgeData=getEdge(graph, head(trace).source,
13         head(trace).target, head(trace).input)
14     nodeData.pass+=1
15     edgeData.pass+=1
16     append(nodeData.diffTab, (head(trace).source,
17         head(trace).input), targetLabel, outputN))
18     if outputE!=head(trace).output:
19       append(edgedict.diffTab, outputE)
20     if head(trace).target not in procN:
21       append(procN, head(trace).target)
22     if (head(trace).source, head(trace).target,
23         head(trace).input) not in procE:
24       append(procE, (head(trace).source,
25           head(trace).target, head(trace).input))
26     (graph, procN, procE)=annotateGraphImpl(trace[1:],
27         diff', graph, procN, procE)
28   return (graph, procN, procE)
```

Listing 4.8: The annotateGraphImpl Function

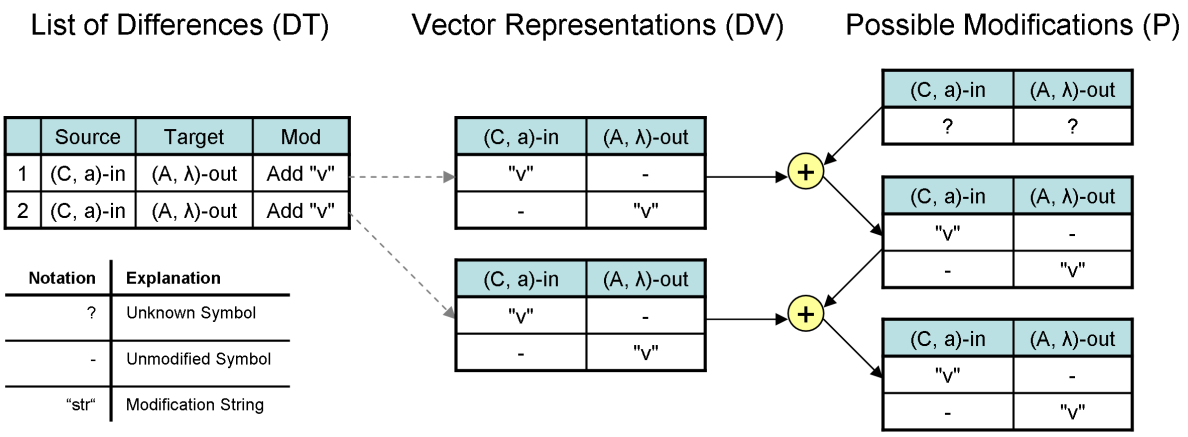


Figure 4.14: Resolution of difference tables

are possible and a contradiction has occurred. Processing node tables are done in two steps: first, they are converted to a new representation in which the columns represent edges connected to the node; then solutions are inferred from the new representation. The detailed algorithm will be explained after the description of an example below.

- **Processing edge tables:** Inferring modifications from the differences in case of the edges are simple: the modifications imposed by all lines in the table must be identical in the table in order to deduce a consistent modification. A difference between the lines means that the same parts of the output instructions are modified in an inconsistent way by the user.

Example: The leftmost table of the Figure 4.14 shows the *difference table (DT)* of Node 1 presented in Figure 4.13. The tables in the middle of the figure represent the *difference vectors (DV)* that corresponds to the lines of the left table. The tables on the right side of the figure show the changes prescribed by the *possible modifications (P)* after the vectors are processed. The difference vector (DV) contains a column for each input and output edge, which is labeled with the identifier of the edge and its direction. The column structure of the possible modification tables (P) are the same.

The correspondence between the lines of the list of differences and their vector representation is shown by the dashed (grey) arrows. (1) The first line of the list of difference table (DT) is represented by two vectors (DV), because the token can be added either before traversing the node or after traversing it. (4) The second element of the difference list represents the same difference again and this element is processed in the same way as the first one.

The uppermost table (P) in the right column represents the initial state of the possible modifications. The second table (P) shows the combination of the initial table (P) and the first difference vector (DV). The third table (P) shows the combination of the second table (P) and the second difference vector (DV); and so on. The bottommost table of right column (P) is the final possible modification table which belongs to Node 1.

Details: The inference of the modification of nodes is implemented by the function *inferNodeMod*. It iterates over the list of the nodes *procN* which have to be processed (lines 3-39). A mapping *vecMapping* is created, which maps the labels of the edges to a position in the difference vector (line 4). Then a difference vector (DV) *defaultDV* is initialized with unknown values (line 5). This vector is used to initialize the possible

modification table (P) (line 6). The difference table (DT) is processed in a loop, in which we distinguish four cases (lines 7-39):

- processing a row which is recorded at the start of the trace (source edge is (?, ?));
- processing a row which is recorded at the end of the trace (target edge is (?, ?));
- processing a row which contains no modification;
- processing a row which contains modification.

The element of the difference vector can be marked with the unknown symbol (*?*) i.e. we have no information about the element; the unmodified symbol (in the algorithm marked with *""* and otherwise with - for better readability); and a difference string (*"a string"*). A difference vector can be formed for each line of the difference list as follows:

- If the row is recorded at the start of the trace, a single vector is created with the output symbol of the row for the target edge and the remaining columns will contain the unknown symbol (lines 9-11);
- If the row is recorded at the end of the trace, a single vector is created with the output symbol of the row for the source edge and the remaining columns will contain the unknown symbol (lines 14-16);
- If no difference is recorded in the list, a single vector is created with the unmodified symbol for the source and target edges and the remaining columns will contain the unknown symbol (lines 19-23);
- If a difference is recorded in the list, two vectors are created: the first one contains an unmodified symbol for the source edge and the modification string for the target edge, and in the other case, vice versa. All other elements will contain the unknown symbol (lines 26-30, 32-37).

In the initial state, the table of possible modifications (P) contains a single row filled with the unknown symbol (line 6). Each row of the difference vectors table (DV) is combined with the rows of the possible modification table (P) and the result becomes the new possible modification table (P) (lines 12, 17, 24, 31, 38). The combination of the rows of the tables is carried through by merging each line of the tables (DV, P). Merging two lines mean that two rows are combined element-wise. The rows can be merged successfully if each pair of the elements is the same or one of the elements is the unknown symbol (in this case the result row contains the other element). Otherwise, the merging fails and no result is produced. The tables are combined by forming the union of the result of the row merges (line 38). The final possible modification table (P) is stored in the result sequence *resN*.

The inference of the modification of edges is implemented by the function *inferEdgeMod*. This function also iterates over the edges (lines 3-14). The loop deals with three possible cases:

- if the output of the edge was left unchanged, the result will be the original output of the edge, the value of which indicates no change (lines 5-7);
- if the output of the edge was changed in some cases, the result will be empty, the value of which indicates the contradiction (lines 8-10);
- if the output of the edge was changed in all cases, depending on whether the changes are contradictory, the result will be empty or the value will be changed (lines 11-18).

```
1  inferNodeMod(Graph graph, ProcNodes procN): NodeDT
2    resN=list(list(Ouput*))
3    for node in procN:
4      vecMapping=buildInOutEdgesMapping(graph, node)
5      (Output*)* defaultDV=initUnknownDV(length(vecMapping))
6      possibleM=[defaultDV]
7      for row in node(graph, node).diffTab:
8        if row.source==(?, ?):
9          newRow=defaultDV
10         newRow[vecMapping((row.target[0], row.target[1],
11               'o'))]=row.output
12         possibleM=mergeSoultions(possibleM,newRow)
13       elif row.target==(?, ?):
14         newRow=defaultDV
15         newRow[vecMapping((row.source[0], row.source[1],
16               'i'))]=row.output
17         possibleM=mergeSoultions(possibleM,newRow)
18       elif row.output=="":
19         newRow=defaultDV
20         newRow[vecMapping[(row.source[0], row.source[1],
21               'i')]]=""
22         newRow[vecMapping[(row.target[0], row.target[1],
23               'o')]]=""
24         possibleM=mergeSoultions(possibleM,newRow)
25       else:
26         newRowA=defaultDV
27         newRowA[vecMappings[(row.source[0], row.source[1],
28               'i')]]=row.output
29         newRowA[vecMappings[(row.target[0], row.target[1],
30               'o')]]=""
31         newpossibleMA=mergeSoultions(possibleM,newRowA)
32         newRowB=defaultDV
33         newRowB[vecMappings[(row.source[0], row.source[1],
34               'i')]]=""
35         newRowB[vecMappings[(row.target[0], row.target[1],
36               'o')]]=row.input
37         newpossibleMB=mergeSoultions(possibleM,newRowB)
38         possibleM=union(newSolutionA, newSolutionB)
39     resN=resN+[possibleM]
40   return resN
```

Listing 4.9: The inferNodeMod Algorithm

```
1  inferEdgeMod(Graph graph, ProcEdges procE): List(Output*)
2    resE=list(Output*)
3    for edge in procE:
4      edgeData=getEdge(graph, edge[0], edge[1], edge[2])
5      if length(edgeData.diffTab)<edgedata.pass
6          length(edgeData.diffTab)==0
7          resE=res+[[edgeData.output]]
8      elif length(edgeData.diffTab)<edgeData.pass
9          length(edgeData.diffTab)!=0:
10         resE=res+[list(Output*)]
11     else
12       solution=[?]
13       for row in edgeData.diffTab
14         if solution!=[row.output]:
15           solution=list(Output*)
16         if solution==[?]:
17           solution=[row.output]
18       resE=res+[solution]
19   return resE
```

Listing 4.10: The inferEdgeMod Algorithm

Modifying the Transformation Two functions are responsible for the modifications of the transducer according to the list of possible modifications. The function *inferEditScript* creates an edit script by selecting one of the modification for each nodes and edges from the possible modifications. The function *modifyTransducer* modifies the transducer according to the edit script.

> inferEditScript(Graph g, ProcNodes resN, List((Output*)*) procN, ProcEdges resE, List(Output*) procE): EditScript
>
> modifyTransducer(Transducer trans, EditScript m): Transducer

The edit script which is used to update transformation is assembled from the sequence of possible modification according to rules as follows:

- If a *consistent modification occurred in an edge*, the original output is replaced by the modified output.
- If *inconsistent modifications occurred in an edge*, an error message can be emitted or the designer can be prompted to resolve the conflict.
- If a *consistent single modification was inferred in a node*, it is appended to the beginning or the end of the output of the appropriate transition.
- If a *modification occurred in a node and multiple solutions were inferred*, the one with smaller number of modifications is selected (if solutions with the same number of modifications are available, the first one is chosen).
- In case of *inconsistent modifications in a node*, an error message can be emitted or the designer can be prompted to resolve the conflict manually.

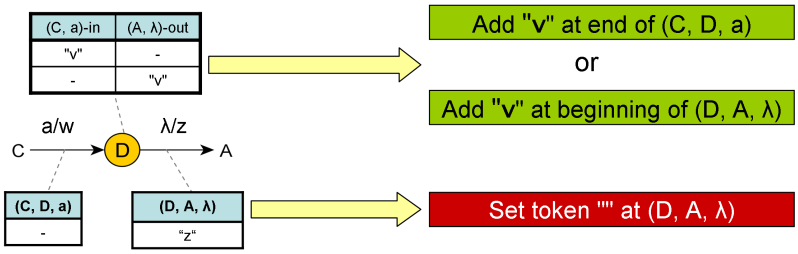


Figure 4.15: Creating the Edit Script from Possible Modification Tables

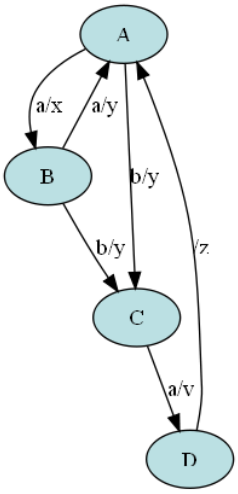


Figure 4.16: The Inferred Transducer

As a result, our error correction technique is automated in case of consistent minor modifications, and it becomes semi-automated when conflict resolution requires input by the designer.

Example: The edit script of the described example is the following: replace with ” the output of the transition from state C reading ’a’; insert ’v’ to the beginning of the output of the transition from state C reading ’a’. The relation between the possible modifications and the inferred edit script of the modified part of the transducer can be seen in Figure 4.15. The final transducer can be seen in Figure 4.16.

4.5 Extension of the Algorithms

In the description of the algorithms we described (for clarity reasons) the processing of a single input/output pair as an example. This section describes a minor modification of the algorithm *Graph Annotation Based Inference (GABI)* to accept a multiple input and output pairs (this makes it more usable for practical problem solving). To make the comparison of the two algorithms easier we also sketch the extension of the algorithm *Direct Trace Modification Based Inference (DTMBI)*.

We present the extended version of the *inferTransducer* in Listing 4.11. This algorithm differs from Listing 4.5 in three points:

- the function returns a list of possible solutions and expects lists of input and expected output combinations;
- it has the capability to handle multiple input/output pairs (lines 3-18);

```
1  inferMultipleTransducer(Transducer trans, List(Input*)
2        inputL, List(Output*) outputL'): Transducer*
3    Graph g=getGraph(trans)
4    List(NodeID) procN
5    List(EdgeID) procE
6    while length(inputL)>0:
7      input=head(inputL)
8      output'=head(outputL')
9      inputL=tail(inputL)
10     outputL'=tail(outputL')
11     Trace tr=execute(trans, inuput)
12     Output* o=getOutput(tr)
13     Diff diff=calculateDiff(output, output')
14     (g',procN', procE')
15           =annotateGraph(tr, diff, g, procN, procE)
16     g=g'
17     procN=procN'
18     procE=procE'
19   List(List(Output*)) resN=inferNodeMod(g', procN)
20   List(Output*) resE=inferEdgeMod(g', procE)
21   List(EditScript) mL
22         =inferEditScriptList(g, resN, procN, resE, procE)
23   List(Transducer) transL'
24   while length(mL)>0:
25     m=head(mL)
26     Transducer trans'=modifyTransducer(trans, m)
27     transL'=transL'+[trans']
28   return transL'
```

Listing 4.11: The Main Algorithm Returning Multiple Solutions

- it has the capability to return all possible solutions (lines 21-27).

Handling Multiple Examples The algorithms need slight modifications to handle multiple input/output pairs. This feature is necessary to create test cases which can produce specific values of the selected metrics e.g. to reach complete transition coverage in a transducer which has a tree structure.

In the GABI algorithm the function *annotateGraph* is executed successively on each difference sequence by keeping the previous annotated graph. Up to this point each input/output pair can be processed separately in the same way as in the simple version of the algorithm (lines 11-13). To handle multiple pairs we have to summarize the inputs in some point of the algorithm execution (lines 14-15). The *annotateGraph* function is extended with two extra parameters to get the list of the processed nodes and edges instead of initializing them with an empty list internally. Consequently, the annotations are summarized in a single annotation graph.

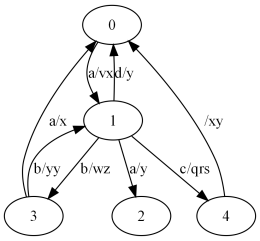
In the DTMBI algorithm the result can be inferred by producing the modified trace for each input separately and then the function modifyTransducer is executed successively on each modified traces. The extra constraint is that we allow only to modify the transitions contained in the processed trace and we do not allow to change the initial state of the transducer.

Producing Multiple Solutions The described version of the GABI algorithm selects the inferred solution from all the possible versions with quite simple heuristics (i.e.: selecting the first combination). To check the correctness of the algorithm we have to verify that the correct solution is always listed among the solution candidates. This feature is also necessary to create better heuristics by evaluating the properties of the inferred solutions. The function *annotateGraph* is modified between the lines. The function *inferEditScriptList* returns a list of solutions instead of a single solution.

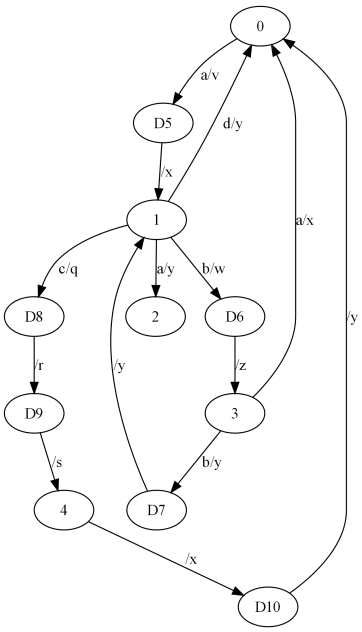
Multiple Output Symbol Handling The DTMBI algorithm is implemented in a way that the transition rules are allowed to produce a single output symbol only. Producing multiple output symbols can be simulated by a sequence of transitions each of which produces a single output (normalization), e.g.: transition [(A, a,B, xyz)] can be encoded by the transitions [(A, a, N1, x), (N1, λ, N2, y), (N2, λ, z, B)]. To make the result of the DTMBI algorithm comparable to the result of the GABI, we have to reverse the effect of the normalization. This can be done by merging the transition into the preceding transition in case of reading empty input symbols (de-normalization). An example of the normalization and the de-normalization can be seen in Figure 4.17a, 4.17b and Figure 4.17c, 4.17d, respectively.

A similar algorithm is described in [34] in connection with the problem of eliminating lambda input in PDFAs. The detailed description of the de-normalization algorithm is as follows:

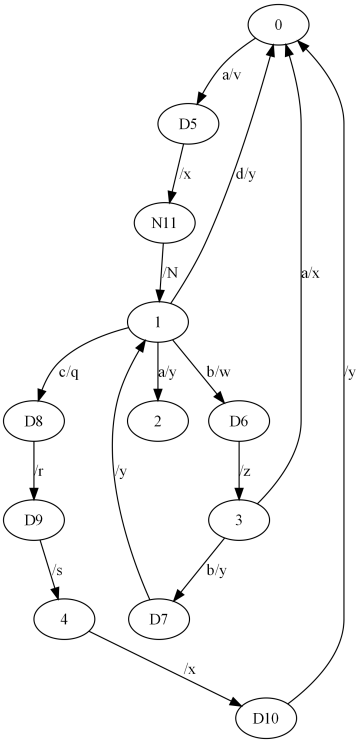
- A transition is selected for de-normalization if it contains multiple output symbols.
- New transitions are created to output the additional output symbols. The transitions are connected to each other through new unique states and they read no input symbols. The target state of the last newly created transition equals with the target state of the selected transition.
- The selected transition is modified to output only its first output symbol and its target state is the source state of the first newly created transition.



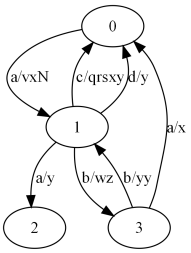
(a) Multiple output symbols per transition



(b) Single output symbol per transition



(c) Single output symbol per transition



(d) Multiple output symbols per transition

Figure 4.17: Example for the Normalization/De-normalization

The detailed algorithm of the normalization is as follows:

- A transition is selected for elimination if it reads no input (a), its source state is not equal with the initial state of the transducer (b), and the number of incoming edges is one.
- The selected transition is merged into the preceding transition by appending its output to the predecessors' output and by overwriting the target-state of the predecessor with its target-state.
- The selected transition and the intermediate state are deleted.

4.6 Summary

This chapter was driven by two ideas: the MTBE approach can be extended to M2T transformation modification by example; and M2T transformations can be modeled by transducers. As a consequence of the two idea we formulated the transducer modifications problem. This formulation is important, because 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 reasonably reduce the solution space of inference process.

We developed two algorithms to infer transducer modifications from input/expected output pairs. We also described extensions and improvements of the algorithms. We have described the two algorithms, while their experimental evaluation will be addressed in the next chapter.

Chapter 5

Evaluating the Inference Algorithms

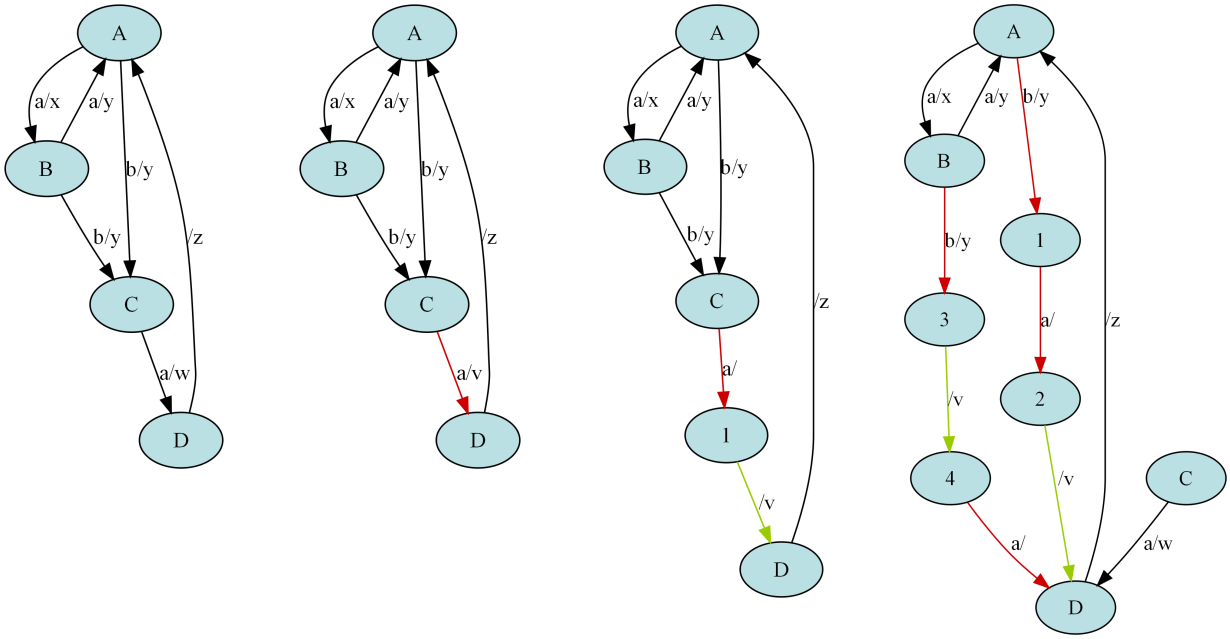
In this chapter we describe various metrics which can be used to evaluate algorithms for transducer modification inference by example. We also evaluate the algorithms which we proposed in the previous chapter qualitatively and quantitatively. The structure of the chapter is explained at the end of the introductory section.

5.1 Introduction

In the previous chapter we have described two algorithms which are capable to infer modified transducer from a set of example pairs and an original transducer. The simplest evaluation we can carry out is that we compare the inferred transducer to the expected result e.g. intuitively we expect from the algorithm to modify the original transducer as little as possible; and preserve the behavior of the transducer with respect to any arbitrary input as much as possible. Evaluating the algorithms merely on bases of comparison the expected and inferred outputs are misleading. The effectiveness of the algorithms depends on several factors: complexity of the original transducers, the extent of the modification, the properties of the examples, the properties of the expected answer. We analyze, how the above mentioned informal requirements can be expressed formally. These formalized requirements can be expressed by metrics which must be minimized by the algorithms. We propose and adapt metrics which can express the quality of the algorithms. We describe white-box and black-box properties of the transducer modifications with structural and behavioral metrics, respectively. We also adapt coverage metrics to express the quality of the example sets with respect to fixed transducers. Finally, we propose metrics which combine white-box and black-box approaches to express the quality of the transformations.

The described algorithms are correct in the sense that they produce the expected output by executing the example input. To study the effectiveness of our algorithms we create a small set of examples which are evaluated manually. Then we check how the proposed metrics reflects the quality of the algorithms on the constructed examples. Finally, we generate a large set of examples which we are going to evaluate quantitatively.

The remainder of this chapter is structured as follows: In Section 5.2 we describe candidate metrics: we collect different kind of metrics and we use them to compare some possible solutions. In Section 5.3 we evaluate the algorithms which have been proposed in Chapter 4.



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

Figure 5.1: Different Modifications of the Transducers

5.2 Description of the Applicable Metrics

We can analyze the quality of the inference algorithms by evaluating the results manually as we did in Section 4.3.3. 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 effectiveness 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. In Figure 5.1 one can see 4 transducers. The left-most is the original one. The three other transducers are various modifications of the original transducer. Two of them are modifications by the developers and the right-most is inferred by an algorithm.

5.2.1 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,

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 5.2: Metrics of the Example

- 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.

In Figure 5.2 we can see the metrics of the different modifications shown in Figure 5.1. 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. We may conclude that solutions provided by human try to create as small modifications as possible.

5.2.2 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 possible input strings with a maximum length n can be generated easily. The simplest behavioral 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 also measure:

- 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 5.3 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 5.4 we can see the behavioral metrics of these inputs up to length four with respect to the example we have studied.

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 5.3: Example Input and Output Pairs with Maximum Length Four

N	All/Different	Avg. length until 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 5.4: Behavioral Metrics with Maximum Length Four

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

Figure 5.5: Coverage metrics of the example

5.2.3 Coverage Metrics for Transducers

In case of white box testing of software, the quality of the test cases is measured by test coverage [98]. 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 traversed paths out of all possible paths.

The coverage results of the example can be seen in Figure 5.2.3. 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 (in case of non safety-critical software). 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.

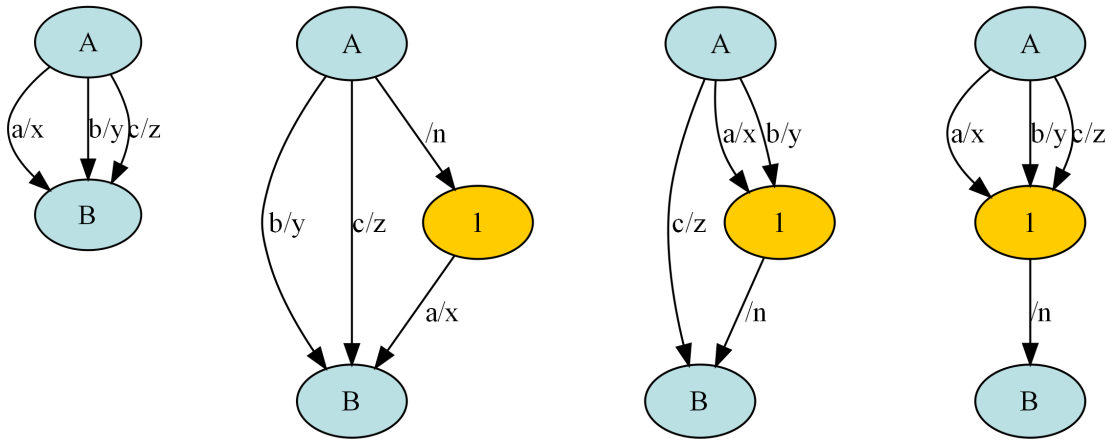
5.2.4 Goal, Questions, Metrics Paradigm

The metrics defined in Section 5.2.1, Section 5.2.2, Section 5.2.3 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 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 adapt the GQM (goal, quality, metrics) paradigm of software engineering [46].

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 in relation with 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 5.2.5.



(a) The Original (b) A Single Path is Modified (c) A Set of Paths are Modified (d) All Paths are Modified

Figure 5.6: Single Point Modifications of Branche

- Are the examples minimal? The question whether the examples are minimal can be decided easily: examples are 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. This can be decided by providing the two sets.

We are going to show in the next section how the above questions may be expressed by metrics.

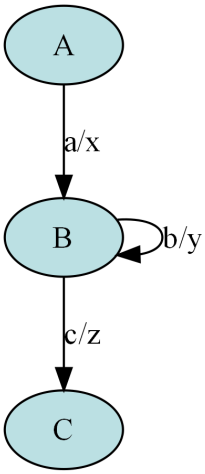
5.2.5 Metrics of the User Intention

In case of algorithms which are learning the modification from examples, it is important to examine whether it is theoretically possible to do so. In this section we 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:

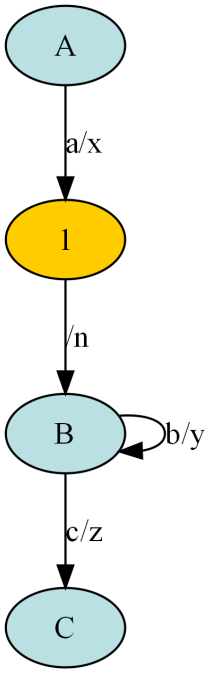
- a combinatorial one and
- a psychological one.

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 section 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 the results of psychological experiments).

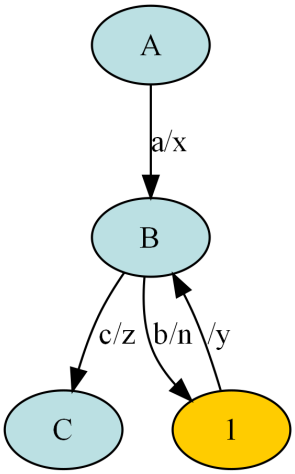
First, we examine the examples of inference settings: we take example transducers and their possible modifications. We list all possible example traces which may be able to demonstrate the modification. Then we investigate which other modifications may be represented by the same example trace. Finally, 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 transducer and sequence of examples.



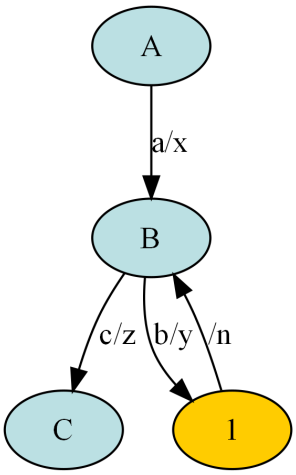
(a) The Original



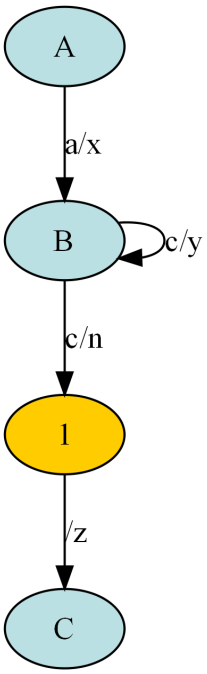
(b) Modified Before the Cycle



(c) Modified at the Beginning of the Cycle



(d) Modified at the End of the Cycle



(e) Modified After the Cycle

Figure 5.7: Single Point Modifications of a Cycle

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 5.6b, 5.6c, 5.6d) contains single point modifications of a transducer containing two states and three edges between these two states (Figure 5.6a). 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 5.7b, 5.7c, 5.7d, 5.7e) contains single point modifications around a loop in a transducer (Figure 5.7a). 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 consider an example sensible if it includes 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 5.8 and Figure 5.9. The first column contains the label of modified transducer. The second column describes the considered examples, demonstrating the example (we do not list the irrelevant examples like paths only covering the unmodified part of the transducer). The third column describes 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 unambiguity (comb.) and intuitive unambiguity (int.). The combinatorial unambiguity is a value between 0 and 1 showing the probability of choosing the same interpretation as the selected modified transducer. The intuitive unambiguity 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 5.10 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 5.11 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 unambiguity and the intuitive unambiguity 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 similarly to 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 the set of states around which the insertion must occur as the basic building block of such metric.

In Figure 5.12 one can see three graphs. In Figure 5.12a one can see a graph which is the output abstraction of the transducers in Figure 5.12b and Figure 5.12c. The edges and the nodes are labeled with output symbols and states, respectively. The graph in Figure 5.12a can be the abstraction of any deterministic transducer having the same states and transitions producing the same outputs symbols. In Figure 5.13a and in Figure 5.13b one can see the output abstractions of Figure 5.6a and Figure 5.7a, respectively.

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	Maybe
		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

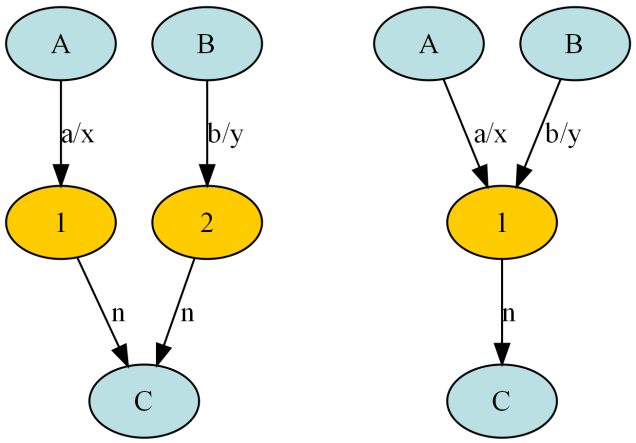
Figure 5.8: The Interpretations of the Examples by the Single Point Modifications of Branches

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

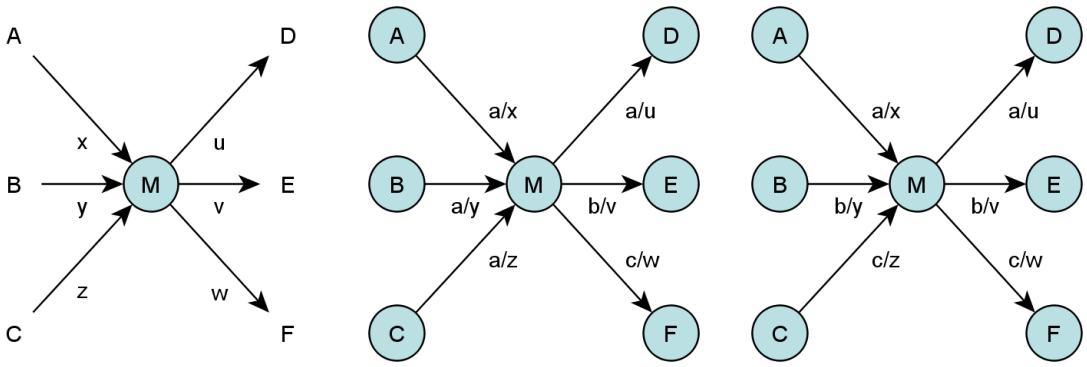
Example	Interpretation	Int.
A modified path	A single path	Yes
	A set of paths	No
	All paths	Maybe
A modified path + an unmodified one	A single path	No
	All paths except the unmodified one	Yes
A set of modified paths	A set of paths	Yes
	All paths	Maybe
All paths	All paths are modified as they are shown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

Figure 5.10: Summary of the Interpretations



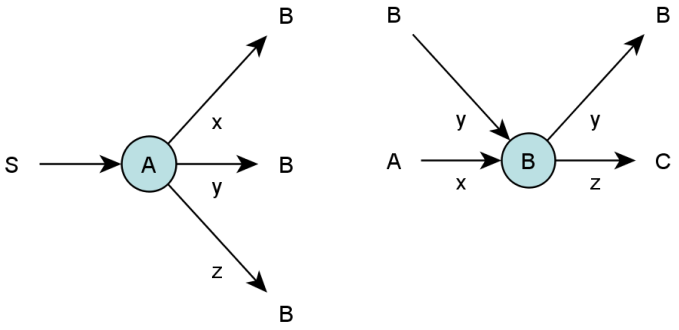
(a) Version A (b) Version B

Figure 5.11: Two Transducers with Identical Behavior



(a) Output Abstraction (b) Example Transducer A (c) Example Transducer B

Figure 5.12: The Output Abstraction of the Transducers



(a) Abstraction of the Branches (b) Abstraction of the Cycle

Figure 5.13: The States to which the Insertions Belonging

Consider an arbitrary but fixed transducer t of which the Figure 5.12a 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;
- several ways exist to insert a state N connected to M and modify the connecting edges to produce extra output n.

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 incoming edges and m outgoing 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 5.12a 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 2^3-1 different way; we can insert the edge n between the node M and the out-going edges (u, v, w) in a similar way, which is additional 2^3-1 permutations; so, the possible ways to insert a single point modification into this example is $2^3-1+2^3-1=14$);

- the number of all possible paths is n*m.

The metrics of the quality of the example with respect to expressing the intention of the specifier can be described 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 unambiguity 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 unambiguity.

The combinatorial unambiguity 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 unambiguity 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 unambiguity 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.

5.3 Evaluation of the Algorithms

In this section we evaluate the algorithms described in Chapter 4, namely, the *Direct Trace Modification Based Inference (DTMBI)* and the *Graph Annotation Based Inference (GABI)*. The qualitative evaluation can be performed by evaluating examples and comparing the inferred results to the expected results. The quantitative evaluation can be carried out by generating several examples and calculating metrics over them.

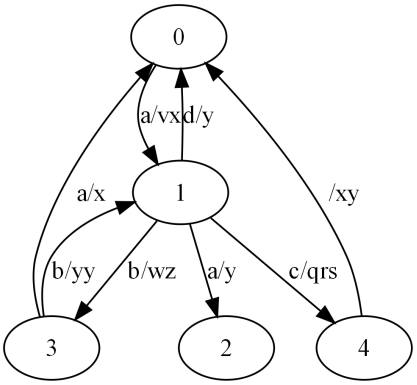
5.3.1 Qualitative Evaluation

In this section we evaluate manually the behavior of the algorithms on three examples. We present the input/expected-output pair, the original transducer, the expected modification and the result of the two inference algorithms.

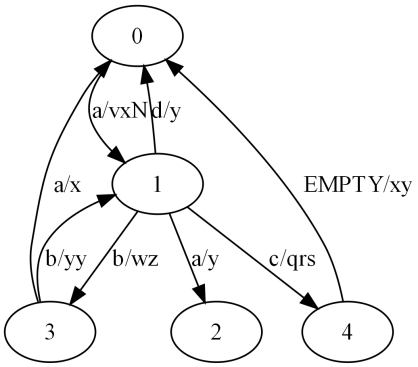
The expected result was derived from a survey which was conducted at Technical University of Budapest among 12 PhD students in computer science. The survey contained the same examples and the participants had to choose between the possible modifications. There were no uniquely preferred solutions. A major critique of the students was that they would select these preferences much more surely if the examples would contain some domain specific labels and not meaningless letters and numbers. In the current experiment most of the students have selected the modifications which cause the smallest change in the transducer.

The first example demonstrates a single point modification of the transducer which can be learned from a single input/output pair. The algorithms take the inputs as follows:

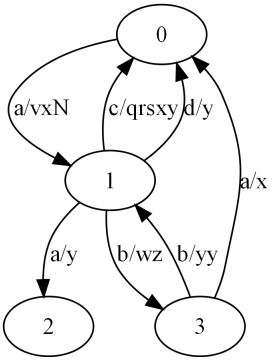
- the transducers can be seen in Figure 5.14a,



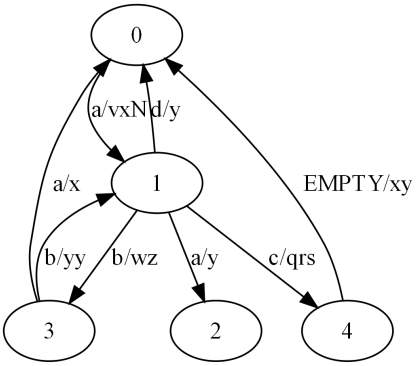
(a) The Original Transducer



(b) Expected Solution

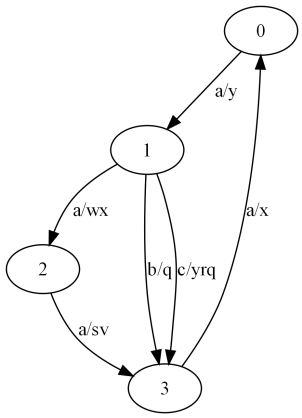


(c) Inferred by DTMBI Algorithm

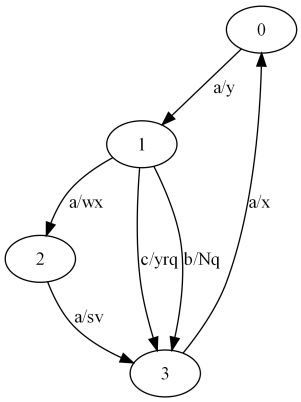


(d) Inferred by GABI Algorithm

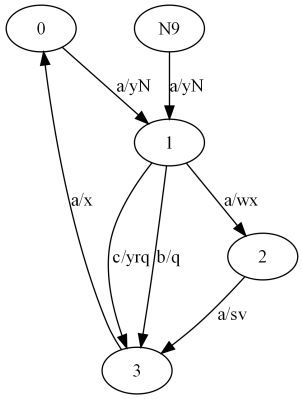
Figure 5.14: Example 1



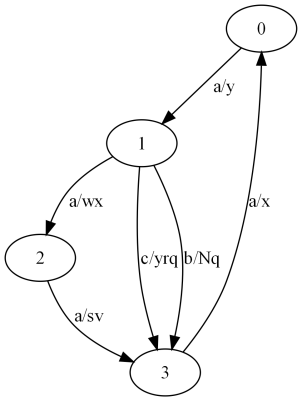
(a) The Original Transducer



(b) Expected Solution



(c) Intermediate result of the DTMBI Algorithm



(d) Inferred by GABI Algorithm

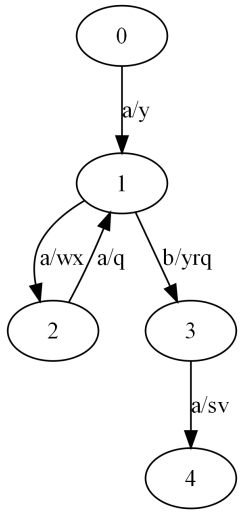
Figure 5.15: Example 2

- the input string is "abb" and
- the expected output string is "vxNwzyy".

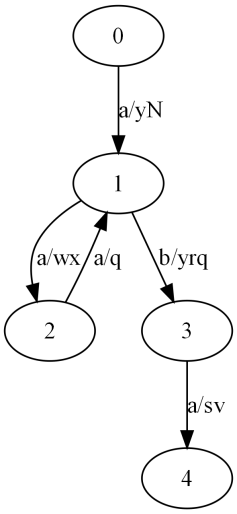
The original transducer produces "vxwzyy" for the specified input string. During the processing of the input string the transducer reaches the states: 0, 1, 3, and 1. We know that the additional output "N" must be produced in one of the following transitions: the transition between 0 and 1 or the transition between 1 and 3. The human solutions prefer appending the solution which appends "N" to the end of the output of the transition between 0 and 1 (Figure 5.14b). We can see that in this case the two algorithms select the same solution (Figure 5.14c, 5.14d). The difference is caused by the normalization/denormalization process which eliminated the node 4.

The second example demonstrates modifications of multiple paths; here multiple input/output pairs are specified. The algorithms take the inputs as follows:

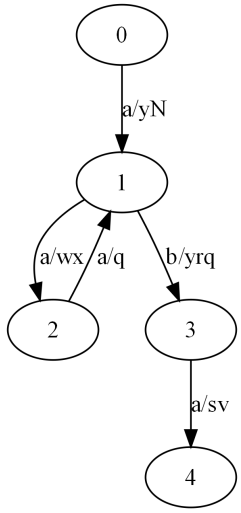
- the transducers can be seen in Figure 5.15a,
- the first input string is "aaa",
- the first expected output string is "yNwxsv",
- the second input string is "aba",



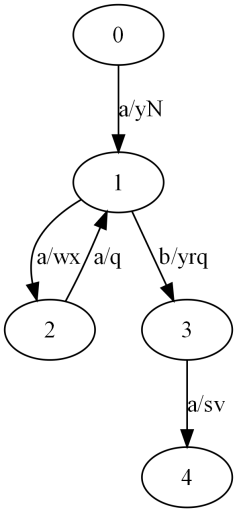
(a) The Original Transducer



(b) Expected Solution



(c) Inferred by DTMBI Algorithm



(d) Inferred by GABI Algorithm

Figure 5.16: Example 3

- the second expected output string is "yNqx".

The original transducer produces "ywxsv" and "yqx" for the first and second input string, respectively. During the processing of the first input string the transducer reaches the states: 0, 1, 2, and 3. During the processing of the second input string the transducer reaches the states: 0, 1, 3, and 1. The transducers can be seen in Figure 5.15. In this example the DTMBI algorithm stops during the processing. The termination is caused by the constraint that the initial state can be modified only once. The renamed states of the reoccurring trace elements do not match to the renamed edges, therefore the algorithm tries to modify the initial state for a second time (which causes the termination). In Figure 5.15c, we present the partially modified form of the transducer before the algorithm terminates (in this transducer the initial state must be modified to the state "N9" to produce the right output to the second input/output pair).

The third example demonstrates a single modification of a cycle in the transducer; single input/output pairs are specified. The algorithms take the inputs as follows:

- the transducers can be seen in Figure 5.16a,
- the input string is "aaaba",
- the expected output string is "yNwxqyrqsv".

The original transducer produces "ywxqyrqsv" for the input string. During the processing of the input string the transducer reaches the states: 0, 1, 2, 1, 3 and 4. The transducers can be seen in Figure 5.16. Both algorithms deliver the expected solution.

The fourth example demonstrates a single modification of a cycle in the transducer; single input/output pairs are specified. In this example we cover partially the same path twice. The algorithms take the inputs as follows:

- the transducers can be seen in Figure 5.17a,
- the input string is "aaaaaba",
- the expected output string is "yNwxqNwxqyrqsv".

The original transducer produces "ywxqwxqyrqsv" for the input string. During the processing of the input string the transducer reaches the states: 0, 1, 2, 1, 3, 1, 3 and 4. The transducers can be seen in Figure 5.17. By covering the cycle between state 1 and 2 the DTMBI algorithm delivers a completely different result than the GABI algorithm.

5.3.2 Quantitative Evaluation

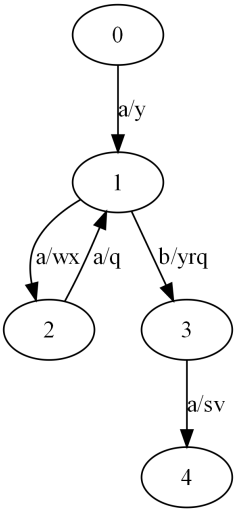
In this section we compute and compare some metrics of the execution of the two algorithms. First, we generate the metrics from the examples presented in the previous section. Then we generate random graphs and inject random mutations to demonstrate the metrics over a larger set of examples.

Metrics of the Hand-Crafted Examples In Figure 5.18 we present the metrics calculated from the presented examples. The presented metrics are described in Section 4.4. The first and the second row contains the number of the exercises and the selected algorithms (D - DTMBI; G - GABI), respectively. The further rows contain the metrics grouped by their types:

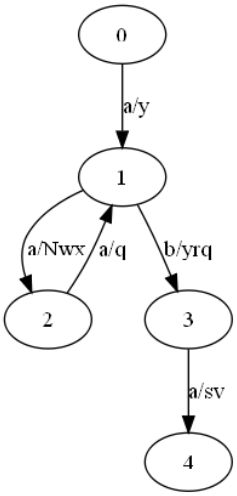
- Structural Metrics of the Transducer Modification,
- Behavioral Metrics of the Transducer Modification,
- Coverage Metrics of the Transducer Modification,
- Metric of the Intention.

We interpret the modifications as additions and deletions. The GABI algorithm can deliver several alternative solutions, which we display in separate columns. If the algorithm cannot produce evaluable results we omit the metrics in the specific column.

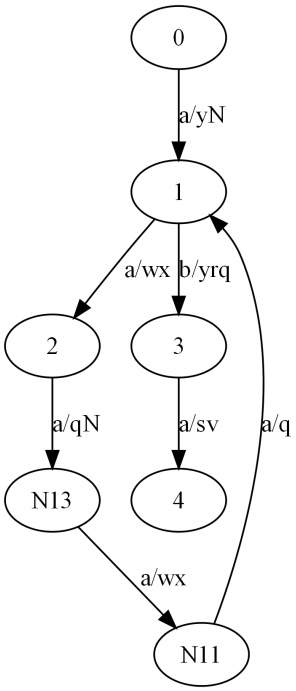
In case of the first and third examples there are no significant differences. In case of the second example we have no comparable results. In the fourth case we can see significant difference between the two solutions: the DTMBI delivers a much larger transducer: the number of the states is increased by 40% and the number of the transitions is increased by 40% (80% addition and 40% deletion).



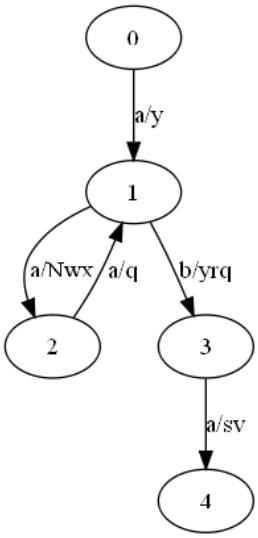
(a) The Original Transducer



(b) Expected Solution



(c) Inferred by DTMBI Algorithm



(d) Inferred by GABI Algorithm

Figure 5.17: Example 4

Example	1		2		3		4	
Algorithm	D	G	D	G	D	G	D	G
No. of new states	0	0	-	0	0	0	2	0
No. of deleted states	1	0	-	0	0	0	0	0
No. of new states / No. of original states	0.00	0.00	-	0.00	0.00	0.00	0.40	0.00
No. of deleted states / No. of original states	0.20	0.00	-	0.00	0.00	0.00	0.00	0.00
No. of new trans.	2	2	-	1	1	1	4	1
No. of deleted trans.	3	2	-	1	1	1	2	1
No. of new trans. / No. of original trans.	0.25	0.25	-	0.16	0.20	0.20	0.80	0.20
No. of deleted trans. / No. of original trans.	0.37	0.25	-	0.16	0.20	0.20	0.40	0.20
Modified Output	1.00	1.00	-	0.68	1.00	1.00	1.00	0.86
Average length till the first difference	2.0	2.0	-	4.3	1.0	1.0	1.0	1.0
Average number of changed characters	3.2	7.9	-	1.5	1.0	1.0	5.9	2.8
State Coverage	0.38	0.38	-	0.83	1.00	1.00	1.00	1.00
Transition Coverage	0.60	0.60	-	0.80	1.00	1.00	1.00	1.00
No. of possible interpretations	2	2	-	1	2	2	1	1

Figure 5.18: Summary of the Metrics

Random Transducer Generation To produce a comparable amount of examples for the measurement we generate transducers randomly. Our random transducer generation is based on the combination of random graph generator algorithms [80].

- We first generate a random connected directed graph. This ensures that all nodes are reachable. Then, we generate a random undirected graph of which edges are added to the first graph as directed edges pointing form higher to lower node id numbers. Adding such edges introduce loops.
- After the creation of the graph random mutations are introduced. Mutation created by selecting edges randomly in the graph and by modifying their output.
- Random paths are created: the shortest path is selected between node 0 and the mutated edges. This path also contains random nodes of the graph.
- The input/output pairs are generated: the input is calculated from the path; the output is calculated by executing the mutated transducer with the generated input.
- The two algorithms are used to generate the solutions from the original transducers and the input/output pairs.
- The results are compared to mutated transducers.
- The metrics are computed from the results.

Example Set	1		2	
Algorithm	D	G	D	G
No. of examples with mutations	97	97	83	83
No. of examples without mutations	3	3	17	17
No. of successful inferences	3	69	17	95
No. of unsuccessful inferences	97	31	83	5
No Output Case 1	17	-	2	-
No Output Case 2	12	-	9	-
No Output Case 3	-	10	-	1
No. of inferences produced results	68	21	72	4

Figure 5.19: Inference Results of the 100 Generated Transducers

Results of the Generated Transducers We generated two sets of the examples with one hundred transducers each. In the first set we have introduced a high number of mutations per transducer (on average 4.5 edges are modified; the maximum is 5). In the second set we have introduced low number of mutations per transducers (on average 1.5 edges are modified; the maximum is 2). The first set and second set contain three and 17 examples with no mutations. The result of the experiment is summarized in Figure 5.19 and the complete list can be found in [61].

Comparing the results we can state that the GABI algorithm delivers much better results (comparing only examples with modifications). From the first set (97 examples) and the second set (83 examples) the GABI was capable to infer the expected result in 64.02 percent and 64.74 percent of the examples, respectively. The DTBMI was not capable to infer anything except those transducers that have no modifications.

We studied why the algorithms have calculated no solution or not the expected solutions. The reasons for the inference algorithms producing no solutions are as follows:

- No Output Case 1: The DTMBI algorithm produces no results for multiple example pairs: the goal of the algorithm is to produce the lowest difference in the output while keeping the constraints to produce the right solution for the already presented examples. This makes the algorithm to decide "early" about the preferred results. In case of single input/output pairs, this behavior leads in the worst case to a solution which only accepts the specified example. In case of multiple input/output pairs the algorithm in the worst case requires two different initial states for the same transducer which is impossible. This is an expected behavior.

- No Output Case 2: This is a defect in the described DTMBI algorithm: the algorithm does not handle the special case in which inserting transitions with empty inputs can form a circle (a transition sequence that has the same source and destination state) that requires no input. This renders the transducers to be nondeterministic, because the "loop" can be executed arbitrarily many times without reading from the input.

- No Output Case 3: The GABI algorithm produces no results if the examples are inconsistent. This case is described in details in Section 4.4.3.

In some cases, the GABI algorithm does not produce output (because it finds the input inconsistent - No Output Case 3) or it produces output which differs from the expected outcome. We have investigated this problem, because all the examples were generated by valid mutations of the transducers on which the algorithm was expected to work correctly. The result of the investigation was that the used differencing algorithm

Example Set	1		2	
Algorithm	D	G	D	G
No. of new states	5.4	0	2.0	0
No. of deleted states	0.6	0	0.5	0
No. of new states / No. of original states	0.54	0.00	0.20	0.00
No. of deleted states / No. of original states	0.06	0.00	0.05	0.00
No. of new trans.	8.2	4.4	3.2	2
No. of deleted trans.	4.7	4.4	2.8	2
No. of new trans. / No. of original trans.	0.33	0.18	0.13	0.08
No. of deleted trans. / No. of original trans.	0.19	0.18	0.12	0.08
Modified Output	0.72	0.67	0.64	0.41
Average length till the first difference	3	3.2	3.1	3.4
Average number of changed characters	3.3	2.5	3.3	2.0
State Coverage	0.44	0.44	0.23	0.23
Transition Coverage	0.77	0.77	0.53	0.53
No. of possible interpretations	2.4	2.4	1.6	1.6

Figure 5.20: Average of the Metrics of the Generated Sets

sometime provides "misleading" difference sequence. If the result of the differentiation algorithm is replaced with a hand crafted difference sequence (i.e. one which reflect the order of modifications) and the rest of the computation is fed with this difference sequence the algorithm works as it is expected.

To demonstrate the problem consider a character sequence 'x', 'y', 'z', 'v' and replace 'y' with '0' and replace 'z' with '1'; the result will be 'x', '0', '1', 'v'. The expected difference sequence by the algorithm is 'x', '-y', '+0', '-z', '+1', 'v'. In practice the selected differentiation algorithm produces 'x', '-y', '-z', '+0', '+1', 'v', which does not reflects the order of the modifications. This behavior explains why more mutations lead to more failed inferences: in case of a random selection of transitions for modification, the more transitions have been selected, the higher the chance is that two successive transitions have been selected. The difference between start and end states contains incomplete information about the intermediate states, thus in general it is not possible to create a better difference sequence generation algorithm. However, we could modify the described algorithm in such a way that it takes the difference sequence as input instead of the expected result, e.g. the modification steps which are recorded by an editor.

Metrics of the Generated Transducers The average values of the metrics are displayed in Figure 5.20. By comparing the structural metrics we can see that GABI results less structural modifications and the differences between the output of the original and the output of the modified transducer are lower than the results of DTMBI. Because the lower number of changes both in the structure and in the output, we can claim that the GABI is the superior algorithm.

- The DTBMI algorithm added and deleted several states and transitions.

- The GABI algorithm behaved as it was expected i.e. it did not introduce or remove any states and modified only transitions (that is why the number of transition additions and deletions are equal).

The behavioral metrics of the algorithms (the differences in the output of the original and the inferred transducer for a bounded length enumeration of the possible inputs) are surprising: we expected a lower number in case of DTBMI. In case of the second set the GABI algorithm has a significantly lower number. This means that with respect to the behavioral metrics the GABI algorithm produces better results too. The coverage metrics and the number of possible interpretations correlate with the number of modification introduced in the modified transducers. These metrics contain information about the quality of the examples. In our experiment the GABI algorithm was superior independently from the quality of the examples.

5.3.3 Threats to the Validity

In this section we review the three main assumptions which we formulated at the beginning of the problem description:

1. M2T transformations are subject to simple modifications (and not complex changes);
2. changes effect the output operations and not the control flow;
3. existing examples are always handled correctly.

These assumptions are derived from our motivational project which is described in Chapter 3. The first and second assumption holds to most of the cases in the described project and we did not intend to solve the complex cases. In cases to which these assumption do not hold we may get no solution or an over generalized solution. The third assumption is a clear technical requirement which is satisfied by the algorithms. The assumptions may not be fully general, but at least, they show a practically relevant problem setup.

The conceptual error in the qualitative evaluation can be the overlook of an important corner case to which our algorithms fails to infer solutions. Such a corner case was found by the quantitative evaluation: the "No Output Case 2" described in Section 5.3.2. We have not found any more such corner cases during tests.

The potential problem in the quantitative evaluation can be that the random examples we generated may differ in distribution from the ones which occur in the reality. E.g. we generated fully random graphs, but the real world systems shows power law degree distribution [118].

5.4 Summary

In this chapter we described metrics which can be used to evaluate the effectiveness of finite-state transducer inference algorithms. To use the metrics for quantitative evaluation we had to create a larger set of examples. We developed a program to generate example transducers based on random connected graphs, random input examples and mutations for the generated transducers.

We evaluated our algorithms qualitatively and quantitatively. Qualitative evaluation was done by the inspection of the three manually created examples. The quantitative evaluation was carried out by the execution of the two algorithms over the generated examples. The result of our evaluation was that we selected the GABI algorithm to be used in the system which infers M2T transformation modification by examples.

Chapter 6

Semi-Automated Correction of M2T Transformations

In this chapter we describe a software system to support the development of M2T transformations by providing automatic corrections if the developer introduces minor changes into the result of a transformation. Our by-example approach maps the modifications of the target code back to the M2T transformation program using the control-flow graph of the transformation and the trace information. We use the algorithms described in Chapter 4 for the infernece task. We demonstrate our approach by automatically correcting XSLT transformations which generate SQL scripts from ER models. The structure of this chapter is described at the end of the introductory section.

6.1 Introduction

As we described in Chapter 3, M2T transformations are usually developed in an iterative way where a transformation expert delivers a first version, which is iteratively tested by using sample inputs obtained from future users (i.e. regular software engineers) of the code generator. The automatically derived output is then compared to the expected output, and the M2T transformation is manually corrected by the transformation expert afterwards.

However, industrial practice shows that in many cases, there are only minor differences between the expected output and the derived output (e.g. forgetting a white space or a semicolon) and the corrections of the M2T transformations are almost trivial. Still, as regular software engineers lack expertise in M2T languages, the transformation expert's assistance is repetitively needed to perform such corrections, which thus significantly hinders development.

Our aim in the chapter is to reduce effort needed to develop M2T transformations by (1) allowing regular developers to introduce changes in the result (output) of the M2T transformation, and (2) providing semi-automated correction techniques for the M2T scripts when developers introduced only minor, syntactic changes in the result of the transformation. As a result, expert intervention is only necessitated when more complex problems arise with the M2T transformation. Intuitively, minor modifications are classified as simple (point-like) textual changes of the output which *does not necessitate changes in the control flow of the M2T transformation.* As a consequence, the main contribution of the chapter is to infer minor modifications (changes, additions, deletions) of the M2T transformation script from the modification of the output.

Our core idea is that such minor modifications can be located in the transformation script with high precision, and the trace information (between the transformation script

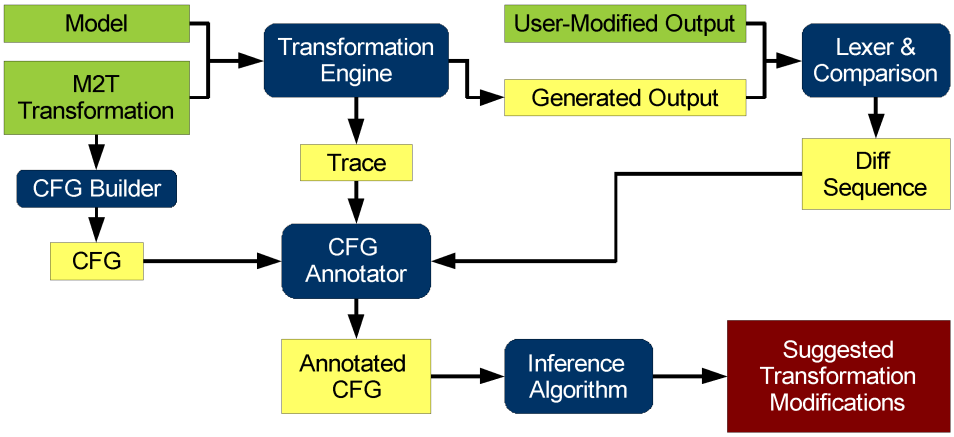


Figure 6.1: System Architecture

and the output stream) can be exploited to map the modification to the control flow graph of the M2T transformation. Our assumptions also help to separate the responsibilities of a transformation expert and our automation technique as the latter will leave the structure of an existing M2T transformation unaltered.

The remainder of this chapter is structured as follows: In Section 6.2 we provide an overview of our approach. Then in Section 6.3 we describe our demonstrative example and according to this example we describe each step in detail. The example is reused from Chapter 4, but in this chapter we explain in full detail the system which is used to pre- and post-process information for our algorithm.

6.2 Overview

In this section we give an overview of our system which is capable to automatically infer minor modifications of the transformation. In Figure 6.1 we depict the architecture of the system. The rectangles represent the processed or produced artifacts: the green (gray) ones are the inputs, the yellow (light gray) ones are the intermediate data, and the red (dark gray) one is the result. The rounded rectangles represent the components of the system.

To implement such a system we assume that the selected M2T transformation language makes it possible to extract the control flow graph of the M2T transformation statically. The interpreter or the compiler is also expected to provide some way to record the trace of the execution. For the output language specification we need only the regular grammars of the lexer to tokenize the output. This generic approach will be exemplified using XSLT [10] as the M2T transformation language and T-SQL [35] as the output language. The exemplified system takes three inputs: the *model*, the *M2T transformation* and the *modified output* (i.e. the modifications of the *generated output*) and produces a list of suggested transformation modifications.

1. The control flow graph (*CFG*) of the *M2T transformation* is first built by the *CFG builder*.

2. Now the *transformation engine* executes the transformation to generate the *output* and to save the *trace* (the sequence of the executed instructions of the M2T transformation).

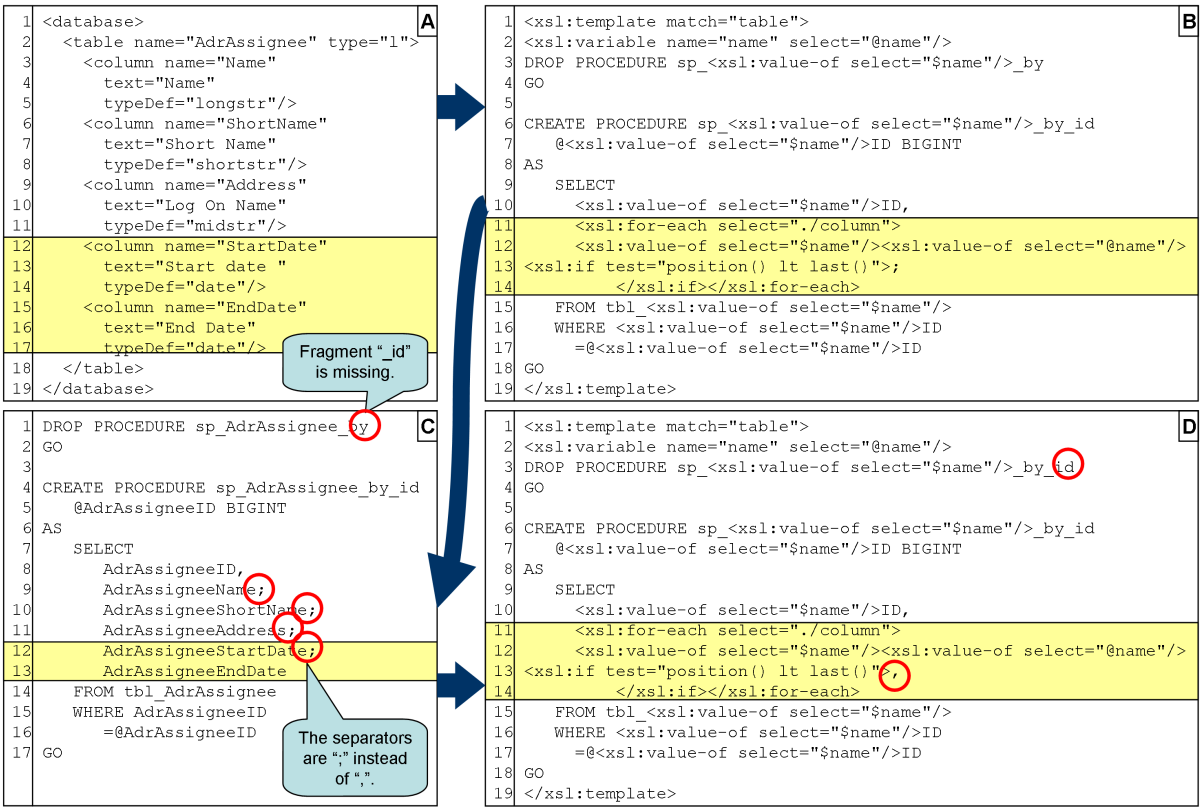


Figure 6.2: Example XSLT correction

3. After the user modifies the output, both the *generated* and the *user-modified outputs* are tokenized by the *lexer* to achieve a meaningful granularity for the comparison. The output *comparison* is done by a standard algorithm to produce a sequence of the tokens in which the differences are marked (*diff sequence* in short). The additions to the generated output and deletions from the generated output can be mapped to the CFG with the help of the trace.

4. The CFG is abstracted to an *annotated CFG*, which contains only the output instructions and the list of modifications at all nodes and edges.

5. The inference algorithm processes the lists of modifications stored in the annotated CFG and proposes a list of *transformation modifications* if the modifications are consistent. If the modifications are contradictory, the algorithm displays the modifications which cause the contradiction.

6.3 The Auto-Correction Process Step-by-Step

To illustrate the basic idea, Figure 6.2 shows (A) a sample XML fragment of a database table description which is transformed into T-SQL code (B) by an XSLT transformation (C). This example is a simplified fragment of the code developed in an industrial project [66, 65]. 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 whose 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).

In this section we describe in detail how our system infers the transformation modification. We describe the components in the following order:

1. the CFG builder extracts the CFG from the M2T Transformation (Section 6.3.1);
2. the transformation engine executes the M2T Transformation to produce the generated output and trace (Section 6.3.2);
3. the lexer processes the generated and the user-modified outputs and creates the diff sequence by the comparison of the token sequences (Section 6.3.3);
4. the CFG annotator pre-processes the CFG by creating an abstract CFG and the trace is modified according to this abstraction (Section 6.3.4);
5. the CFG annotator annotates the abstract CFG by assigning modification tables to the nodes and edges according to the trace and the difference sequence (Section 6.3.5);
6. the inference algorithm builds the list of possible transformation modifications and prepares the suggested transformation modifications from the possible ones (Section 6.3.6);

The last two steps are the application of the GABI algorithm of which description we repeat here in compact form for make the understanding easier.

6.3.1 Extraction of the Control Flow Graph

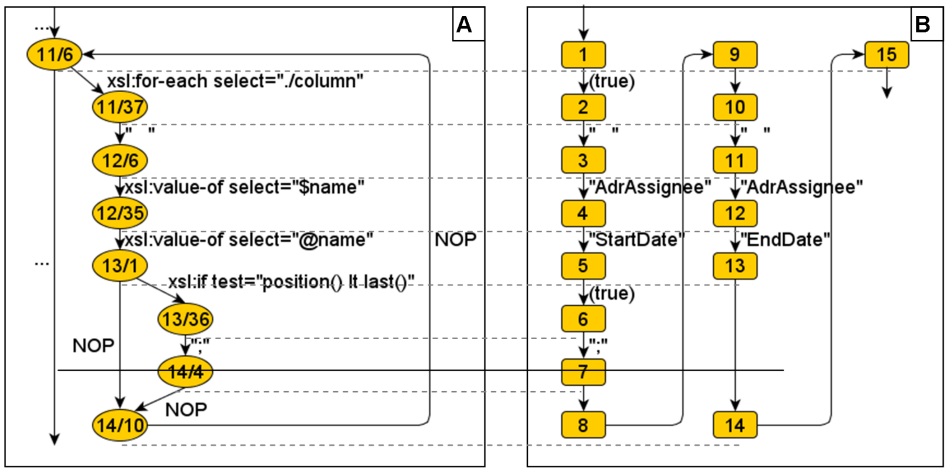


Figure 6.3: Control Flow Graph of the XSLT

First, the Control Flow Graph (CFG) is extracted from the transformation program. In Figure 6.3/A the CFG of the part of the example (Figure 6.2/B from line 11 to 14) is shown. The nodes are locations in the source code (line/column numbers of the first character of the following instruction) and the edges represent instructions. The source node, the target node and the instruction itself are stored in the CFG. Control flow related instructions are represented by multiple edges which are marked by *no operation (NOP)* labels. The CFG has to be derived in a similar way as the transformation engine executes the script, otherwise it cannot be correlated with the trace. A *for-each* loop

and an *if* instruction can be seen in Figure 6.3/A. The *for-each* loop contains three print instructions before the *if* instruction: the first one prints the " " constant, the second and third ones print variables. The *if* instruction contains a print instruction too.

We implemented the extraction of the CFG of the XSLT's in Java; the results are stored in a GraphML file for further processing.

6.3.2 The Transformation and the Recording of the Trace

As a next step, the M2T transformation is executed on the model to generate the output. The transformation is executed by an existing M2T engine (e.g. XSLT engine). We assume that (1) the transformation is free from run-time errors and (2) the output can be tokenized by the lexer of the output language (but not necessarily parsed by the grammar of the output language). In this step, an execution trace is also recorded containing the location of the executed instructions, the instructions themselves, and the produced output (if any).

The part of such a trace which corresponds to the example shown in Figure 6.2/B (from line 11 to line 14) is as follows:

```
(11/6, 11/37, xsl:for-each, -), (11/37, 12/6, constant, " "),
(12/6, 12/35, xsl:value-of select="\$name", "AdrAssignee"),
(12/35, 13/1, xsl:value-of select="@name", "StartDate"),
(13/1, 13/36, xsl:if, -), (13/36, 14/4, constan, ","),
(14/4, 14/10, NOP, -), (14/10, 11/6, NOP, -),
(11/6, 11/37, xsl:for-each, -), (11/37, 12/6, constant, " "),
(12/6, 12/35, xsl:value-of select="\$name", "AdrAssignee"),
(12/35, 13/1, xsl:value-of select="@name", "EndDate"),
(13/1, 14/10, NOP, -), (14/10, 11/6, NOP, -), ...
```

The trace fragment shows how the elements in Figure 6.3/A from line 12 to line 17 are processed, and it can be seen in Figure 6.3/C from line 12 to line 13 that output is generated. The first tuple in the fragment of the trace represents an *xsl:for-each* instruction (located in Line 11 Columns 11 to 37), which produces no output. The second tuple in the trace fragment represents a constant print instruction (Line 11 Column 37 to Line 12 Column 6), which emits " " (actually produces "/n ", but we represent this output with three space in the rest of the examples to save space). This trace is represented graphically in Figure 6.3/B. The nodes are shown only to make it easier to understand the correspondence to the CFG represented in part A of the figure. The actual trace is the sequence of the edges labelled by the produced strings or the results of expression evaluations.

Modern compilers/interpreters typically provide some means of the interface to capture the trace. The Saxon XSLT engine used by our system provides a call-back mechanism through the TraceListener interface. Our Java program records the trace into an XML file.

6.3.3 Tokenizing and Comparing the Text Outputs

After the modification of the generated output by the user both the generated and the user-modified output are tokenized and compared. Instead of a naive approach based on character-wise comparison of the two outputs, we compare the tokens in accordance with the lexical syntax of the target language. For the token sequence comparison algorithm, a slightly modified version of the Ratcliff/Obershelp pattern recognition algorithm [101] is used (as provided by the standard Python library). As a result, a sequence of tokens is created where each token is marked either as unchanged, to be added, or to be deleted.

Note, however that the whitespace and comment elements of the output language are also kept and tokenized.

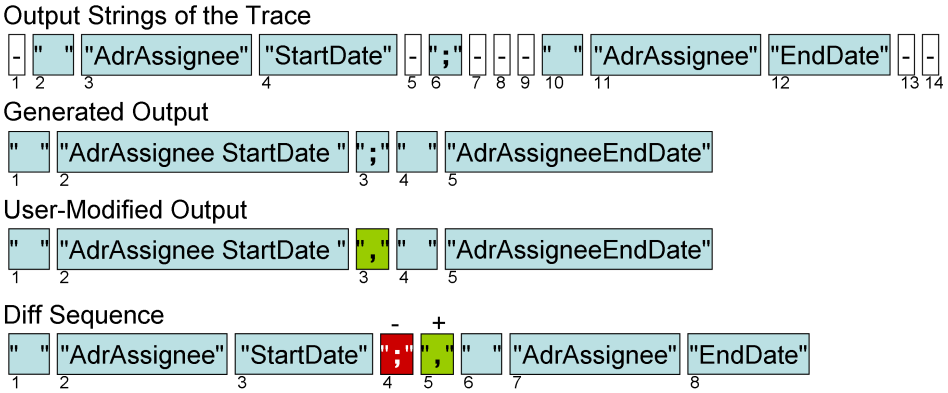


Figure 6.4: Fragments of the Trace, the Outputs, and the Diff Sequence

In Figure 6.4 the output fragment (Figure 6.3/B) can be seen in various processing phases. The first line shows output strings of the trace. The second and the third lines represent the tokens of the generated output and the user-modified output, respectively. The differences between the output strings of the trace and the tokens of the generated output are: the trace contains dummy elements representing no output "-" and they are tokenized in a different way. The last line shows the *diff sequence* in which the addition and removals are marked with + and -, respectively. The *diff sequence* is the result of the token comparison of the generated and modified outputs after some post-processing. In the example, the post-processing means that the "AdrAssigneeStartDate" ("AdrAssigneeEndDate") token is sliced in two tokens "AdrAssignee" and "StartDate" ("AdrAssignee" and "EndDate") according to the tokenization in the trace.

These outputs ideally can be tokenized in such a way that each token corresponds to a single trace element. In practice, it can happen that multiple instructions produce a single token. In this case we slice the token into multiple parts according to the output strings of the trace, if this is possible.

This component is implemented in Python and the lexer is generated by ANTLR. By using ANTLR, re-targeting our tool to different output languages is rather easy: the new rules of the lexer have to be specified and the old lexer can be replaced by the newly generated one.

6.3.4 Preprocessing of the Control Flow Graph and the Trace

To prepare the annotation process, an abstraction of the CFG is created. The edges of the CFG represent instructions, which can be divided into two groups according to the fact whether they produce output or not. The constant string output and the value-of instructions (printing the value of an expression) are output instructions. The control instructions (if, for-each, NOP, etc.) and variable assignment instructions are non-output instructions. The abstract CFG is created from the CFG by merging the nodes connected by edges representing non-output instructions, thus we basically remove instructions which produce no output.

Figure 6.5/A shows the original CFG and Figure 6.5/B shows the abstract CFG. The mapping between the two representations is recorded, and marked with gray arrows. For instance, node 12/6 of the original CFG is mapped to node 2 in the abstract CFG as both the incoming and the outgoing instructions print a textual output (a whitespace

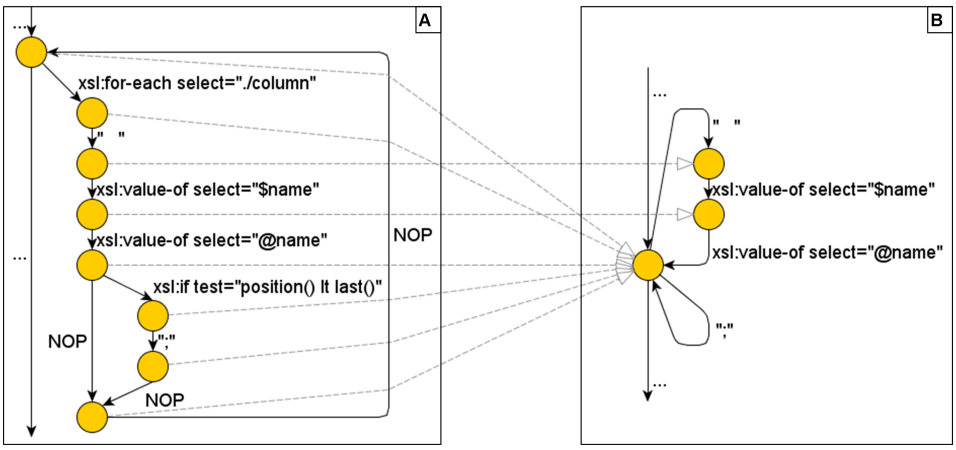


Figure 6.5: Abstraction of the Control Flow Graph

and the value of variable *name*, respectively). On the other hand, nodes 11/6, 11/37 13/1, 13/36, 14/1 and 14/10 are merged to Node 1. From this example one can observe that the abstract CFG represents the relevant information for locating the position of the addition/removal in a more compact way than the CFG does.

This abstraction can be carried out as the location of the additions and deletions can be identified in the context of the output instructions. To demonstrate this principle, imagine that a print of a constant expression is inserted before or after the edge between 13/1 and 14/10 nodes (else branch of the xsl:if expression). If the trace contains no information from the exection of the *if* branch, no change in the behavior with respect to the output of the program can be observed between the two variants.

This component is implemented in Python; its NetworkX package is used to store and manipulate graphs. We had to enhance the capability of the GraphML reader of NetworkX to load files stored by the other modules.

6.3.5 Annotating the Control Flow Graph with the Modification

The CFG annotator converts the information from the *diff sequence* into the difference tables of the edges and nodes of the abstract CFG with respect to a specific trace (code generation run). This annotation marks if a certain abstract CFG node or edge is represented in the trace or not. If it is represented then we also mark whether it is left unmodified, or new tokens are inserted, or existing ones are removed.

The output instruction which produced a specific token can be identified by simultaneously processing the *diff sequence* and the trace itself. As soon as the instruction which produced (or removed) the token is known, the position of the token can be located in the abstract CFG.

In Figure 6.6 the annotation tables for Node 1 are shown, which is generated from the trace of Figure 6.2/B (together with an extract of the abstract CFG node itself). The node and the edges are labeled the same way as can be seen in Figure 6.5/B; the XSLT instructions (edges) are additionally marked with letters. The trace elements which traverse Node 1 are represented by (numbered) gray arrows. The connection between the figures can be understood as follows (each line startes with the number of the trace in Figure 6.6, followed by the corresponding element numbers in Figure 6.4/Diff Sequence, and closed by the informal description of the trace elements):

- 1 (token 1): Node 1 is passed for the first time, when the control flow reaches the

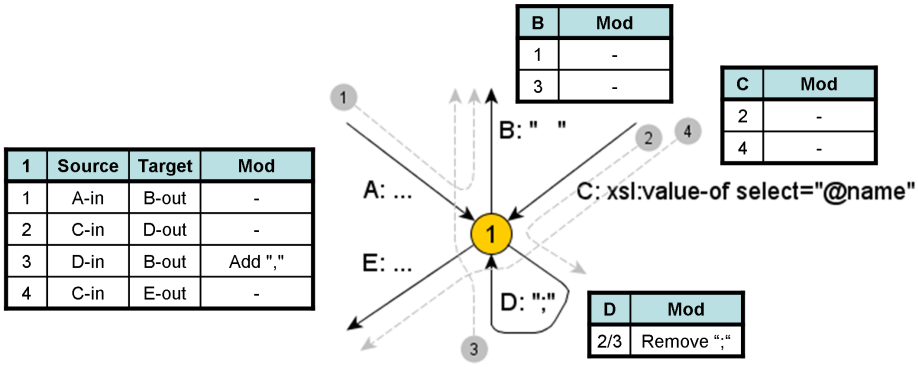


Figure 6.6: A Node of the Abstract CFG with Traces and Difference Tables

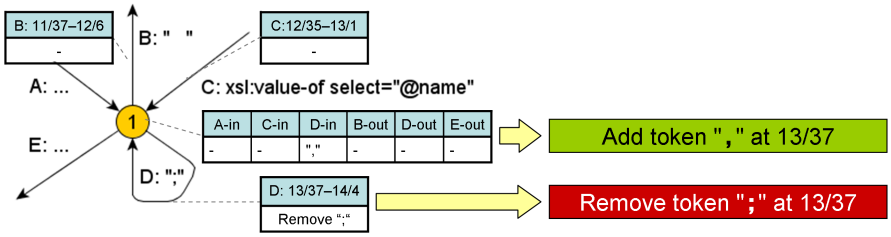


Figure 6.7: Inferring Modification from the Annotated CFG

fragment (A) and then a whitespace " " is printed (B);

- 2 (tokens 3-4): Node 1 is passed for the second time, when the value of the `@name` attribute is printed (C) and then a semicolon ";" is printed (D);
- 3 (tokens 4-5-6): Node 1 is passed for the third time, when the ";" is printed (D) and then a whitespace " " is printed (B);
- 4 (token 8): Node 1 is passed for the last time, when the value of the `@name` attribute is printed (C) and the control flow leaves the fragment (E) .

The elements 4 and 5 of the *diff sequence* represent differences between the generated output and the user-modified output. The element 4 is stored in the edge table of the Edge D and the element 5 is stored in the node table of Node 1.

6.3.6 Inferring the Modifications

In this step the possible transformation modifications are inferred from the difference tables in the annotated CFG with the help of the transformation algorithm described in Section 4.4. Then the edit script which updates the M2T transformation is assembled according to the possible modifications. In Figure 6.7 the inferred modifications and their conversion to edit script lines can be seen. The figure represents the same fragment which can be seen in Figure 6.6. The tables belonging to the edges represent whether the edges have to be left unchanged (*B*, *C*) or have to be modified (*D*). The table belonging to the Node *1* shows that a new print token instruction has to be inserted at the end of Edge *D*.

As a result, our error correction technique is automated in case of consistent minor modifications, and it becomes semi-automated when conflict resolution is needed by the designer.

The suggested transformation modifications (edit script) of the described example are the following: add token "select_by_id" at 3/48, remove token "select_by" at 3/48,

add token "," at 13/37, and remove token ";" at 13/37. The mapping from the possible modifications table to the suggested transformation modifications of the last two elements can be seen in Figure 6.7.

6.3.7 Implementation Issues

A proof-of-concept prototype was also implemented to carry out the automatic correction of XSLT transformations in case of minor modifications. We implemented our system in Java and Python. We have chosen Saxon as XSLT Transformation Engine. The use of Java was practical to interface our system to Saxon. The Java and Python components communicated through XML files.

6.4 Summary

In this chapter we described a system which was aimed in the introduction i.e. a supporting tool for M2T transformation modification by example. Our implementation is capable to sucessfully infer the result of our motivational example. We carried out the initial experiments using the context of a previous industrial project described in Chapter 3, but a more thorough experimental evaluation is an ongoing work.

Chapter 7

Conclusion

We have described the development of a system which modifies a transformation semi-automatically according to presented examples. We have produced several novel results in the process of carrying through our research.

Main Contributions The main contributions of this thesis are the detailed description of the two inference algorithms which can infer modifications of transducers from example input/expected output pairs.

- The first algorithm is called *Direct Trace Modification Based Inference (DTMBI)*, which starts with the modification of the trace information according to the new expected input/output; then it attempts to correct the original transducer to become consistent with the modified trace information.
- The second algorithm is called *Graph Annotation Based Inference (GABI)* which records the expected modification as an annotation of the graph representation of the original transducer. If all of the expected modifications are recorded, it selects one of the consistent combinations of the modifications.

We evaluated the algorithms qualitatively and quantitatively; the results demonstrated that the GABI algorithm behaved significantly better than the DTBMI algorithm. Furthermore, the results of the inference from the automatically generated examples show that in case of larger examples the GABI algorithm can be applied in a much better.

Theoretical Contributions We investigated the theoretical background of model transformation modification by example.

- We modeled M2T transformations with transducers. This idea opens the possibility for the researchers to adapt several other techniques from the area of formal languages.
- We also reformulated the transducer inference problem as transducer modification problem. This may be a feasible approach to produce results by requiring an approximative solution in the case when from the scratch approaches are not applicable effectively.

We adapted and proposed various metrics for evaluating the effectiveness of transducer modifications algorithms.

- We adapted white-box and black-box metrics to describe the structural and behavioral properties of the transducer modifications, respectively. We also adapted coverage metrics to express the quality of the examples used for specifying the transducer modifications with respect to the structure of the transducers.

- We proposed a novel metric to express the quality of the examples with respect to the user's modification intention.

We evaluated the algorithms by computing the proposed metrics on the inference results of two large generated example sets. Conclusions, drawn with the help of our metrics used in this quantitative evaluation, can be correlated with the result of our qualitative evaluation carried out on four hand-crafted examples.

Practical Contributions These algorithms are made applicable to industrial practitioners by developing a system which includes one of the algorithms as its inference engine:

- Our system is capable of semi-automatically infering minor modifications of M2T transformations from modifications of the derived output.

Our motivational example is an XSLT transformation to generate a T-SQL script from an XML data model whose minor errors were corrected. The described implementation is capable to correct minor errors similar to our motivational example; it also can be used to infer transformation modifications corresponding to local refactoring. Our system can provide valuable help in cases when inference is not possible, only by aiding to locate parts of the transformation corresponds to a specific output.

Our other contribution is the description of a lightweight MDD process and a supporting XML-based technology which were successfully applied in industrial projects.

- We provided a precise description of the roles, the artifacts, the activities of our process and the process itself. The reference tool chain and best practices of the application of our process is also documented.

The process fits well into the existing methodologies; we provide a comprehensive description of the minor details which need to be understood for the real world projects. Our approach is not only usable for small and medium size projects, but it can be also used as preparatory step before the introduction of a heavyweight approach.

By the two practical contributions, we fulfilled our original goals. We did not expect at the beginning of our research that we had to have developed new algorithms to solve the problems.

Future Research We are going to evaluate our system on real industrial projects. To do so, the current system needs some additional heuristics and glue logic to integrate with some development system like Eclipse. The system itself can be also enhanced with several minor functionalities: involving the user interactively in the inference process by asking for further information or letting the user choose from alternative solutions; furthermore, the components of the system can be fine-tuned e.g. by testing different comparison algorithms.

Our ongoing research also includes theoretical investigations to study the transducer modification problem (which are essentially different classes of automata with output). We believe that the approaches and algorithms which are developed in the frame of this

research are also useful or inspiring in other research areas e.g. the automatic inference of simple transformations.

Finally, we found a completely new research topic: the human-computer interaction aspects of such modifications by example problems i.e. which solution is preferred by the user of an algorithm in the case of several possible outcomes. This problem is also closely related to the topics of what kind of test cases have to be presented to validate a system.

Bibliography

[1] AndroMDA 3.2. http://www.andromda.org.

[2] Borland Together 2007. http://www.borland.com/us/products/together/index.html.

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

[4] Eclipse Modeling Project. http://www.eclipse.org/modeling.

[5] *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2nd edition.

[6] Iron Speed Designer. http://www.ironspeed.com.

[7] JET: Java Emitter Templates. http://www.eclipse.org/modeling/m2t/?project=jet.

[8] Object Management Group. http://www.omg.org/.

[9] Xpand: a statically-typed template language. http://www.eclipse.org/modeling/m2t/?project=xpand.

[10] *XSLT 2.0 Programmer's Reference*. 3rd edition.

[11] *Enterprise Solution Patterns Using Microsoft .Net: Version 2.0 : Patterns & Practices*. Microsoft Press, 2003.

[12] *ISO/IEC 19757-3 First edition, Information technology - Document Schema Definition Languages (DSDL) - Part 3: Rule-based validation - Schematron*. ISO, 2006-06-01.

[13] MOF 2.0 / XMI Mapping Specification, v2.1.1. http://www.omg.org/technology/documents/formal/xmi.htm, 2007-12-01.

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

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

[16] Thorsten Arendt, Enrico Biermann, Stefan Jurack, Christian Krause, and Gabriele Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Model Driven Engineering Languages and Systems*, volume 6394 of *LNCS*, pages 121–135. Springer, 2010.

[17] K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing Applications Using Model-Driven Design Environments. *Computer*, 39(2):33–40, 2006.

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

[19] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for Agile Software Development. `http://agilemanifesto.org`, 2001.

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

[21] Cliff Binstock, Dave Peterson, Mitchell Smith, Mike Wooding, Chris Dix, and Chris Galtenberg. *The XML Schema Complete Reference*. Addison-Wesley, 2002.

[22] Paloma Cáceres, Esperanza Marcos, Belén Vela, and Rey Juan. A MDA-Based Approach for Web Information System Development. In *Proc. of Workshop in Software Model Engineering*, 2004.

[23] S.P. Caskey, E. Story, and R. Pieraccini. Interactive grammar inference with finite state transducers. In *Proc. of IEEE Workshop on Automatic Speech Recognition and Understanding, ASRU-03*, pages 572–576, 2003.

[24] Philippe Charles. *A Practical method for Constructing Efficient LALR(k) Parsers with Automatic Error Recovery*. PhD thesis, New York University, 1991.

[25] Kwang Ting Cheng and A. S. Krishnakumar. Automatic Functional Test Generation Using the Extended Finite State Machine Model. In *Proc. of 30th International Conference on Design Automation*, pages 86–91. ACM Press, 1993.

[26] Fatemeh Chitforoush, Maryam Yazdandoost, and Raman Ramsin. Methodology Support for the Model Driven Architecture. In *Proc. of 14th Asia-Pacific Software Engineering Conference, APSEC'07*, pages 454–461. IEEE Computer Society, 2007.

[27] 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.

[28] 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, 2002.

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

[30] Allen Cypher, Daniel C. Halbert, David Kurlander, Henry Lieberman, David Maulsby, Brad A. Myers, and Alan Turransky, editors. *Watch What I Do: Programming by Demonstration*. MIT Press, 1993.

[31] Krzysztof Czarnecki. Overview of Generative Software Development. In *Proc. of Unconventional Programming Paradigms, International Workshop UPP 2004*, volume 3566 of *LNCS*, pages 326–341. Springer, 2005.

[32] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. ACM Press/Addison-Wesley, 2000.

[33] Krzysztof Czarnecki and Simon Helsen. Classification of Model Transformation Approaches. In *Proc. of OOPSLA03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

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

[35] Kalen Delaney. *Inside Microsoft SQL Server 2000*. Microsoft Press, 2000.

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

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

[38] X. Dolques, M. Huchard, C. Nebut, and P. Reitz. Learning Transformation Rules from Transformation Examples: An Approach Based on Relational Concept Analysis. In *Proc. of 14th IEEE International Enterprise Distributed Object Computing Conference Workshops, EDOCW*, pages 27–32, 2010.

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

[40] Alexander Egyed. Instant consistency checking for the UML. In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 381–390, 2006.

[41] Bassem Elkarablieh, Ivan Garcia, Yuk Lai Suen, and Sarfraz Khurshid. Assertion-Based Repair of Complex Data Structures. In *Proc. of 22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 64–73. ACM, 2007.

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

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

[44] Bill Evjen, Scott Hanselman, Devin Rader, Farhan Muhammad, and Srinivasa Sivakumar. *Professional ASP.NET 2.0 Special Edition*. Wrox Press, 2006.

[45] Jean-Rémy Falleri, Marianne Huchard, Mathieu Lafourcade, and Clémentine Nebut. Metamodel matching for automatic model transformation generation. In *Proc. of 11th International Conference of Model Driven Engineering Languages and Systems, MoDELS 2008*, volume 5301 of *LNCS*, pages 326–340. Springer, 2008.

[46] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing, 1998.

[47] Pierre Flener. *Logic Program Synthesis from Incomplete Information*. Kluwer Academic Publishers, 1995.

[48] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2002.

[49] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[50] David S. Frankel. *Model Driven Architecture: Applying MDA to Enterprise Computing*. Wiley, 2003.

[51] Rodrigo García-Carmona, Juan C. Dueñas, Félix Cuadrado, and José Luis Ruiz. Lessons Learned on the Development of an Enterprise Service Management System Using Model-Driven Engineering. In *Software and Data Technologies*, volume 50 of *Communications in Computer and Information Science*, pages 59–68. Springer, 2011.

[52] 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 *Proc. of the 2nd International Conference on Theory and Practice of Model Transformations, ICMT 2009*, volume 5563 of *LNCS*, pages 52–66. Springer, 2009.

[53] 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.

[54] 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.

[55] Anastasius Gavras, Mariano Belaunde, Luís Ferreira Pires, and João Paulo A. Almeida. Towards an MDA-Based Development Methodology. In *Proc. of First European Workshop Software Architecture, EWSA 2004*, volume 3047 of *LNCS*, pages 230–240. Springer, 2004.

[56] Marie-Pierre Gervais. Towards an MDA-Oriented Methodology. In *Proc. of 26th International Computer Software and Applications Conference, Prolonging Software Life: Development and Redevelopment, COMPSAC 2002*, pages 265–270. IEEE Computer Society, 2002.

[57] Jean-Yves Giordano. Grammatical Inference Using Tabu Search. In *Grammatical Interference: Learning Syntax from Sentences*, volume 1147 of *LNCS*, pages 292–300. Springer, 1996. 10.1007/BFb0033363.

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

[59] Jack Greenfield and Keith Short. Software Factories: Assembling Applications with Patterns, Models, Frameworks and Tools. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, pages 16–27. ACM, 2003.

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

[61] Gábor Guta. Results of DTMBI and GABI algoritms on Random Transducer Sets. `http://www.risc.jku.at/people/gguta/res_random_tr.xls`.

[62] Gábor Guta. Finite State Transducer Modification by Examples. Technical Report 10–18, RISC Report Series, University of Linz, Austria, 2010.

[63] Gábor Guta. A Graph Annotaion Based Algorithm for Transducer Modification Inference. Technical Report 11-00, RISC Report Series, University of Linz, Austria, 2011.

[64] Gábor Guta, András Pataricza, Wolfgang Schreiner, and Dániel Varró. Semi-Automated Correction of Model-to-Text Transformations. In *Proc. of International Workshop on Models and Evolutions*, pages 43–52, 2010.

[65] Gábor 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.

[66] Gábor Guta, Barnabás Szász, 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.

[67] Zef Hemel, Lennart C. L. Kats, Danny M. Groenewegen, and Eelco Visser. Code Generation by Model Transformation: a Case Study in Transformation Modularity. *Software and System Modeling*, 9(3):375–402, 2010.

[68] Jack Herrington. *Code Generation in Action*. Manning Publications, 2003.

[69] Tobias Hildenbrand and Axel Korthaus. A Model-Driven Approach to Business Software Engineering. *Computing*, 4:74–79, 2004.

[70] Richard Hubert. *Convergent Architecture: Building Model Driven J2EE Systems with UML*. John Wiley & Sons, 2002.

[71] Wim Janssen, Alexandr Korlyukov, and Jan Van den Bussche. On the Tree-Transformation Power of XSLT. *Acta Inf.*, 43(6):371–393, 2006.

[72] Frédéric Jouault and Ivan Kurtev. Transforming Models with ATL. In Jean-Michel Bruel, editor, *Proc. of Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *LNCS*, pages 128–138. Springer, 2006.

[73] G. Karsai. Why XML is Not Suitable for Semantic Translation. Research note, Nashville, TN, 2000.

[74] Marouane Kessentini, Houari Sahraoui, and Mounir Boukadoum. Model Transformation as an Optimization Problem. In *Model Driven Engineering Languages and Systems*, volume 5301 of *LNCS*, pages 159–173. Springer, 2008.

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

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

[77] Alan S. Koch. *Agile Software Development: Evaluating The Methods For Your Organization*. Artech House Publishers, 2004.

[78] Dimitrios Kolovos, Richard Paige, and Fiona Polack. The Epsilon Transformation Language. In *Theory and Practice of Model Transformations*, volume 5063 of *LNCS*, pages 46–60. Springer, 2008.

[79] Dimitrios S. Kolovos, Richard F. Paige, and Fiona Polack. An Agile and Extensible Code Generation Framework. In *Proc. of 6th International Conference Extreme Programming and Agile Processes in Software Engineering, XP 2005*, volume 3556 of *LNCS*, pages 226–229. Springer, 2005.

[80] P. L. Krapivsky and S. Redner. Organization of Growing Random Networks. *Physical Review E*, 63(6):066123, 2001.

[81] Per Kroll and Philippe Kruchten. *The Rational Unified Process Made Easy: a Practitioner's Guide to the RUP*. Addison-Wesley, 2003.

[82] Charles W. Krueger. Software Reuse. *ACM Comput. Surv.*, 24:131–183, June 1992.

[83] Vinay Kulkarni and Sreedhar Reddy. Introducing MDA in a Large IT Consultancy Organization. pages 419–426, 2006.

[84] C. Larman and V.R. Basili. Iterative and Incremental Developments. A Brief History. *Computer*, 36(6):47–56, 2003.

[85] Craig Larman. *Agile and Iterative Development: A Manager's Guide*. Addison-Wesley Professional, 2003.

[86] Xabier Larrucea, Ana Belen Garca Dez, and Jason Xabier Mansell. Practical Model Driven Development Process. In *Proc. of Second European Workshop on Model Driven Architecture (MDA) with an emphasis on Methodologies and Transformations* , pages 99–108, 2004.

[87] 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.

[88] Jin Li, Dechen Zhan, Lanshun Nie, and Xiaofei Xu. A construction approach of model transformation rules based on rough set theory. In *Enterprise Interoperability*, volume 76 of *Lecture Notes in Business Information Processing*, pages 19–35. Springer Berlin Heidelberg, 2011.

[89] Francisco J. Lucas, Fernando Molina, and Ambrosio Toval. A systematic review of UML model consistency management. *Inf. Softw. Technol.*, 51:1631–1645, December 2009.

[90] Vincent Lussenburg, Tijs van der Storm, Jurgen Vinju, and Jos Warmer. Mod4J: a qualitative case study of model-driven software development. In *Proc. of 13th international conference on Model driven engineering languages and systems: Part II*, pages 346–360. Springer, 2010.

[91] Christopher D. Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, 1999.

[92] Jason Xabier Mansell, Aitor Bediaga, Régis Vogel, and Keith Mantell. A Process Framework for the Successful Adoption of Model Driven Development. In *Proc. of Second European Conference on Model Driven Architecture - Foundations and Applications, ECMDA-FA 2006*, volume 4066 of *LNCS*, pages 90–100. Springer, 2006.

[93] Stephen J. Mellor and Marc J. Balcer. *Executable UML: A Foundation for Model-Driven Architecture*. Addison Wesley, 2002.

[94] Stephen J. Mellor, Kendall Scott, Axel Uhl, and Dirk Weise. *MDA Distilled: Principles of Model-Driven Architecture*. Addison Wesley, 2004.

[95] Tom Mens and Pieter Van Gorp. A Taxonomy of Model Transformation. *Electronic Notes in Theoretical Computer Science*, 152:125–142, 2006.

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

[97] Makoto Murata, Dongwon Lee, Murali Mani, and Kohsuke Kawaguchi. Taxonomy of XML Schema Languages Using Formal Language Theory. *ACM Trans. Inter. Tech.*, 5(4):660–704, 2005.

[98] Glenford J. Myers. *The Art of Software Testing*. Wiley, 2nd edition, 2004.

[99] J. Oncina and P. García. *Identifying Regular Languages in Polynomial Time*. World Scientific Publishing, 1992.

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

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

[102] Joachim Rossberg and Rickard Redler. *Pro Scalable .NET 2.0 Application Designs*. Apress, 2005.

[103] Grzegorz Rozenberg, editor. *Handbook of Graph Gramars and Computing by Graph Transformation: Volume I. Foundations*. World Scientific Publishing, 1997.

[104] Yasubumi Sakakibara and Mitsuhiro Kondo. GA-based Learning of Context-Free Grammars using Tabular Representations. In *Proc. of 16th International Conference on Machine Learning*, ICML '99, pages 354–360. Morgan Kaufmann, 1999.

[105] S. Sarkar. Model Driven Programming Using XSLT: an Approach to Rapid Development of Domain-Specific Program Generators. *www.XML-JOURNAL.com*, pages 42–51, August 2002.

[106] Douglas C. Schmidt. Guest Editor's Introduction: Model-Driven Engineering. *Computer*, 39:25–31, 2006.

[107] S. Schrödl and Stefan Edelkamp. Inferring Flow of Control in Program Synthesis by Example. In *Proc. of 23rd Annual German Conference on Artificial Intelligence*, volume 1701 of *LNCS*, pages 171–182. Springer, 1999.

[108] B. Selic. The Pragmatics of Model-Driven Development. *Software*, 20(5):19–25, Sept.-Oct. 2003.

[109] Bran Selic. Model-Driven Development: Its Essence and Opportunities. *Object and Component-Oriented Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on*, pages 7 pp.–, 2006.

[110] S. Sendall and W. Kozaczynski. Model Transformation: the Heart and Soul of Model-Driven Software Development. *Software*, 20(5):42–45, 2003.

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

[112] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, 2nd edition, 2009.

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

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

[115] Galen S. Swint, Calton Pu, Gueyoung Jung, Wenchang Yan, Younggyun Koh, Qinyi Wu, Charles Consel, Akhil Sahai, and Koichi Moriyama. Clearwater: Extensible, Flexible, Modular Code Generation. In *Proc. of 20th IEEE/ACM international Conference on Automated software engineering, ASE '05*, pages 144–153. ACM, 2005.

[116] Axel Uhl. Model-Driven Development in the Enterprise. *Software*, 25(1):46–49, 2008.

[117] Axel Uhl and Scott W. Ambler. Point/Counterpoint: Model Driven Architecture Is Ready for Prime Time / Agile Model Driven Development Is Good Enough. *Software*, 20(5):70–73, 2003.

[118] S. Valverde and R. V. Solé. Hierarchical Small Worlds in Software Architecture. *arXiv: cond-mat/0307278*, 2003.

[119] Dániel Varró and András Balogh. The Model Transformation Language of the VIATRA2 Framework. *Sci. Comput. Program.*, 68(3):214–234, 2007.

[120] 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.

[121] Ståle Walderhaug, Erlend Stav, and Marius Mikalsen. Experiences from Model-Driven Development of Homecare Services: UML Profiles and Domain Models. In *Models in Software Engineering*, volume 5421 of *LNCS*, pages 199–212. Springer, 2009.

[122] Manuel Wimmer, Michael Strommer, Horst Kargl, and Gerhard Kramler. Towards Model Transformation Generation By-Example. In *Proc. of Hawaii International Conference on System Sciences*, volume 40, 2007.

C U R R I C U L U M V I T A E

Personal data

Name:	**Gábor GUTA**
Date of Birth:	02 May 1980
Nationality:	Hungarian
Current Address:	Kalap utca 12. H-1037 Budapest, HUNGARY
E-mail:	g.guta@computer.org
Mobile phone No.:	+36-30-3747769, +43-699-81563372
Memberships:	IEEE, IEEE Computer Society and John von Neumann Computer Society

Key professional competences

Team working experience, advanced presentation skills, in depth knowledge of the recent software development technologies

Advanced programming skills in Java, Python, C++, C#, C, assembly languages (Microchip PIC16, TI TMS320)

Several years of experience with web application development (ASP.NET 2.0/1.1, JEE 1.5)**, relational database design** (MS SQL Server 2000/MySQL 5.0), **XML technologies** (XML Schema, XSLT) **and Linux server administration** (RHEL/CentOS, Ubuntu Server Edition)

Background in Software QA: QA infrastructure (Subversion, JIRA, Bugzilla, CruiseControl, Jenkins/Hudson,), QA tools (Clover, Checkstyle, FindBugs), software process improvement, code reviews, software testing

Expertise in high-end software verification tools: SRI's PVS, MTI's Alloy, PRISM, SPIN

Experiences

Quality Manager at ChemAxon (http://www.chemaxon.com)
Release management, introducing peer review process, improving testing practices, facilitating the modularization/refactoring initiative of a monolithic software system. (Oct. 2010 – present)

Member of Scientific Staff at Software Competence Center Hagenberg (http://www.scch.at)
QM group: supporting Software Process Improvement activities, evaluating and introducing new development tools. KBT group: developing software (MATLAB, Mathematica, C++) based on machine learning technologies for industrial applications. (Jan. 2007 – Oct. 2010)

Work out of a model driven development process to develop ASP.NET 2.0 based enterprise intranet applications supported with tools and formal methods techniques. Designed and developed a production quality code generation tool in the frame of this project. The tool is in use since middle of September 2006. Reference user: DLK Informatik GmbH (Jul. 2006. - Oct. 2006)

Development of a database integration middleware to integrate a FoxPro based accounting software with a J2EE based web-application. Used languages/technologies: C# (ADO.NET), MySQL 5.0 (triggers, stored procedures), Java (J2EE). Reference installation done by DLK Informatik GmbH (Jun. 2006- Jul. 2006)

Projects at Research Institute for Symbolic Computation (http://www.risc.uni-linz.ac.at)
Implementing algorithms in fields of computer algebra and automated theorem proving (Java). I also did various network administration tasks on the Linux network like software installations and security hardening. (Oct. 2005 – present)

Projects at University of Paderborn
Projects on compiler construction (ELI) and parallel computing (MPI, OpenMP) (2003 Summer Semester)

University of Debrecen - Department of Information Technology (http://it.inf.unideb.hu)
Developing a Java software measurement framework: Framework implemented in Java as an **Eclipse plug-in** using JDT. The tool heavily used XML technologies (**XSLT** and **XQuery**) to process measurement results and SVG and XHTML as export format. (Apr. 2004 – Aug. 2005)
Tutor of "The technology of system development" course – The course provides basic "Enterprise Application" development skills. Small students groups designed (**RUP/XP style methodology, UML**) and implemented applications with GUI and RDBS back-end using Java (**J2EE**) or J# (**.NET**) under my supervision. (Sept. 2004 - Dec. 2004)

Tutor of the "Programming languages 1" course – I taught C programming language and best practices. (Apr. 2005-Jun. 2005)

Developing software for PIC16C-based laboratory instruments in assembly language. (Sept. 2001 - Jun 2002), and **Teaching assistant** (Sept. 2001 - Jun. 2004)

Team leader of the team of the University of Debrecen at Texas Instruments DSP and Analog Challenge 2000 (http://delfin.unideb.hu/~uddspg/index.html)
Implementing a Speedometer for Walking and Running; **hardware design** of TI TMS320F243 DSP based circuit containing flash-memory, sensors, user controls and serial interface; design of the embedded software and development of the critical part in mixed **C and assembly** environment (TI Code Composer Studio). (Dec. 2000 - April 2001)

Education

Sept. 2004 – present PhD studies in Computer Science - RISC (Research Institute for Symbolic Computation), Johannes Kepler University (Linz) and Budapest University of Technology and Economics (earlier also at University of Debrecen)

Sept. 1999 – Jun. 2004 MSc in Computer Science - University of Debrecen
Title of MSc thesis: "Measurement possibilities and development trends of software development"

Feb. 2004 – March 2004 **Technical University of Vienna** (scholarship)
Sept. 2002 – Aug. 2003 **University of Paderborn** (scholarship)

Awards

3rd place and ESA special award on 12th European Union Contest for Young Scientists; Project title: Stimulator with ultra low noise for cellular electrophysiological experiments
Amsterdam, The Netherlands, 2000

Relevant papers and technical reports

Gábor Guta, András Pataricza, Wolfgang Schreiner, Dániel Varró: **Semi-Automated Correction of Model-to-Text Transformations,** International Workshop on Models and Evolution, 13th International Conference on Model Driven Enginering Languages and System, p. 43-52, Oslo, Norway October 3-8, 2010.

Gábor Guta, Wolfgang Schreiner, Dirk Draheim: **A Lightweight MDSD Process Applied in Small Projects**, Software Engineering and Advanced Applications, Euromicro Conference, p. 255-258, Patras, Greece August 27-29 2009.

Gábor Guta: **Towards error-free software**
IT-Business (Hungary), Vol. 4, Issue 38, p. 26-27, Vogel Burda Communications, 19 Sept. 2006.

Gábor Guta, et al**.: Implementing a Speedometer for Walking and Running with TMS320F243 DSP Controller**
TI DSP and Analog Challenge 2000 - Top 20 European project CD-ROM, France, 2001.

Other qualifications

English: fluent (Intermediate State Language Examination Certificate 2001)
German: advanced (Intermediate Oral State Language Examination Certificate 2004)
Driving License (B) 1999

Hobbies

Sailing, Indoor climbing

Budapest, 3rd February 2012

Gábor Guta