DATA TRANSFORM COMPOSITION FOR EFFICIENT INFORMATION INTEGRATION

By

JUNGMIN SHIN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2009

To my husband Heon-Mo Koo, daughter Jimin Koo, mother Whaja Koo,
and father Shangkil Shin for their love and encouragement

ACKNOWLEDGMENTS

Though only my name appears on the cover of this dissertation, the completion of my dissertation was possible with the help and efforts of many people. First, I express my appreciation to my advisor Dr. Joachim Hammer. Through my graduate career at University of Florida, his guidance, support, and patience helped me overcome many crisis situations and complete this dissertation. He often brought me to the threshold of knowledge, and ignited the interest to cross the threshold. He also encouraged me to be an independent thinker with a high research standard. I deeply appreciate to my co-advisor, Professor Herman Lam. He is kindly willing to spend a lot of time and effort in improving my work. Without him, I could not accomplish my work. Thanks also go out to the members of the dissertation committee, Professors Abdelsalam Helal, Markus Schneider, and Paul Avery for their valuable guidance. Professors Abdelsalam Helal tries to hear and understand my hardships and his encouragement helps me to keep my self-esteem. I am grateful to many people on the faculty and staff of the Department of Computer and Information Science and Engineering for all that they taught and supported me in various ways. Finally, and most importantly, I sincerely thank my family who have been a constant source of help, support, and strength during doctoral studies. None of my achievement would have been possible without their love. My very special thanks to my husband for his support upon which the path to completing my Ph.D. was built. I warmly appreciate my parents for their unwavering faith in me as well as unending encouragement and support. I thank my sisters for their love and support. I appreciate parents-in-law for consistent encouragement and support. I really thank to God for giving me my daughter, Jimin who was the source of energy to get through my journey. I love you, Jimin.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

DATA TRANSFORM COMPOSITION FOR EFFICIENT INFORMATION INTEGRATION

By

Jungmin Shin

August 2009

Chair: Joachim Hammer
Cochair: Herman Lam
Major: Computer Engineering

Data transformation resolves heterogeneities between disparate schemas and is indispensable process in many applications where data sharing and exchange happen. Creating a data transform which converts source data to target data is extremely time-consuming and labor-intensive. This dissertation presents the data transform composition problem using a large repository of reusable transforms. Recent work on data transforms have focused on structural data mapping or applying a restricted set of data transforms for composition. In order to do automatic data transform composition with existing transforms, we first design our RDF-based transform meta model including meta data on data a transform. Next, we model the data transform composition problem as a graph search problem and use A* algorithm with our transform meta model based sophisticated distance measures. Our experiment shows that our meta model greatly speeds up searching for complete compositions and provides high precision. Our distance measures enable our search to progress to a complete composition correctly and to reduce exponential search space by pruning. Using our system, users can reduce their efforts in time-consuming and error-prune steps of data transformation process, thereby reduce efforts in information integration that requires in many applications.

CHAPTER 1
INTRODUCTION

Data Transformation is a necessary process in such important areas as data integration, data cleansing, and data warehousing that require data sharing [50, 11]. It resolves heterogeneities of data in disparate sources.

In data integration, when disparate data sources have to be integrated, heterogeneities among data in previously independent data sources must be resolved in order to provide a uniform interface to users. For example, when two sales data sets in different currencies are integrated, revenue in one currency (e.g., the Korean won) must be transformed to the other currency (e.g., the US dollar) in order to provide sales data in one currency. Through the data transformation process, data in Korean wons are converted to data in US dollars.

Data cleaning deals with detecting and removing errors and inconsistencies in data to improve the quality of data. When multiple data sources are integrated, the need for data cleaning increases significantly since the sources often contain redundant data in different representations. In order to provide access to accurate and consistent data, consolidation of different data representations through data transformation and elimination of duplicate information becomes necessary [50].

Data transformation plays an important role in a data warehousing system [54]. In a data warehousing system, a extract-transform-load (ETL) process is required to integrate information. The ETL process consists of extracting data from multiple sources, transforming data, and loading data into the data warehouse. As a part of the ETL process, to create and manage transformations, typical data warehousing architectures require external tools. As a result, most existing ETL tools perform the necessary transformations outside the repository where data is stored.

Generally, the data transformation process undergoes several essential steps [55]. In the first step, called *schema matching*, we find semantic correspondences between

elements in the source and target schemas. In the second step, called *semantic mapping*, we analyze syntactic and semantic differences and find relationships between matching elements. We elaborate the matches to create program logic that resolves the differences and enables automated data translation [18]. Finally, the program logic is translated into an executable one and deployed to an execution environment. The program logic is a *data transform* that converts data in source schema to data fit in target schema.

Users should find schema matching and semantic mapping listed above and research the actual program logic of a transform that solves syntactic and semantic heterogeneities among data. Finding the program logic of a transform requires a lot of trial-and-error and is time-consuming [49, 48]. The schema matchings between elements in two schemas include *1-1 matches* and *complex matches*. A complex match means a combination of attributes in one schema corresponds to a combination in another [18]. Creating mappings for complex matches [7, 28] is more difficult than 1-1 matches where one attribute in a schema is matched to a single attribute in another schema.

In our work, we introduce a *data transform composition* problem that reuses existing transforms in a repository instead of creating a new transform from scratch, thus reducing development time and effort. First, we assume that there is a repository that has a large number of transforms. Those transforms can have been created by users previously or harvested from the Internet (e.g, Web services or Java functions). A transform in a repository was previously used for another purpose and generates output with inputs. There is a semantic mapping between inputs and outputs of a transform illustrated by the inside program logic of the transform. Our data transform composition approach tries to use the semantic mapping of a transform to find schema matches and semantic mappings between two schemas. Using a previously available meaningful single or composite transform, we can find semantic correspondences (i.e., schema matches) automatically between two schemas by reusing input/output mappings of a transform, and at the same time semantic mappings can be generated using the inside program logic of transforms.

Even though we have large reusable transforms in a repository, creating a new transform with manual composition is challenging since a user should search a large transform repository to find a compatible (or "right") transform and needs to understand the semantics of transforms and the relationship among transforms, such as whether the output of a previous transform can be matched to the input of the next transform to connect two transforms in composition. Therefore, we need a way to find a desired transform (i.e., goal transform) from a repository as automatically as possible.

Our goal is to find a *complete composition* that are the same semantic mapping (i.e., program logic) as a user's desired transform. However, we cannot guarantee that transforms in a repository are complete to generate any new transforms. We claim that we can give a guide to a user by showing possible *partial compositions* that can be similar to the user's desired transform. As a result, finding schema matchings and semantic mappings with our data transform composition approach reduces time and labor in data transformation.

Recent work on data transforms has focused on finding a data transform as part of the data mapping or schema matching problem. However, those studies concentrated mostly on structural data mapping or applying a restricted set of transforms in composition. There is much research on Web service composition, but we find that we cannot apply the solutions in Web service composition directly to our problem. As far as we investigated, there has been no effort to find 1-1 or complex matches with data transform composition. In our work, we provide a solution for constructing a user's desired transform by composing multiple transforms in a repository, thereby reducing users' overall efforts to perform data transformation, such as analyzing syntactic and semantic differences among relevant data and creating a program logic of a transform that resolves the differences.

1.1 Motivating Example

We introduce a simple example that motivates our data transform composition approach. Suppose each Starbucks franchise in Korea reports daily sales data back to the

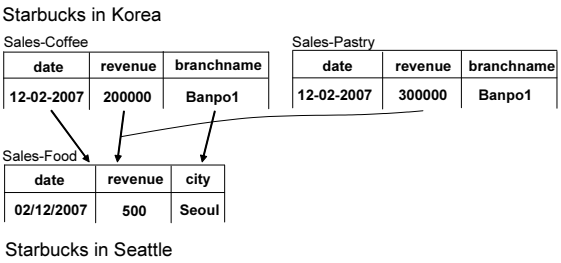

Figure 1-1. The schemas of our example.

Starbucks headquarters in Seattle. The Korean Starbucks operation provides two relations, `Sales-Coffee` and `Sales-Pastry` in Fig. 1-1. `Sales-Coffee` has three attributes, `date`, `revenue`, and `branchname`. The `date` field is in the format of "DD-MM-YYYY", `revenue` is a float value representing revenues from coffee sales in Korean won and `branchname` is the name of a branch in Korea that is a source of the revenue (there can be multiple branches in a city). `Sales-Pastry` has three attributes that are the same as `Sales-Coffee` except `revenue` represents sales from muffins and other snacks.

The Starbucks headquarters in Seattle, however, uses only one relation of the form `Sales-Food` shown in Fig. 1-1 where `date` is in the format "MM/DD/YYYY", `revenue` represents the sum of coffee and pastry revenues in US dollars, and `city` represents the name of the city where a branch is located. The term `city` can be retrieved by looking up a relation storing addresses of branches with `branchname` of `Sales-Coffee`. The IT department of Starbucks is tasked with the job of providing a transform that converts data in the two sales relations (i.e., `Sales-Coffee` and `Sales-Pastry`) from Korean franchises into data in `Sales-Food` that the US corporation can use.

We assume that the IT department has a repository of transforms that can be reused to compose a new data transform. By reusing existing transforms that have been debugged, it simplifies the effort of creating a new transform.

Table 1-1 shows a sample of available transforms in the repository. The input/output of the transforms in Table 1-1 are specified as composite data types. In our example, the IT department of Starbucks wants to create a new transform using one or more available

Table 1-1. Sample transforms in the repository.

Name of Transform	Input Data Type data type (*field*:*type*,...)	Output Data Type data type (*field*:*type*)
Currency-Conversion	Korean won (Korean:float, date1:date)	Dollar (USD:float)
KRW2USD	Korean (KRW:float)	Dollar (USD:float)
Conversion2USD	Currency (amount:float, country:string)	Dollar (USD:float)
AddKRW	TwoKorean (KRW1:float, KRW2:float)	Korean (KRW:float)
Payment-Conversion	Korean won (Korean:float, date1:date)	Dollar (USD:float)
DateFormat2-MMDDYYYY	DATE (date1:date)	DATE (date2:date)
DateFormat2-YYYYMMDD	DATE (date1:date)	DATE (date2:date)
getCity	BranchName (branch:string)	CityName (city:string)

transforms. However, even with reusable transforms, the user must be able to find the appropriate transforms and connect them by correctly matching the output of the current transform to the input of the next transform.

Browsing and finding relevant transforms in a repository is not a trivial task. A user must be able to connect contiguous transforms by correctly matching the output of current transform to the input of the next transform. In Table 1-1, if the user wants to call the *DateFormat2MMDDYYYY* transform first and then call the *CurrencyConversion* transform, the user is not sure whether the output of the DateFormat2MMDDYYYY is matched to the input of the CurrencyConversion transform exactly.

It is clear that the increase of available transforms entails more and more effort to understand existing transforms. As the number of transforms is increased, finding appropriate reusable transforms becomes a time consuming and laborious process. Also,

relationships between transforms for matching the output of a transform to the input of another transform or lack of relationships among transforms need to be identified.

We propose a data transformation composition algorithm that investigates all transforms in a repository on behalf of users and that guides users to create a new transform using existing transforms. The algorithm provides users a sequence of existing transforms that is identical to users' expected transform. If any sequence of existing transforms cannot provide users' expected transform, the algorithm produces similar sequence of transforms to the users' expected transform.

1.2 Challenges and Contribution

In this section, we summarize the challenges in the data transform composition problem and our contributions to the problem.

1.2.1 Challenges

First, we need a formal meta model to represent transforms, such as a user's goal transform, transforms in a repository, and newly created transforms by connecting multiple transforms in search of a goal transform. Existing research on data transformation [5, 17, 21] centers on how to describe, manage, and store data transforms efficiently by providing a tool or language to specify the program logic of a transform and storing the entire executable procedural description. The research provides procedural information rather than structural information about data transform behavior. Procedural information is very helpful for understanding the behavior, but it is inappropriate when comparing two transforms where is necessary to calculate the similarities between two transforms in composition. Existing Web service models are not sufficient to characterize data transforms because they do not have specific data format information and the behavior of an operation is described in one semantic word, which is hard to create when multiple transforms are connected.

In our problem, we need a structural meta model to characterize different data transforms using limited information. Since transforms in a repository can be Web

services, we do not always have the entire program logic of a transform. In addition, our model should be able to be used to calculate how much two transforms are similar to each other to realize that we find out the goal transform in data transform composition.

Second, our problem is to find a data transform that can convert data in the source schema to data in the target schema as automatically as possible using reusable transforms in a repository. It can be solved by a single transform in a transform repository or a composite transform that is composed of multiple single transforms. Since there can be a large number of transforms in a repository, the potential search space for the compositions is huge. Unless we have schema matching between source and target schemas, the search space is increased even more because we need to find a set of attributes in the source schema that matches a specific attribute in the target schema.

In addition, we assume that transforms in a repository can be Web service harvested from the Internet (e.g., looking up the exchange rate between two different currencies). It is difficult to execute transforms and use their output to find compositions because it can take time to execute remote Web services.

Existing research on data transformation find mappings by intensively using outputs generated by executing transforms. Existing research on Web service composition cannot be directly applied to our problem since the Web service models (e.g., WSDL or WSDL-S) [14, 1, 23] are not sufficient to characterize different transforms. We need a novel, sophisticated solution on data transform composition that can efficiently find a goal transform without depending mainly on the data generated by executing transforms.

Third, we cannot guarantee that we always will find a complete composition since our repository may be incomplete. Therefore, it is necessary to provide partial compositions to a user that are useful to construct a goal transform.

1.2.2 Contributions

Taking into consideration the challenges delineated in the previous section, we outline the following contributions toward the solution as presented in this dissertation.

First, we create a Resource Description Framework (RDF) [35] -based transform meta model to represent transforms. Meta data about transforms are represented semantically in RDF triples and those RDF triples can construct structural RDF graphs. We can compare two transforms semantically using those RDF graphs. In addition, an RDF graph can easily be expanded by adding RDF triples when we add more meta data to accommodate new requirements to describe a data transform. Since RDF is widely accepted as a standard for semantic representation of information on the Web and it is represented in an XML, data transforms in our meta model are interchangeable and can be readily compared with other resources on the Web.

Our model includes not only the primitive data type and semantic meaning of a parameter as are provided in existing standards in Web services, but also the data format of a parameter. In addition, meta data on operations describe the relationships between inputs and outputs of a transform. Our experiment shows that our meta model greatly speeds up the search for complete compositions and provides high precision.

Second, we model the data transform composition problem as a graph search problem and use the A* algorithm with our transform meta model-based distance measures. Our sophisticated distance measures enable our search to progress to a correct complete composition and to reduce exponential search space by pruning. In composing transforms, our algorithm preserves the behavior of transforms as much as possible using our transform meta model, thus we can find a goal transform correctly. When there is no complete composition, our algorithm provides partial compositions that are useful to construct a goal transform.

Third, we design and implement our prototype system for semi-automatic data transform composition using transforms in our model in a repository. Our system provides an automatic search of a large repository to find or compose a desired transform. Our semantic annotation tool is used to semi-automatically convert crawled Web services in WSDL to our model. Using our system, users can reduce their efforts in time-consuming

and error-prune schema matching and semantic mapping steps of data transformation process, thereby reducing efforts in information integration that requires in many applications.

This thesis is organized as follows: Chapter 2 reviews related work; Chapter 3 introduces our transform meta model, and Chapter 4 defines the data transform composition problem and shows our algorithm. Chapter 5 describes implementation of our system and Chapter 6 presents our experiments. Chapter 7 provide conclusions.

CHAPTER 2
RELATED WORK

Research related to our work can be categorized into the following areas: Web service composition, data mapping and schema matching, similarity measures, and Morpheus prototype. Before we investigate those areas, we first compare data transform with Web service and workflow.

2.1 Data Transform vs. Web Service and Workflow

Web services are software applications and are provided on top of a service-oriented architecture (SOA). SOA [62, 53] is a technology paradigm characterized by sharing and integrating software components over the Internet without considering platforms [3].
A web service description in WSDL includes only signature information. Web service composition integrate individual Web services to create another new Web service. This is a new trend for creating a new software application using individual software components offered by different service providers.

Workflows [33] model and execute business processes. A workflow can be modeled in XML process definition language (XPDL) [15] and executed in a workflow management system. The XPDL specification covers a broad range of elements that are required in usual business processes. Web services can be used as one implementation type of an activity in a workflow.

A data transform (especially semantic transform) maps attributes in one schema to attributes in another. This mapping converts data from the source data format to target data format. Mostly, data in source and target schemas have different formats but can have the same or similar semantic meanings. For example, in Fig. 1-1, `revenue` in `Sales-Coffee` is represented in Korean won, but `revenue` in `Sales-Food` is represented in US dollars. Both `revenue` attributes share the same semantic meaning.

Considering just signatures and semantic meanings of arguments are not enough to differentiate transforms. Data in different schemas can be expressed in different formats,

like in Korean won or US dollars, while they share the same signatures and semantic meanings. Therefore, existing Web service descriptions in WSDL are not enough to describe a transform because those descriptions include only signatures. Recent work in the semantic Web service area enhanced the descriptions of Web services by adding precondition and postcondition to the signature. We can leverage the semantic approach of Web service description to differentiate data transforms in more detail.

2.2 Data Mapping and Schema Matching

The iMap [18] system semi-automatically identifies both 1-1 and complex matches between database schemas. A complex match specifies that a combination of attributes in one schema corresponds to a combination in the other. The generation of complex matches is done by searching the space of possible matches. A set of search modules, called searchers, are employed and each considers a meaningful subset of the space. For example, a text searcher may consider only matches that are concatenations of text attributes, while a numeric searcher considers combining attributes with arithmetic expressions. Using a beam search, only a pre-specified number of highest-scoring match candidates are selected, and, among candidates, those that have a close semantic distance to the target attribute are selected. To elaborate the search process, domain knowledge and user interaction are used in the process of searching the mapping.

A data mapping problem [47, 64, 7] is about automatic discovery of effective mappings between structured data sources. Data mappings are fundamental in data cleaning, data integration, and semantic integration and include sub problems, such as schema matching and semantic schema mapping. Existing solutions typically have focused on discovering restricted mappings, such as only discovering one-to-one schema matching. There are also structure differences among relations and complex semantic mappings among attributes in different relations. In Tupelo [25], starting from user provided example instances of the source and target schemas, a mapping is semi-automatically discovered by searching within the transformation space based on a set

of mapping operators. Those mappings involve schema matching, dynamic data-metadata restructuring, and complex (many-to-one) semantic functions. One operator is involved in complex semantic transformations, such as date format, weight unit, and international financial conversions. A heuristic function is used for selectively searching the space based on the rankings. Six different heuristics are implemented and evaluated for estimating the distance of a current search state from a target state in terms of number of intermediate search states.

The Piazza [27] project proposed the peer data management system (PDMS), where mapping composition is studied [40] and proposed to serve as one of its main optimization techniques for answering queries efficiently [59]. Yu and Popa [65] applied mapping composition to maintain mappings under some schema evolution scenarios.

2.3 Web Service Composition

Much work has addressed the problem of Web service composition and a wide range of techniques for solving the problem has been used [39, 45, 51, 9, 8, 4, 22, 38]. For example, the work in [43] adapts the A* algorithm using the input/output arguments of Web services. In our work, we also consider the semantic meaning of the arguments and the behavior of transforms. Authors in [39] use resource description framework (RDF) triples for representing pre/post conditions of a Web service. A semantic network can be found among a set of Web services using pre/post conditions.

In the sematic Web service area, a new composite Web service is created using simple Web services with the help of semantic information. Explicit semantics will enable automatic Web service composition without human intervention [46]. Currently, many approaches to semantic Web service composition concentrate on just sematic matching of input/output arguments. However, considering the internal functionalities of services is important since Web service with the same input/output interfaces could have different functions [39].

In the logic-based approach [63, 45], users specify a desired composite application by a first-order formula that represents the logic that must be satisfied by the application. With the assumption that all necessary simple Web services are available, this approach finds a combination of services where conjunctions of logics are equivalent to a formula given by a user. In [4], individual atomic Web services are represented in finite state automata (FSA). Given a set of descriptions of component Web services as an automaton, this approach finds a subset of the component services and a mediator with the input of a desired global behavior specified in an automaton.

In addition, there is a template-based approach [61, 8], but this approach requires technical knowledge and experience for describing desired transforms. Furthermore, we have not seen an approach that uses sematic behavior information (inside of function) for comparing two services for solving service composition problems.

2.4 Calculating Similarity

Unlike a simple keyword search of Web service descriptions in previous Web service search engines, the Web service search algorithm in Woogle [21] exploits the structure of the Web services. The Woogle employs a novel clustering mechanism that groups parameter names into semantically meaningful concepts, and these concepts are leveraged to determine similarity of inputs (or outputs) of Web-service operations. The algorithm depends on only the information provided in the WSDL file without additional annotated information. This approach focuses more on searching similar Web services than on calculating the similarity between two Web services.

Semantic Tools for Web Services, developed by IBM [32], has Web service matching, discovery, and composition features. The Web services are annotated in Web Services Semantics (WSDL-S) [1]. Using the Web Service Interface Matching feature, one can semi-automatically map the interfaces of two given Web services. Domain-independent and domain-specific ontologies are used to compute an overall semantic similarity score between ambiguous terms. This technology resolves semantic ambiguities in the

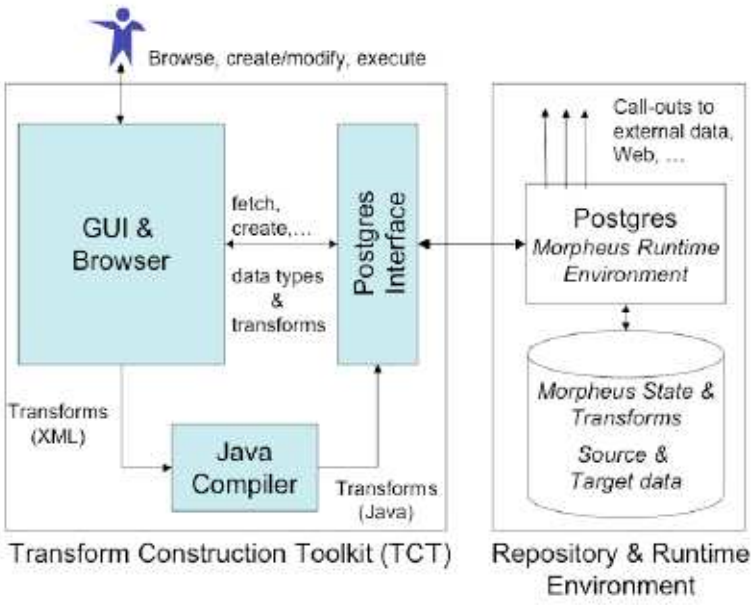

Figure 2-1. Conceptual architecture of the Morpheus system.

descriptions of Web service interfaces by combining information retrieval and semantic Web techniques. Matches from the two approaches are combined to determine an overall similarity score to help assess the quality of a Web service match to a given request [58, 2]. In cases where single services do not match a given request, the system can compose multiple services by employing artificial intelligence (AI) planning algorithms in order to fulfill a given request.

2.5 Morpheus Prototype

The Morpheus system [19, 20] provides an environment for creating, storing, and searching, then executing transforms in order to facilitate the data transformation process.

Fig. 2-1 [20] shows the architecture of the Morpheus system. The Morpheus system consists of two parts: the *transform construction toolkit* (TCT) and the associative repository. The Morpheus system uses the *Postgres* DBMS system [26] as a repository of transforms and at the same time as a platform to run the transforms.

Unlike typical ETL tools that we mentioned in Chapter 1, a data transformation construction tool in the *Morpheus* system facilitates in creating a new transform and the created transforms are stored and executed inside a DBMS. Therefore, the Morpheus

Data types

Create type KoreaRevenue as (date, coffee-revenue, pastry-revenue, branchname)

Create type SeattleRevenue as (date, revenue, city)

Transform

```
StarbucksT (KoreaRevenue D1, SeattleRevenue D2)
{
    D2.date = DateConverter (D1.date)
    D2.revenue = Won2Dolloar(D1.coffee-revenue) +  Won2Dolloar(D1.pastry-revenue)
    D2.city = GetCity(D1.branchname)
}
```

Figure 2-2. Data types and a transform in Morpheus.

system takes advantage of amenities provided by a modern DBMS, such as efficient storage of data and support for transactions and recovery.

The TCT facilitates the creation of a new transform. A user interacts with TCT, which has a browser and GUI for building transforms. Using TCT, the user can create a transform (which we call a *Morpheus Transform*) consisting of Morpheus primitives, namely *Control*, *Wrapper*, *Computation*, *Lookup*, and *Java function*. At the end of the creation, a new transform is first written in XML and translated into a program written in PLjava. The Java program is compiled and registered as a *user defined function* (UDF) in Postgres. A Wrapper primitive is created by wrapping Web services or Web forms available in the Internet. Web services in WSDL [14] are converted to Java functions, then registered as a UDF. A UDF can take simple types, composite types, or a combination of these as arguments. Postgres users can design a new composite type as a *user defined data type* (UDD) and UDDs are registered in the Postgres DBMS. Users can create two composite data types as UDDs, including fields in a source and target schema. In Postgres DBMS, data transformation is achieved by adding source data into the database and then running a query that invokes a transform stored as a UDF. The resulting target data are generated in the database after the execution of the query containing the transform.

Related to the example in Fig. 1-1 in Chapter 1, Fig. 2-2 shows the input and output UDDs (`KoreanRevenue` and `SeattleRevenue`) and one possible description of the `StarbucksT` transform with primitives.

The composite data type `KoreaRevenue` which includes relevant elements (i.e., `date`, `coffee-revenue`, `pastry-revenue`, `branchname`) in the source schema (i.e., the schema of `Starbucks` franchises in Korea in Fig. 1-1.) and `SeattleRevenue`, which includes elements (i.e., `date`, `revenue`, `city`) in the target schema (i.e., the schema of the `Starbucks` headquarters in Seattle) are created. Then, a user creates a new transform `StarbucksT` in Fig. 2-2 which maps the input data type `KoreaRevenue` to output data type `SeattleRevenue`.

Fig. 2-2 shows a simplified description of `StarbucksT`. The date element in `SeattleRevenue` is converted from the format of "DD-MM-YYYY" to "MM/DD/YYYY" through the `DateConverter` function. The revenue element of `SeattleRevenue` is generated by `adding` two values which are converted from `coffee-revenue` and `pastry-revenue` elements of `KoreaRevenue` using `Won2Dollar` function. The `city` element is derived from the `branchname` element using the `GetCity` function. `StarbucksT` is registered as a user-defined function (UDF) in Postgres DBMS and is invoked in an SQL query for actual transformation. For example, Fig. 2-2 shows an SQL query that executes `StarbucksT` transform over the two relations (i.e., `Sales-Coffee`, `Sales-Pastry`) in the source schema. The result of the query execution is data that fit in the target schema `Sales-Food`.

Morpheus transforms in the repository could be used for creating a new transform. A user can view a transform in TCT, then edit the transform to create another transform. Currently, all the steps are performed manually in Morpheus. It has no automatic support for composing a new transform by reusing transforms in the repository.

In Morpheus, a created transform is complied into a Java class file, registered in the Postgres DBMS and invoked on the source data set to be transformed using an SQL query for actual transformation. Currently, the registered transform in UDF is treated as a black-box [13, 12, 31, 30, 60, 42] during query optimization, so there are limitations in

optimizing a query invoking a transform. It would be beneficial to find an opportunity for more optimization by looking inside the transform description.

CHAPTER 3
TRANSFORM META MODEL

Our transform repository can have transforms created from scratch by users or harvested from the Internet using a crawler. We need an abstract model that can reflect various kinds of transforms and has information useful to find a desired transform.

3.1 Definition

In this chapter, we introduce our formal model for representing transforms. A transform is a program that has inputs and outputs. However, unlike general programs, the important role of a (semantic) transform is a data conversion of matching attributes in a source and target schema of disparate data sources. Matching attributes usually share the same semantic meaning, but are represented in different data formats. Therefore, in order to differentiate the function of a transform, we associate to each parameter not only the semantic meaning, as in the semantic Web service area, but also the representation (in other words, data format). In composition, outputs of a previous transform cannot be exactly connected to inputs of the next transform unless they share the same representation. In addition, in our model, we have operations between inputs and outputs.

Existing standards, such as WSDL [14] and WSDL-S [1], are used to represent Web services. In WSDL-S, semantic words defined in the underlying ontology are used to refer to input/output, precondition/effect, and operation. The standard tries to make standards simple and transfer detailed descriptions of semantic meanings to ontology. Our intuition is one semantic word for a parameter (i.e., input or output parameter) is not enough to represent the semantic meaning and representation of the parameter. We can use precondition/effect of WSDL-S, but one precondition per operation is not enough to represent information about multiple parameters. In addition, we claim one semantic word per operation is not enough to reflect the relationships of inputs and outputs.

Hence, as the first step in developing a solution for data transform composition, we formally define a transform [52]. The definition below formalizes our transform meta model.

Definition 1. Transform Meta Model (Morphues-M). A transform T has I, O, and OP. I is a set of input parameters and O is a set of output parameters of T. OP is a set of operations between parameters in I and O. Each parameter $p \in I$ (or $p \in O$) has a tuple including S, D, and R, which means *Semantic Meaning*, *Primitive Data Type*, and *Representation* of p, respectively. S is a word in the *WordNet* [41] dictionary, D is a data type among *float*, *string*, *int*, and *date*, and R is a word in the *format dictionary* we make. Each operation $op \in OP$ is a tuple with OP_I and OP_O. OP_I is a set of inputs and OP_O is a set of outputs of op where $OP\_I \subset I$ and $OP\_O \subset O$.

Example 1. A transform *CurrencyConversion* in Table 1-1 in Chap. 1 converts money in Korean won to money in US dollars with an exchange rate at a given date. The amount of money in Korean won and date in "DD-MM-YYYY" are inputs and the amount of money in US dollars is an output. The transform can be represented in our model as follows:

I = $\{m_1, d_1\}$, O = $\{m_2\}$, OP = $\{op_1\}$,

m_1 = (S, D, R) = ('money', 'float', 'Korean'),

d_1 = (S, D, R) = ('date', 'date', 'dd-mm-yyyy'),

m_2 = (S, D, R) = ('money', 'float', 'USD'),

op1 =(OP_I, OP_O) = ($\{m_1, d_1\}$, $\{m_2\}$),

where m_1 is one of the input parameters of CurrencyConversion and its semantic meaning S is *money*, data type D is *float* and it is represented in *Korean* currency. The d_1 transform is another input parameter of CurrencyConversion and its semantic meaning S is *date*, data type D is *DATE* and it is represented in *DD-MM-YYYY*. The m_2 is an output parameter and its semantic meaning S is *money*, data type D is *float* and it is represented in *US dollars*. This transform has an operation op_1 in which the input set OP_I includes m_1 and d_1 and the output set OP_O includes m_2.

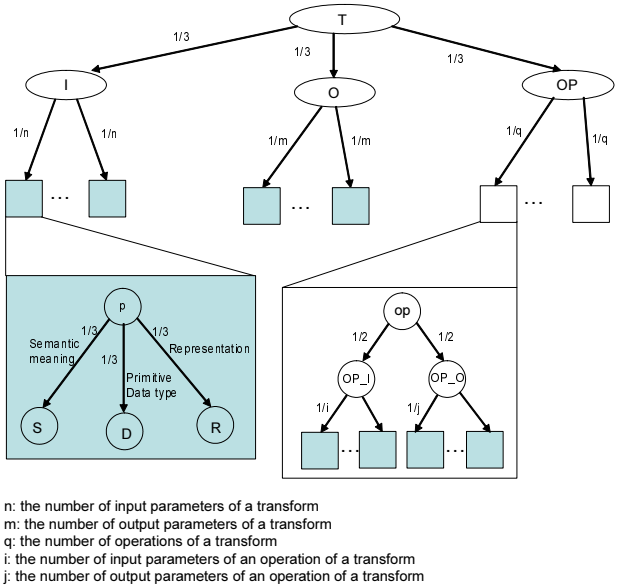

Figure 3-1. A transform represented in graph.

In addition, a transform can be represented as a graph, as shown in Fig. 3-1. We define a transform graph in Definition 2. We use the transform graph for calculating a similarity between two transforms.

Definition 2. Transform Graph. A transform graph, $G(V, E)$ for T is constructed as in Fig. 3-1. The node set V is a union of nodes representing the following: (1) elements in sets I,O, and OP, (2) S,D, R of each element in I, O, and (3) dummy nodes representing T, I, O, OP, OP_I, and OP_O. Each edge $e \in E$ between nodes in V is associated with a weight.

Example 2. Fig. 3-2 represents the CurrencyConversion transform. There are dummy nodes T, I, O, OP, OP_I, and OP_O, which means sets in Definition 1. The input parameter set I has two child nodes p_1 and p_2 and O has one child node p_3. p_1 is a node for the input parameter m_1, therefore p_1 has three child nodes that represent S, D, and R of m_1. An operation set OP has one child node that means an operation op_1 of CurrencyConversion

Our transform meta model can be represented using RDF triples. RDF [52] is widely accepted as a standard for semantic representation of information on the Web and we can use RDF for representing a transform semantically. Meta data about transforms are

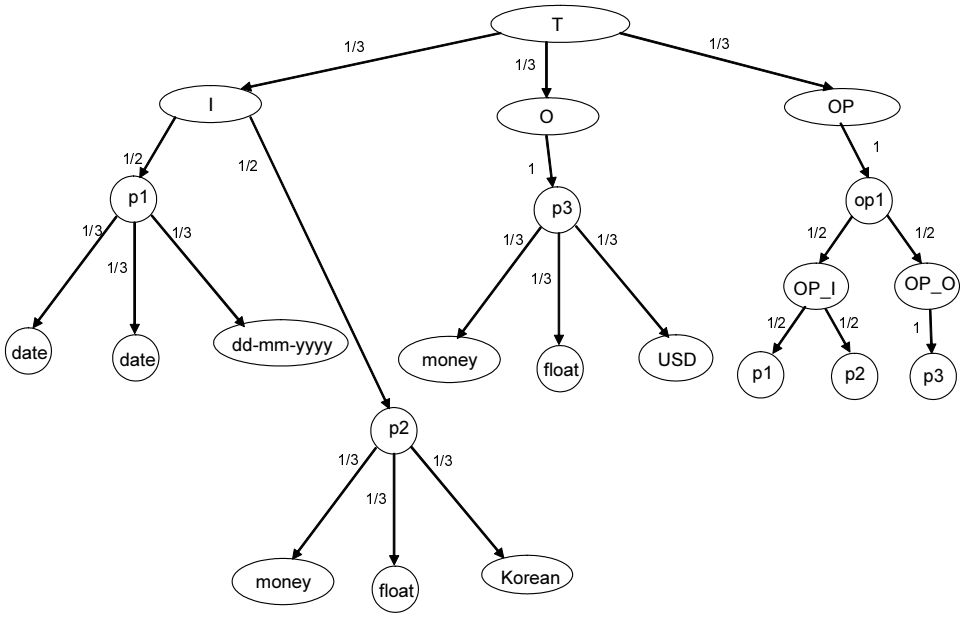

Figure 3-2. CurrencyConversion in transform graph.

represented in RDF triples semantically and those RDF triples can construct structural RDF graphs. We can compare two transforms semantically using RDF graphs. In addition, the RDF graph is easily expandable by adding RDF triples when we add more meta data to accommodate new requirements to describe a data transform. Since RDF is widely accepted as a standard for semantic representation of information on the Web and it is represented in an XML, data transforms in our meta model are interchangeable and can be compared with other resources on Web easily.

3.2 Semantic Similarity of Two Transforms

In this section, we illustrate how we calculate the similarity of two transforms. The corresponding formula is introduced in Sec. 4.2.1.

Example 3. We use a transform graph in order to calculate the similarity between one transform and another transform. Here, we compare the CurrencyConversion transform in Example 1 with *KRW2USD* from Table 1-1. We first show the detailed description about calculating how much KRW2USD is similar to the CurrencyConversion transform. Fig. 3-3 shows a transform graph of the KRW2USD transform where KRW2USD converts

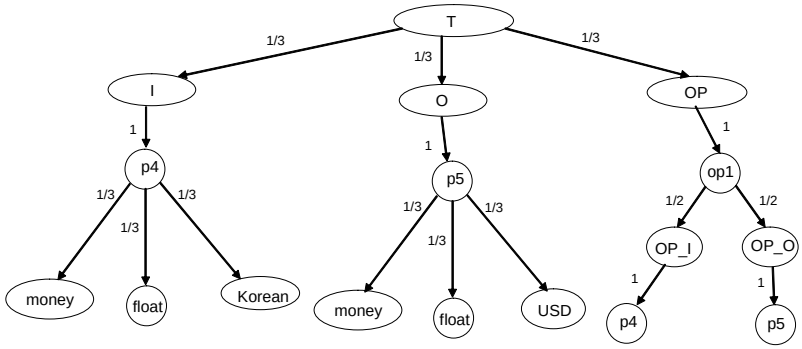

Figure 3-3. KRW2USD in transform graph.

the amount of money in Korean won to US dollars with the exchange rate at the time of execution.

We apply the similarity function introduced in [66] to our transform model. In [66], information available on the Internet is collected and represented in RDF graphs. In order to do a semantic information search, [66] introduces a formula that compares two RDF [35] graphs. Unlike other approaches in information retrieval that are based on term frequency analysis, matching RDF graphs considers the structural information represented in a graph. By using transform graphs, we calculate how much transform B is similar to transform A (i.e., f(A, B)) in terms of how much information in A is covered by B. If B has the same or more information than A, B is the same as A. Therefore, we can say that A and B are exactly the same when both f(A,B) and f(B,A) are 1.

We calculate f(CurrencyConversion, KRW2USD), how much KRW2USD is similar to CurrencyConversion. First, we compare the child nodes of I in the transform graphs in Fig. 3-2 and Fig. 3-3. Each child node has three additional child nodes that hold semantic meaning, data type, and representation (in other words, data format) of an input parameter (see Definition 1.). Let f_1, f_2, and f_3 be functions calculating similarities in terms of semantic meaning, data type, and representation, respectively (these three functions will be explained in detail in Sec. 4.2.1). The similarity of (p2, p4) is calculated

by f_1('money', 'money')* w_1 + f_2('float', 'float')* w_2 + f_3('KRW', 'KRW')* w_3 = 1*1/3 + 1*1/3 + 1*1/3 = 1. (The values w_1, w_2, and w_3 are weights assigned to the edges from node p2 to its child nodes). We find a pair-wise combination of input parameters that can generate the maximum similarity. For example, the (p2, p4) pair has greater similarity than the (p1, p4) pair. At node I of Fig. 3-2, we have 0*1/2 + 1*1/2 = 1/2 as a result of applying weights from node I to its child nodes. Using the same method, node O has 1 and node OP has 3/4. Finally, at node T of Fig. 3-2, we get the result 1/2*1/3 + 1*1/3 + 3/4*1/3 = 3/4, which means the similarity of KRW2USD to CurrencyConversion is 3/4. In short, we compare leaf nodes under parameter nodes in both transform graphs, apply weight values on edges to get value at their parent node, and calculate the final result at root node T.

CHAPTER 4
TRANSFORM COMPOSITION

In this chapter, we introduce our data transform composition problem and solution approach. We use the A* algorithm with similarity measures we designed and a function Merge to combine transforms correctly while we find a desired transform using transforms in a repository.

4.1 Transform Composition Problem

In our problem, we have schema S_1 for the source data and schema S_2 for the target data. S_1 has n attributes and S_2 has m attributes. There can be 1:1 or i:1 (where $1 < \text{i} \leq$ n) matchings between attributes in S_1 and S_2. For each given schema matching between S_1 and S_2, we find a (single or composite) transform using transforms in a repository. When there are k transforms in a repository, there are $k!$ possible combinations for constructing sequences of transforms. The problem generates a large search space, so efficient heuristics are needed to solve a transform composition problem in a reasonable time.

Hence, we model our transform composition problem as a graph search problem and use the A* search algorithm with our heuristic function for finding a user's desired transform. The resulting composite transform cannot be exactly the same as the desired transform because we cannot guarantee that all required transforms for creating a desired transform exist in the repository. Therefore, we suggest partial compositions that are useful to construct the desired transform.

For the rest of this chapter, we use ST and T_d to denote the set of transforms in a repository and a desired transform (i.e., goal transform), respectively. Each $T_i \in ST$ and T_d are represented in our transform meta model in Definition 1.

Definition 3. Desired Transform. A desired transform $T_d = \{d_1, d_2, d_3, ..., d_m\}$, where m is the number of attributes of the target schema S_2. The d_k is a match between j

elements in S_1 and one element in S_2 (where $1 \leq$ k $\leq$ m and $1 \leq$ j $\leq$ n) and is represented as a transform in Definition 1 as follows:

(1) j elements in S_1 and one element in S_2 have S, D, and R, which are introduced in Definition 1.
(2) j elements are described with a clause, which is defined in Definition 4.
(3) A set I of d_k has j elements and a set O of d_k has one element
(4) OP has operations between elements in I and O

Next, we give a definition of a clause that is used to specify I of d_k.

Definition 4. {A}[B]. Both A and B are a set of elements in S_1 and $A \cap B = \emptyset$. The set A includes elements that are necessary to generate an output of d_k and the set B includes elements that might or might not be necessary to generate an output of d_k. We specify these as $d_k.I.A$ or $d_k.I.B$ and $d_k.I.A \cup d_k.I.B = d_k.I$.

Example 4. In our example in Chap. 1, we can describe j elements in S_1 for a match to `revenue` in S_2 as follows.

(1) {coffee_revenue, pastry_revenue, date}[]
(2) {coffee_revenue, pastry_revenue} [date]
(3) {} [coffee_revenue, pastry_revenue, date, branchname]

where coffee_revenue = S_1.revenue and pastry_revenue is another S_1.revenue in S_1.

(1) means there is a schema match between three elements (i.e., S_1.revenue, S_1.revenue, S_1.date) and S_2.revenue. (2) means S_1.revenue, S_1.revenue are certainly required and S_1.date can be necessary to match to S_2.revenue. (3) means it is not certain which one is matched to S_2.revenue, therefore all elements in S_1 are considered. (1) is the case that finds a composite transform with a given schema match and (3) is the case where we do not have a given schema match. With (3), our approach tries to find schema matching and semantic mapping between two data sources.

We define the {A}[B] clause because we try to reduce the burden of users to find the exact schema matchings. With the clause, our approach finds all possible compositions that have all elements in A as input and some of elements in B as input. Therefore, our

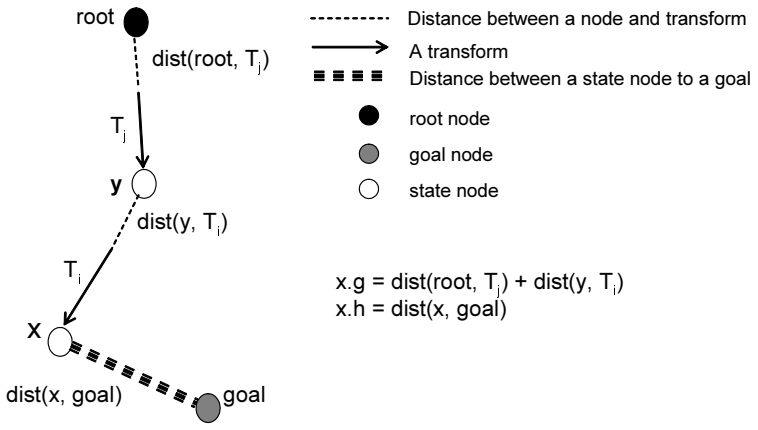

Figure 4-1. A part of transform composition graph.

approach reduces the burden of finding schema matching and semantic mapping in the data transformation process.

Next, we define the transform composition problem.

Definition 5. Transform Composition Problem. With ST and T_d, our goal is (1) find or compose the same transform(s) as T_d within a given amount of time.

In order to solve our transform composition problem, we model our problem as a graph search problem. Definition 6 introduces the transform composition graph.

Definition 6. Transform Composition Graph. A directed acyclic graph, $G'(V', E')$, is constructed as follows: the node set, V', is the union of the root node, goal node, and state nodes. Each directed edge $e \in E'$ between nodes is associated with a transform $T_i \in ST$. Each node x has following information.

- *path*: an ordered list of transform ids that are connected from root to x
- MT: a transform that is generated by merging transforms in a path from root to x
- g: a distance from root to x
- h: a distance from x to goal

In G', the transform composition problem is to find paths from root to goal that generate a transform that is the same as T_d within a given time t. The detailed algorithm is introduced in the next section.

Fig. 4-1 shows a part of a transform composition graph. Two transforms T_j and T_i are connected from root to x through y. There is a distance between a node and the connecting transform, such as dist(root, T_j) and dist(y, T_i). In addition, the distance from x to goal is dist(x, goal). The g of x means the distance from root to x and it is the sum of distances from root to x that are dist(root, T_j) and dist(y, T_i). The h of x means dist(x, goal). MT of x is generated by merging T_j and T_i. We use the A* algorithm to find a transform path from root to goal node.

4.2 Algorithm

We apply the A* algorithm to the transform composition graph G' defined in Definition 6. The A* algorithm is a best-first search algorithm that advances searching from a node that is the most promising to generate a path to the goal node [16]. As shown in Fig. 4-1, each node x in G' has g and h values. The g is the distance from root to x and h is a heuristically estimated distance from x to goal. The A* finds the least-cost path from a given initial node to one goal node out of one or more possible goals. The A* uses f(x) = g(x)+h(x) for expanding nodes in G'. The intuition of using f(x) is that advancing from the node x that has the smallest $f(x)$ value is the fastest way (the shortest in terms of a distance) to find a path to the goal node. The A* incrementally builds all paths leading from the root until it finds one that reaches the goal node, but only builds paths that appear to lead toward the goal node.

Algorithm 1 shows how the A* algorithm finds a composite transform. At the current expanding node x, all transforms in a repository are connected and a new state node for each connection is created with information in Definition 6. The new state node is put in a priority queue and then a state node that has the least f(x) is selected as the next expanding node. In the following sections, we explain our distance measures for calculating g and h of a state node and the *Merge* function for merging transforms while a path is lengthened.

Algorithm 1 A* Search Algorithm

Input: a set of transforms ST and a desired transform T_d in our model
Output: transform paths

for all d_k in T_d **do**
  Initially, E-node x is the root node
  Let $x.path$ = "0", $x.g = 0.0$, $x.h = \infty$, $x.MT$ = null
  Let $OPEN$ = the list of live nodes. Initially, the list is empty.
  loop
    for all transforms T_i in ST **do**
      M = Merge($x.MT$, T_i)
      if M is not empty **then**
        for all transforms m_j in M **do**
          Create a node e
          $e.path = x.path$ + "-" + i + "_" + j
          $e.MT = m_j$
          $e.g = x.g + dist_1(x.MT, T_i) * w_1$
          $e.h = dist_2(d_k, e.MT, I + OP) * w_2 + dist_2(e.MT, d_k, O) * w_3$
          if $d_k.I.A \subset m_j.I$ **then**
            Create $tempd_k$ with $m_j.I$ and $d_k.O$
            $temph = dist_2(tempd_k, e.MT, I + OP) * w_2 + dist_2(e.MT, tempd_k, O) * w_3$
          end if
          if $temph == 0$ and $e.g == 0$ **then**
            `"Find an answer"`
          end if
          if $e.h == 0$ and $e.g == 0$ **then**
            `"Find an answer"`
          else
            Add e to $OPEN$
          end if
        end for
      end if
      if $OPEN$ is empty **then**
        print `"No more E-nodes"`
        return
      end if
      Let x = a node in $OPEN$ that has the least $x.g + x.h$
    end for
  end loop
end for

4.2.1 Similarity Measures

We use two distance measures $dist_1$ and $dist_2$ to calculate f(x). As shown in Fig. 4-1, the node x is created and connected to the node y with the edge associated with T_i. The f(x) is calculated using the following formula.

$$f(x) = g(x) + h(x)$$

$$g(x) = g(y) + dist_1(y, T_i) * w_1$$

$$h(x) = dist_2(d_k, x.MT, I + OP) * w_2 + dist_2(x.MT, d_k, O) * w_3, \quad (4\text{–}1)$$

where w_1, w_2, and w_3 are weight values.

The g(x) is the addition of the g value of the previous node y and the distance between the node y and the subsequent transform T_i, and the latter is calculated by the function $dist_1$. When T_i is connected to y, for a parallel composition, the unmatched inputs of d_k so far in the path from the root to y are also considered as a part of the output of a transform at y in order to match the input of T_i (Sec. 4.2.2 includes a detailed explanation).

Therefore, $newO = y.MT.O \cup (T_d.I - (T_d.I \cap y.MT.I))$, which means a union of $y.MT.O$ (the output of the transform MT at node y) and unmatched inputs of d_k at node y. We make all combinations between elements in $newO$ and $T_i.I$, and find the best combination generating the maximum sum of similarity values (For the connection, all parameters in $T_i.I$ should be matched with any parameter in $newO$). For calculating the similarity between an element in $newO$ and an element in $T_i.I$, we use the following formula:

$$sim_e(u_p, v_q) = (sim_S(u_p.S, v_q.S) + sim_D(u_p.D, v_q.D) + sim_R(u_p.R, v_q.R)) * 1/3., \quad (4\text{–}2)$$

where $u_p \in newO$ and $v_q \in T_i.I$ and sim_e is the sum of three similarities. As in Definition 1, each parameter in $newO$ and $T_i.I$ has S, D, and R. The sim_S compares S of two parameters. We use the *JWordNetSim* library for the comparison (if a similarity value

is less than a threshold, the result becomes zero). The sim_D compares the primitive data type of two parameters and sim_R compares the representation (i.e., data format) of two parameters. The result of sim_S is between 0 and 1, the result of sim_D and sim_R is 0 or 1, and the result of sim_e is between 0 and 1.

Let z = $\mid T_i.I, \mid$ $u_p \in newO$, and $v_q \in T_i.I$, then $dist_1$ is as follows:

$$\begin{aligned} dist_1(y, T_i) &= 1 - sim_1(newO, T_i.I) \\ &= 1 - \frac{\max \sum_{q=1}^{z} sim_e(u_p, v_q)}{z} \end{aligned} \tag{4–3}$$

The g(x) is the sum of g(y) and $dist_1(y, T_i)$. The h(x) is the heuristically estimated distance from the node x to the goal node. At the first time, we design h(x) as the distance between the output of the node x and the output of a desired transform using dist1, but our search cannot find the desired transom well since just output matching is not enough to find a desired transform.

Hence, we design sim_2 to calculate the similarity between the transform at the node x (i.e.,x.MT) and a desired transform d_k. Both transforms are represented in our model, thus can be represented in RDF graphs. The $sim_2(d_k, x.MT, I+O+OP)$ means how much inputs, outputs and operations of d_k is covered by inputs, outputs and operations of x.MT. Once x.MT has the same inputs, outputs and operations as d_k, $sim_2(d_k, x.MT, I+O+OP)$ becomes 1. However, x.MT can have more inputs, outputs, and operations which are not in d_k. Thus, when we use $1 - sim_2(d_k, x.MT, I+O+OP)$ as our h(x), our answer can have many useless inputs, outputs or operations besides the ones related to a desired transform. If both $sim_2(d_k, x.MT, I+O+OP)$ and $sim_2(x.MT, d_k, I+O+OP)$ are 1, d_k and x.MT are exactly the same.

The input of sim_2 is two transforms to compare, and flags (i.e., I, O, and OP), which mean input, output, and operation, respectively. Unlike sim_1, all the input of T_i do not need to be matched. The sim_e is used to compare inputs and outputs, and sim_{op} is defined to compare operations of two transforms.

Let w = | $T_i.I$ |, v = | $T_i.O$ |, z = | $T_i.OP$ | , l = | $e_q.OP\_I$ | , m = | $e_q.OP\_O$ |

$a_q \in T_i.I, b_p \in T_j.I$, $c_q \in T_i.O, d_p \in T_j.O$, $e_q \in T_i.OP, f_p \in T_j.OP$

$g_q \in e_q.OP\_I$, $h_p \in f_p.OP\_I$, $k_q \in e_q.OP\_O$, $l_p \in f_p.OP\_O$,

$$\begin{aligned} sim_2(T_i, T_j, I+O+OP) = & \frac{\max \sum_{q=1}^{w} sim_e(a_q, b_p)}{w} * w_4 \\ & + \frac{\max \sum_{q=1}^{v} sim_e(c_q, d_p)}{v} * w_5 \\ & + \frac{\max \sum_{q=1}^{z} sim_{op}(e_q, f_p)}{z} * w_6 \end{aligned} \tag{4–4}$$

$$\begin{aligned} & sim_{op}(e_q, f_p) = sim_{OP\_I}(e_q, f_p) + sim_{OP\_O}(e_q, f_p) \\ & = \frac{\max \sum_{q=1}^{l} sim_e(g_q, h_p)}{l} + \frac{\max \sum_{q=1}^{m} sim_e(k_q, l_p)}{m} \end{aligned} \tag{4–5}$$

where w_4, w_5, and w_6 are weights of I, O, and OP, respectively.

We can give different weight values to each matching but here we give the same weights. Using sim_2, we can calculate how much a transform constructed from a root to a current node x is similar to d_k in terms of inputs, outputs, or operations. The $dist_2$ is calculated by subtracting sim_2 from 1. The formula for calculating h(x) is appeared in (4-1).

In (4-1), the first part of h(x) means that how much x.MT has the inputs and operations of d_k and the second part means how much x.MT has other outputs besides the output of d_k. Considering the second part makes our search proceeds not to generate unnecessary outputs.

Using $f(x)$ in (4-1), our approach advances the search where has less distance between connecting transforms and has more similar input and operations to d_k but fewer outputs besides the one in d_k.

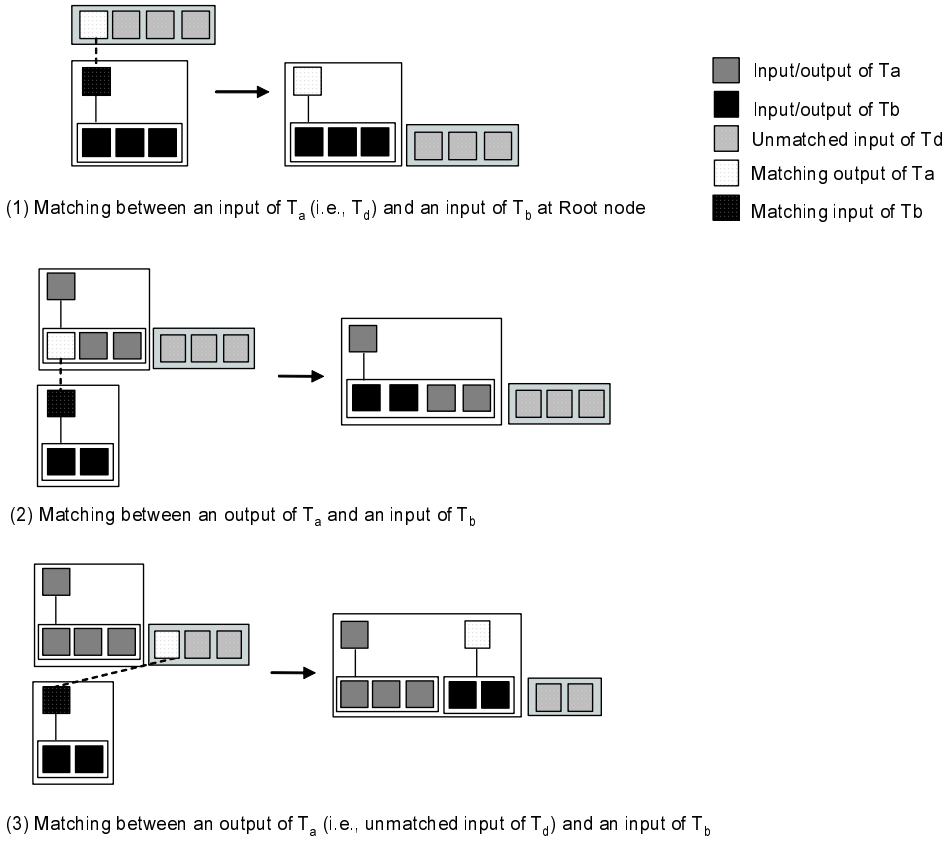

Figure 4-2. Single match between two transforms.

4.2.2 The function Merge

The function $Merge$ combines two transforms and generates a new transform in our model in Definition 1. We need to generate the new transform in order to compare it with the desired transform $d_k \in T_d$ and check whether we have found a complete composition. In this section, we denote T_a for a transform at node N_a, T_b for a connecting transform to N_a, and T_M for a transform created by connecting T_b to T_a. In addition, $d_k.I$ is a set of inputs of d_k, $T_a.O$ is a set of outputs of T_a, and $T_b.I$ is a set of inputs of T_b. As mentioned before, at N_a, we define $newO = T_a.O \cup (d_k.I - (d_k.I \cap T_a.I))$, which means a union of $T_a.O$ and the unmatched input of d_k so far in the path from root to N_a. When T_a and T_b are merged, all parameters in $T_b.I$ should be matched with any parameters in $newO$.

There can be *single match* and *multiple match* between the output of a previous transform ($newO$) and the input of a subsequent transform ($T_b.I$), as in Fig. 4-2 and

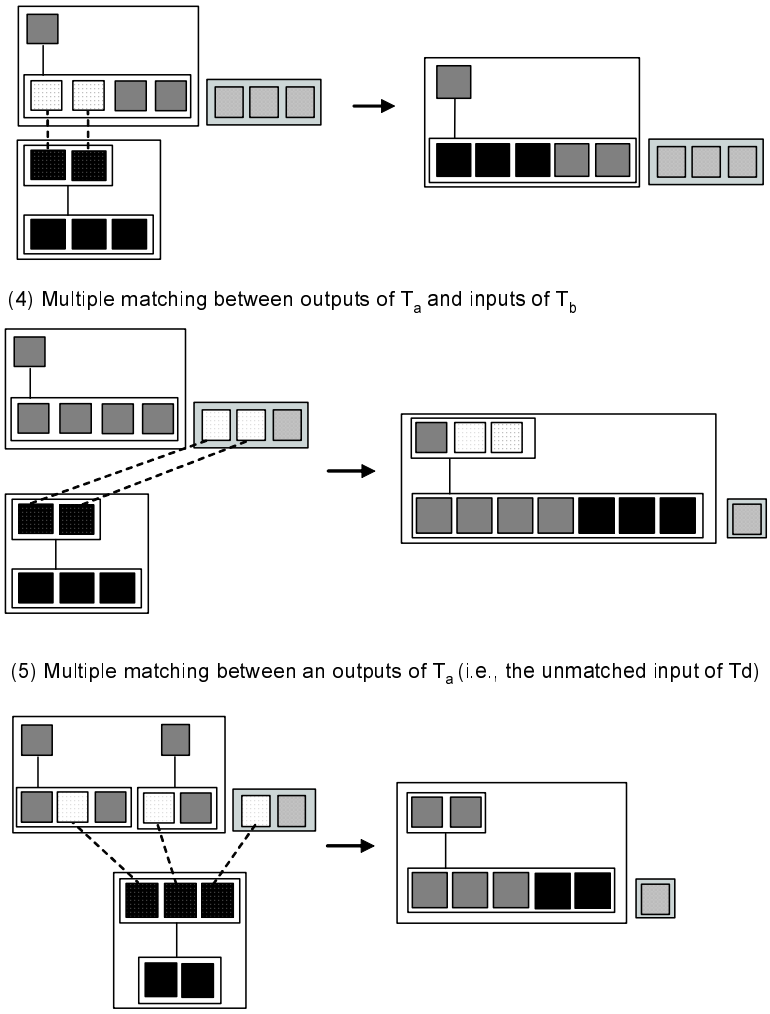

Figure 4-3. Multiple match between two transforms.

Fig. 4-3. The figures include all possible merging scenarios. The function Merge keeps inputs, outputs, and operations that characterize the resulting transform as much as possible and removes useless ones.

At root, $d_k.I$ and $T_b.I$ are compared and matched (with a distance calculated using $dist_1$), as in the case of (1) in Fig. 4-2. $T_M.I$ becomes the parameter in $d_k.I$ that is matched with a parameter in $T_b.I$, and $T_M.O$ becomes $T_b.O$. Finally, a new transform T_M is created like the one on the right side of the arrow in (1). The unmatched parameters in $d_k.I$ remaine in a gray box.

If N_a is not the root node, the parameter in $T_b.I$ (since all parameters in $T_b.I$ should be matched for connection, in single match, there is only one parameter in $T_b.I$) can be matched with the parameter in $T_a.O$ (as in (2) in Fig. 4-2) or the unmatched parameter in $d_k.I$ (as in (3)). By keeping the unmatched input of d_k, we allow not only serial but also parallel connections of transforms. As shown in (2) and (3), the unmatched $T_a.O$ together with $T_b.O$ are included in $T_M.O$, and $T_a.I$ and the matched $d_k.I$ become $T_M.I$. When a parameter in $T_b.I$ is matched with an element in $d_k.I$, an additional operation is added to $T_M.OP$ as in (3) in Fig. 4-2. The Merge function preserves as much as possible the semantic of inside operations of transforms.

In a multiple match, as shown in Fig 4-3, elements in $T_b.I$ can be matched with just the parameters in $T_a.O$ (as in case (4)) or in unmatched $d_k.I$ (as in case (5)) or parameters in both $T_a.O$ and $d_k.I$ (as in case (6)). As shown in (6), two separate operations in T_a can be merged into one operation by a multiple match.

4.3 Partial Composition

If our algorithm cannot find a complete composition, partial compositions are provided. As mentioned in [36, 56, 18], partial compositions can be useful because the user can get an idea about useful single transforms without manually searching the transform repository and can elaborate on partial compositions to make a complete composition.

As in Fig. 4-1, there are root, goal, and state nodes in our transform composition graph. A state node that is created during the search for a complete composition has information, as shown in Sec. 4.1, such as a *path* that is the ordered list of transform ids connected from the root node to this state node, g is the distance from the root to this state node, and h is the distance from this state node to the goal node. We record the path, g, and h of each state node in a file. Therefore we have information about state nodes created during the search for a complete composition.

Those state nodes include *pruned state nodes*. When state node s is expanded, transforms in a repository are considered to be connected. If the distance (e.g., calculated

by $dist_1$ in Sec. 4.2.1) between the output parameters of the transform in s and the input parameters of the transform t being considered for connection is zero, transform t is connected to s. Otherwise, the new state node created by connecting t to s is pruned, since the goal of our algorithm is to find a complete composition that has no distance between connecting transforms.

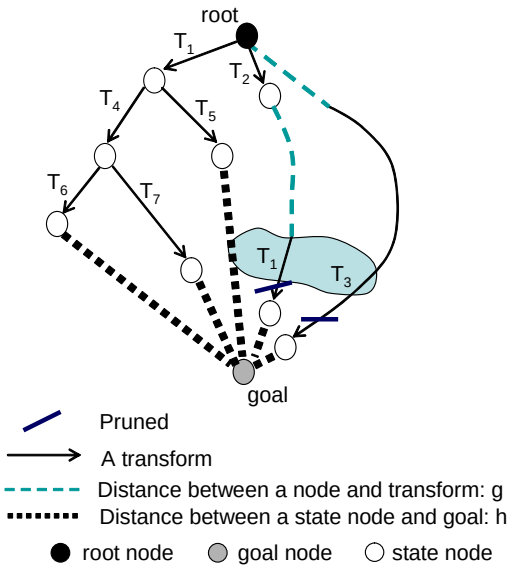

Figure 4-4. Part of a data transform composition graph.

The paths of all state nodes become candidates for partial compositions. Among candidates, the partial compositions that we filter out are the union of the following:

(1) A set of paths that are bottom-k based on f(x)(i.e., g + h) in (4-1). It means the path is close to the goal node and has less distance between transforms. The measure we use is the same as what we use in a search for a complete composition in a data transformation graph.

(2) A set of paths that are bottom-k based on h(x)(i.e., h) in (4-1) It means the path is close to the goal node. The intuition is that the last transform (e.g., T_1 or T_3 in Fig. 4-4) that is close to the goal node can be useful to construct a complete composition even though the path may have a large gap between transforms.

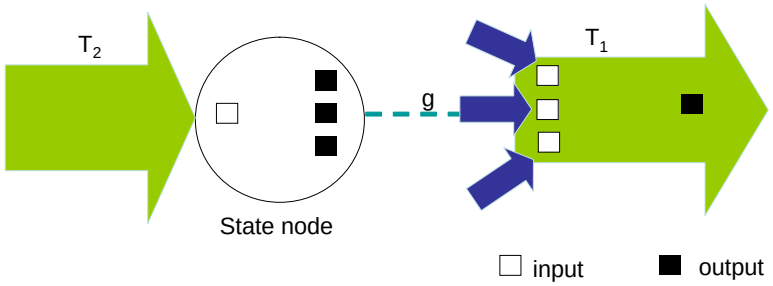

Figure 4-5. Backward search for suggesting partial compositions.

As shown in Fig. 4-5, we can try to fill the big gap between T_2 and T_1 using the backward search from the inputs of T_1. The reason for the gap is that the input of T_1 is not matched to the output of the transform at the previous state node. We then try to find any transform that can generate any input of T_1. Using a backward search, we may find a path from the root to one of the inputs of T_1.

CHAPTER 5
IMPLEMENTATION

In this chapter, we introduce the overall architecture of our data transform composition system we have implemented, and then explain important components in detail.

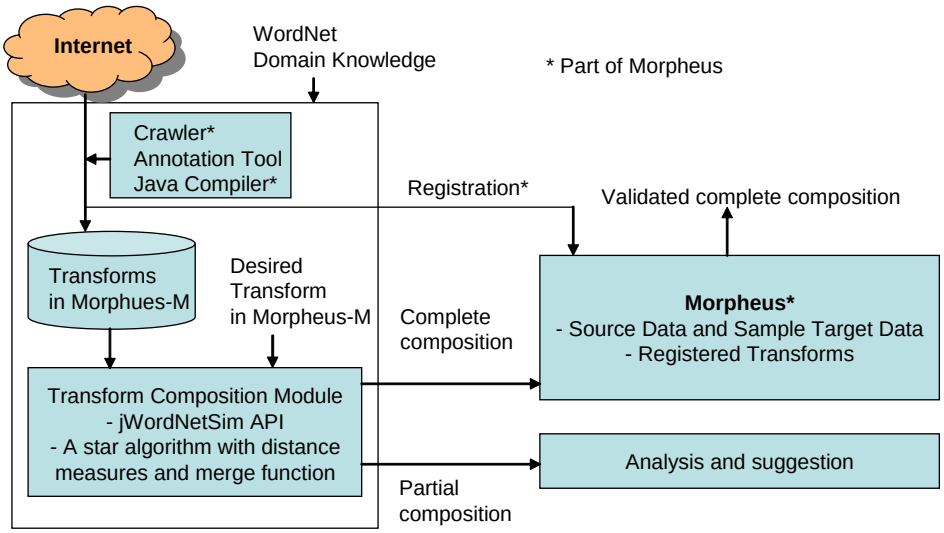

Figure 5-1. The architecture of our system.

Fig. 5-1 shows the overall architecture of our solution approach to data transform composition problem. Broadly, our architecture has two parts: one is where complete compositions are generated using transforms in repository, and the other is where the complete compositions are executed over source data. We use Morpheus (see Section 2.5) as an execution platform of the complete compositions.

In order to construct a transform repository, we need to harvest transforms and describe them in our transform meta model. As shown in Fig. 5-1, using our *annotation tool*, crawled software components from the Internet are represented in our transform meta model, Morpheus-M, and stored in repository. At the same time, they are compiled into java programs and registered in Morpheus for future execution.

With a given desired transform specification by a users in Morpheus-M, the *transform composition module*, which includes the A* algorithm, distance measures and Merge function introduced Chapter 4, finds complete compositions that are the same as the desired transform or similar to the desired transform using transforms in repository. The transform composition module uses jWordNetSym library to compare transforms in Morpheus-M. When we store and compose transforms, we use the WordNet dictionary and internal domain knowledge.

In fact, it can be challenging to discern the inside program logic of a transform which is crawled from Web services. Therefore, in order to decide the correct composition among the complete compositions our algorithm has found, we need to apply the complete compositions to actual source data to validate whether the solution generates the target data correctly.

After one complete composition is found, the complete composition is compiled into java programs and registered in Morpheus. We execute an SQL query including the registered composite transform as a UDF over the source data in Morpheus. If the complete composition generates data that are the same as sample target data correctly, the complete composition is the transform that a user wants to create. In case our system cannot find the validated complete composition, we suggest the partial compositions to a user.

5.2 Semantic Annotation and Registration Tool

In our problem, we assume that we have a large number of transforms in repository and our challenge is to find or compose a new transform using those transforms. The transforms can be created from scratch using tools such as Morpheus TCT or harvested from the Internet using the crawler in Morpheus [19]. Transforms in repository must be represented in our transform model Morpheus-M.

We developed a semantic annotation tool in order to represent crawled transforms in our model. The Crawler developed in Morpheus project harvests URLs that have web

services, java functions, and web forms. Those three software components are introduced next.

- Java (or other programming languages like c++) functions: These are from web sites that provide programs in HTML pages written in specific languages. The codes can be crawled and wrapped as executable software components.
- Web forms: These are HTML pages that have forms in order to get or provide information from/to users. The data given by a user are submitted to a Web server and results from the server appear on the next html page.
- Web services: These are Web services that provide services and advertise their services using WSDL. A wrapper can use WSDL description to create a transform that wraps operations in WSDL. Finally, the wrapped function in service consumer calls a service provider and a result will be back to the service consumer after the service provider process the request.

From the above three software components, we use Web services in order to construct an experimental repository. We developed a tool that parses pages in WSDLs. We parsed the WSDL files and translated them into our transform meta model. Currently, we do the annotation manually, but meta data we annotate once are accessible into the future consistently to enrich automatic data transform composition [44].

Besides web services, web forms could be another candidate for crawling. Unlike web services that are provided with WSDL files and text-based explanatory web pages, web form has better extractable information on related HTML pages. For example, web form has a page that include a form and corresponding java script in order to check whether the form input given by a user is correct. For example, for a text edit control in a form, we can extract a name and label of the control. The label of the control is provided to the users of the web form to explain the text control, therefore it itself has a very precise meaning. In addition, Java scripts are usually accompanied by a form that has functions that let a user give the input in the correct format. Therefore, we can extract more correct semantic meaning and format of the inputs through the Web forms.

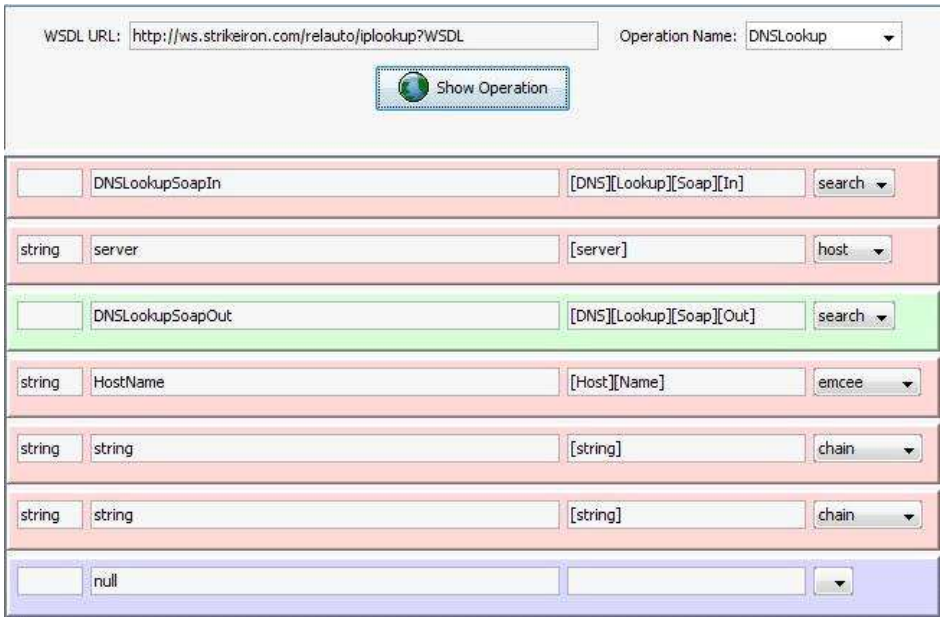

Figure 5-2. Web service semantic annotation tool.

Fig. 5-2 shows a GUI of our annotation tool. We parse a WSDL file and extract *operations*. For each operation, we extract information regarding input and output *messages* that are about input/output parameters of the operation. For each parameter, we extract the *name*, *data type* of the parameter. Using the *name*, we list words in the WordNet dictionary in order to let a user be able to select a word that has the same semantic meaning as the name. In addition, our tool lists possible formats for the word and a user can select the representation of the parameter among them. In short, using our semantic annotation tool, a user can annotate each parameter with the data type, semantic meaning, and representation that are required in order to be represented in our transform meta model.

5.3 Transform Composition Module

In this section, we explain the techniques we use to improve the performance of our transform composition module. With n and m attributes in the source and target schemas, respectively, and k transforms in the repository, the search space necessary for finding semantic mappings between the source and target schemas by transform composition

is large. For each attribute in a target schema, there must be matching between some attributes in the source schema and one attribute in a target schema. Our problem was to find all schema mappings for attributes in a target schema by composing transforms in a repository.

In order to reduce the search space, we apply following techniques:

Rather than trying to find mapping between all attributes in the source schema and one candidate attribute in the target schema, a user can specify attributes of the source schema in two sets, A and B. A is a set of attributes in the source schema that is certain to be usable for generating a specific attribute in the target schema and B is a set of attributes that might or might not be helpful for generating schema mappings to generate the specific attribute. We define a clause in Definition 4 in Sec. 4.1 for this purpose.

In the Web service composition area, usually six single Web services are enough to create a new complex Web service [37]. We limit the search depth to six based on the justification. In addition, we use a threshold to select connecting nodes from the current node using the distance between transforms. This can be justified because our approach tries to find a complete composition without distances between transforms.

CHAPTER 6
EVALUATION

6.1 Experimental Environment

In order to evaluate our algorithm, we construct an experimental repository. We use 640 Web services collected in [24] and gather 200 Web services on the Internet by using the Morpheus crawler [19]. In adition, we surf HTML pages that have Web forms (e.g., `http://www.onlineconversion.com`) and purposely create 25 transforms for use in our experiment. The 25 transforms include required transforms for constructing goal transforms and transforms similar to the required transforms in order to show our approach can find a complete composition.

All transforms in repository are represented in our transform meta model Morpheus-M. Using our semantic annotation tool introduced Sec. 5.2, we convert Web services in WSDL to our transform meta model Morpheus-M. Table 6-1 shows part of our transform repository.

The machine we use has an Intel(R) Pentium Dual CPU 2Ghz with 3GB RAM and use Windows Vista. We use two examples in our experiments. *ExampleA* is the *Starbucks Revenue Conversion* example introduced in Chap. 1, and *ExampleB* is the *Employee Payment Conversion* example.

ExampleB has two relational schemas, S and T shown in Fig. 6-1. In a global company, suppose employees working in the branch located in the US move to a branch in Korea and will get be paid in Korean Won instead of US dollars. The schema S is the relation to the branch in the US and T is to the one in Korea. We need to convert data in S to data fit in T. For the conversion, we need schema matchings and semantic mappings between two schemas.

6.2 Experimental Results

In this section, we show a series of experiments. We first show the case when we can find complete compositions with ExampleA. Second, we verify our transform meta model

Table 6-1. Sample transforms

ID	Input	Output	Function
1	m_1:money in Korean d_1:date in dd-mm-yyyy	m_2:money in USD	Convert money in Korean to money in USD with an exchange rate of a given date
5	m_1:money in Korean m_2:money in Korean	m_3:money in Korean	Add two moneys
13	m_1:money in Korean	m_3:money in USD	Convert money in Korean to money in USD with an exchange rate at the execution time
28	m_1:money in USD m_2:money in USD	m_3:money in USD	Add two moneys in USD
34	m_1:money in Korean m_2:money in Korean	m_3:money in USD	Add two moneys in Korean currency and then convert to USD with an exchange rate at the execution time
104	d_1:date in dd-mm-yyyy	e_1:exchange rate from Korean to USD	Get exchange rate from Korean to USD at the given date
105	m_1:money in Korean e_1:exchange rate	m_2:money in USD	Convert money in Korean currency to money in USD with a given exchange rate
122	d_1:date in dd-mm-yyyy m_1:money in Korean m_2:money in Korean	m_3:money in USD	Add two moneys in Korean currency and then convert to USD with an exchange rate of a given date
127	m_1:money in Korean m_2:money in Korean	m_3:money in USD	Subtract m_2 from m_1 and assign to m_3

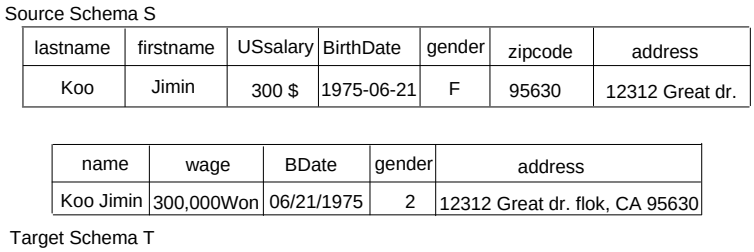

Figure 6-1. The schema of ExampleB.

Table 6-2. Experimental result

Experiment	**Composition**
d_2 of T_d	112
	1-5
	5-1
	5-104-105
	28-104-5
	104-105-5
	104-5-105
	104-28-5

and show the contribution of our model. Third, we show the scalability of our approach, and finally, we show the case when our approach finds partial compositions in case there is no complete composition. We use ExampleA for the first three experiments, and use ExampleB for the last experiment.

6.2.1 Experiment with ExampleA: Find Complete Compositions

Fig. 6-2 shows the specification when we have given schema matches. The d_2 means there is a match between three attributes (i.e., $S_1.date$, $S_1.revenue$ and $S_1.revenue$) of S_1 and one attribute (i.e., $S_2.revenue$) of S_2. Our composition algorithm finds all possible compositions that can generate $S_2.revenue$ with $S_1.date$, $S_1.revenue$ and $S_1.revenue$ with transforms in the repository.

Table 6-2 shows our experimental results. Our algorithm finds a set of single or composite transforms for d_2. A unique number for each transform is connected with a dash as a delimiter. For example, `5-104-105` is a connection of three transforms identified as 5, 104 and 105 (cf., Table 6-1). We can not be sure that the result of our algorithm is the correct transform that converts source data to target data because we do not know the

$T_1 = \{d_1, d_2, d_3\}$

$d_1.I = \{S_1.date\}[]$
$d_1.O = \{S_2.date\}$ $d_1.OP = \{op_1\}$
$S_1.date = \{$'date', 'date', 'MM-DD-YYYY'$\}$
$S_2.date = \{$'date', 'date', 'DD/MM/YYYY'$\}$
$op_1 = \{OP\_1, OP\_O\} = \{ d_1.I, d_1.O\}$

$d_2.I = \{S_1.date, S_1.revenue, S_1.revenue\}$ []
$d_2.O = \{S_2.revenue\}$ $d_2.OP = \{op_1\}$
$S_1.date = \{$'date', 'date', 'MM-DD-YYYY'$\}$
$S_1.revenue = \{$'money', 'float', 'Korean'$\}$
$S_1.revenue = \{$'money', 'float', 'Korean'$\}$
$S_2.revenue = \{$'money', 'float', 'USD'$\}$
$op_1 = \{OP\_1, OP\_O\} = \{ d_2.I, d_2.O\}$

$d_3.I = \{S_1.branch\}$ []
$d_3.O = \{S_2.city\}$ $d_3.OP = \{op_1\}$
$S_1.branchname = \{$'branch_name', 'string', ''$\}$
$S_2.city = \{$'city_name', 'string', 'fullname'$\}$
$op_1 = \{OP\_1, OP\_O\} = \{ d_3.I, d_3.O\}$

Figure 6-2. The desired transform specification 1.

exact internal program logic of each transform in the repository and they are abstractly represented in our model. We simply find possible answers using our model and distance measures. Therefore, as a next phase, we apply those transforms we found to the source data and check whether the transforms can generate target data correctly. As shown in Fig. 5-1, we execute the resulting single or composite transforms on Morpheus over existing source data with a query that has a transform as a UDF and executes.

We execute the resulting composite transforms over source data. The compositions 5-104-105(or 104-5-105), 122, and 5-1 generate correct target data.

6.2.2 Experimental Case 1: Model Verification

In this experimental case, we show that utilizing meta data defined in our transform meta model greatly improves our search in terms of efficiency and precision, thus increases the usability of our data transform composition approach.

In search of a goal transform in a transform composition graph, we compare output parameters of the previous transform to input parameters of the next transform to see whether two transforms can be connected or not. We use our meta data, namely primitive

data type, semantic meaning, and data format of parameters, in the comparison. Once output parameters and input parameters are matched by our algorithm with zero distance, two transforms are connected. Otherwise, the path from the root to the next transform is pruned. The primitive data type and semantic meaning are also used in WSDL and WSDL-S, but those languages do not have the data format of a parameter specifically. We define the meta data utilization levels from 1 to 3 to verify our model.

1 use data type only
2 use data type and semantics
3 use data type, semantics, and data format

Formally, the sim_e in formula (4-2) (i.e., the formula calculating the distance between two transforms) is changed as follows as we change the level of the meta data utilization. Below formula (6-1) means level 1, (6-2) means level 2, and (6-3) (the same as (4-2)) means level 3 (see Sec. 4.2.1 for a detailed explanation of these formulae).

$$sim_e(u_p, v_q) = sim_D(u_p.D, v_q.D), \quad (6\text{–}1)$$

$$sim_e(u_p, v_q) = (sim_S(u_p.S, v_q.S) + sim_D(u_p.D, v_q.D)) * 1/2., \quad (6\text{–}2)$$

$$sim_e(u_p, v_q) = (sim_S(u_p.S, v_q.S) + sim_D(u_p.D, v_q.D) + sim_R(u_p.R, v_q.R)) * 1/3., \quad (6\text{–}3)$$

where u_p is an output parameter of the previous transform and v_q is an input parameter of the next transform.

Additionally, we define the *model-based answer* and *validated answer*. Our algorithm uses the meta data of a transform to find a complete composition rather than using the intermediate data generated by executing a transform. Therefore, the complete composition found by our algorithm, namely a model-based answer, may not be the right data transform, the one that can convert the source data to target data correctly. Thus, as soon as we find a model-based answer, we apply it to the source data and check whether the model-based answer generates target data correctly. If a model-based answers generates target data correctly, we call it a validated answer. We can apply a model-based

answer to the validating process as soon as it is found, so the validating process can be performed in parallel with the search for model-based answers. When a validated answer appears, our search for the complete composition can be stopped.

Next, we define a *participating transform*. In a transform composition graph, all transforms in the repository are considered to be connected to the current expanding node. The current expanding node has a transform that is constructed by merging transforms in the path from the root node to the current expanding node. Therefore, the outputs of a transform in the current expanding node are compared to the inputs of the transform that is considered to be connected. If there is no distance between output and input parameters, two transforms can be connected, and the transform connected to the current expanding node becomes a participating transform. The execution time of our transform composition algorithm can be exponential to the number of participating transforms.

Meta data utilization vs. execution time. With 150 transforms, we run our algorithm by varying the meta data utilization levels from 1 to 2 to 3. Fig. 6-3 shows that as the meta data utilization is increased, the execution time is greatly decreased. The execution time is the time to find all model-based answers using our approach (i.e., the worst case is that a validated answer appears last among all model-based answers).

Our experiment shows that our meta data (specifically, data format of a parameter) are useful to quickly find the model-based answers, which is a set of candidate answers, including validated answers. The more we use meta data in a search, the better we can filter out useful transforms for composing a goal transform. Consequently, the number of participating transforms in a search is decreased, and it decreases the execution time exponentially. In short, using our meta data reduces exponential search space. As in [44], once manually created, meta data can consistently improve the searching capability in our data transform composition.

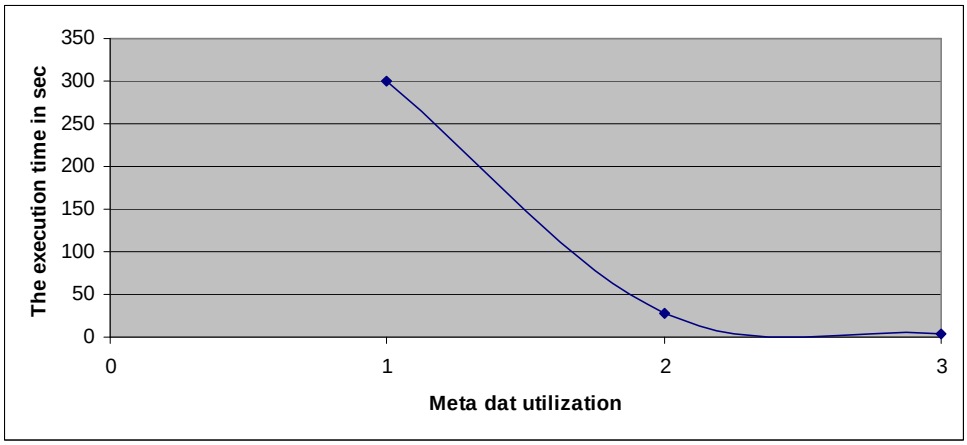

Figure 6-3. Efficiency by varying meta data utilization with 150 transforms.

Meta data utilization vs. precision. In this experiment using 150 transforms, we run our algorithm by varying meta data utilization level from 1 to 2 to 3. Fig. 6-4 shows how the precision of our algorithm is changed as the meta data utilization level is increased. The precision and recall in our experiment are defined as follows:

Precision = retrieved validated answers/total retrieved model-based answers

Recall = retrieved validated answers/total validated answers

Since our approach finds all validated answers even as we vary the meta data utilization level, we can show how the precision is changed. As in Fig. 6-4, meta data utilization level 3 has the highest precision. The more we use our meta data, the better our algorithm can filter out useful transforms. Consequently, the number of model-based answers is reduced as we increase meta data utilization. Since the number of retrieved validated answers is the same even as we increase the meta data utilization, the precision is increased as we increase the meta data utilization. Our goal is to find the first validated answer among model-based answers. Our experiment shows how precisely we can find the first one we want, and our transform meta model assures the high precision.

Other interesting possible experiments can be performing by applying distance measures used in other research (e.g., Web service composition) to our framework and showing the accuracy of answers. However, we can demonstrate with our experimental

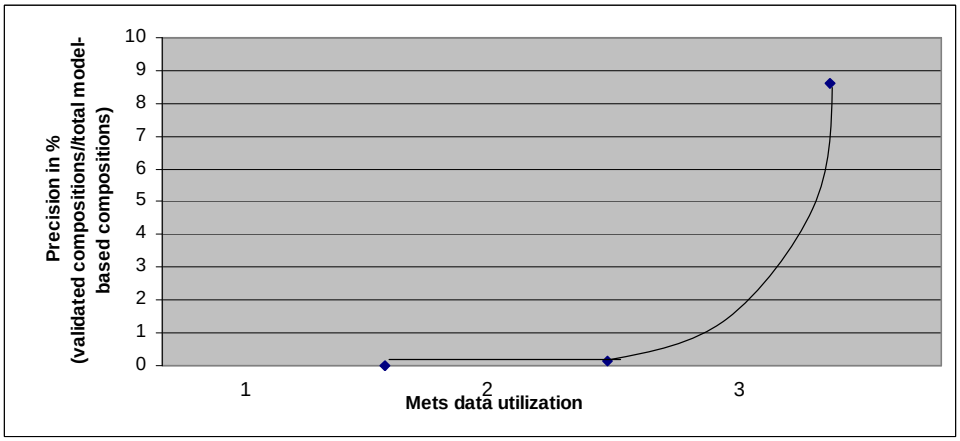

Figure 6-4. Precision by varying meta data utilization with 150 transforms.

results that those will not narrow down well to the correct answers since they use less meta data than our approach. This will generate the same phenomenon as the second experiment in this section.

$dist_2$ **vs. the number of expanded nodes**. In the search for complete compositions, we use the following h(x) that is a heuristic distance from a node x to goal.

$$h(x) = dist2(d_k, x.MT, I + OP) * w_1 + dist2(x.MT, d_k, O) * w_2, \tag{6–4}$$

For the first part of h(x), we can change the flag (i.e., I+OP) of dist2. The flag I means inputs of a transform, O means outputs of a transform, and OP means operations of a transform. $I+OP$ means that we consider inputs and operations when we calculate the distance and do not consider the outputs. We experiment how many nodes in a transform composition graph are expanded to find each complete composition by varying the flag (i.e., I+OP, I+O+OP, I+O). Three experiments by varying the flag generate the same 53 complete compositions. Our experiment shows the quality of distance measures. If we can estimate the heuristic distance better, search will go to the answers faster with less expanded nodes. Fig. 6-5 shows that using $I+OP$ or $I+O+OP$ flags are better than using $I+O$ flag. In other words, considering the operation defined in our transform meta model can proceed to the answers faster. This justify the necessity of operation in our

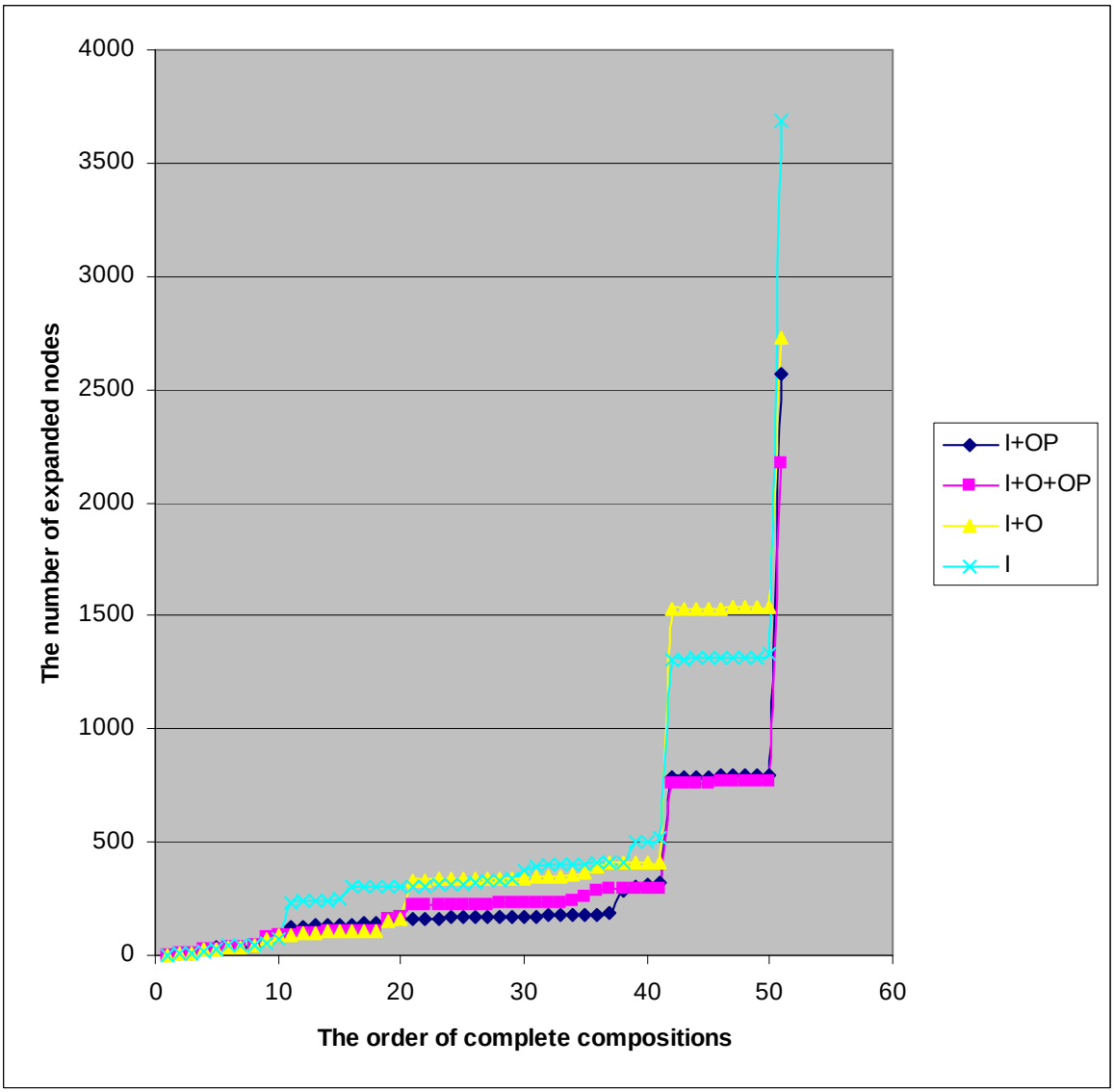

Figure 6-5. The number of expanded nodes so far when each complete composition appears (total 53 answers). We vary h(x) by changing the flag (e.g., I+O, I+O+OP, I+OP, I). We test with 1000 transforms in repository. The search is terminated when total 3788 nodes are expanded.

transform meta model. The number of expanded nodes is also related to the execution time of the search.

6.2.3 Experimental Case 2: Efficiency of Our Algorithm

In this experimental case, we show the efficiency of our algorithm. First, we show how the execution time for finding all model-based answers is changed as we vary the size of

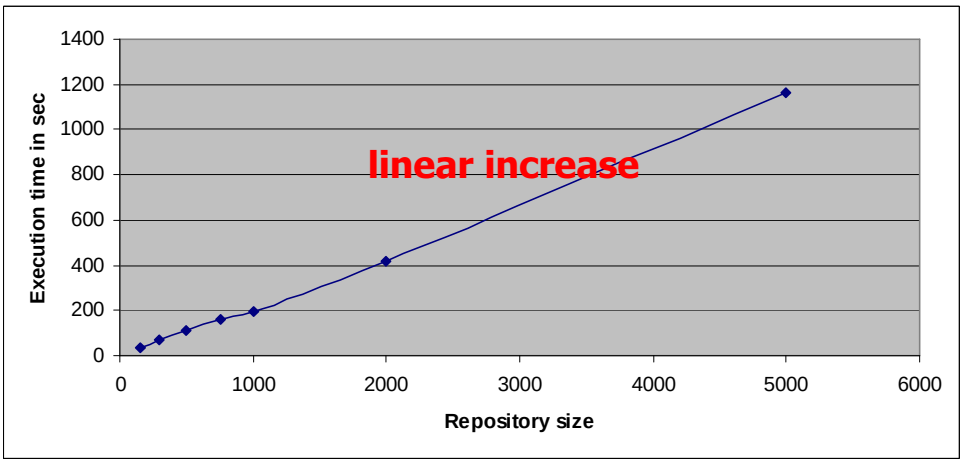

Figure 6-6. Efficiency by varying the size of repository.

repository. Next, we show how the worst/best case execution time is changed as we vary the number of participating transforms.

Size of repository vs. execution time. We vary the size of the repository from 150 to 5000 and each has the same number of participating transforms. Fig. 6-6 shows that our algorithm is scalable to the size of repository if the number of participating transforms are equal. This is because of our algorithm, which prunes search space with meta data of our transform meta model. The number of participating transforms is related to the exponential search space, but our algorithm (specifically, matching parameters) is linear to the size of repository. The algorithm tries to find all possible model-based answers in a transform composition graph that can lead to a goal node by matching parameters of consecutive transforms. Then we apply those model-based answer to the source data in order to find a validated answer.

Recent techniques in semantic mapping use intermediate data generated by executing transforms (or operator, searcher in their context) intensively [6, 18, 25] to find complex matchings between two schemas. Those techniques can reach the correct answer directly (i.e., they have one step to the final answer, unlike our two steps), but they are not able to easily prune a search space using intermediate data generated by executing a connecting transform because it is difficulty to figure out by intermediate data whether it is useful

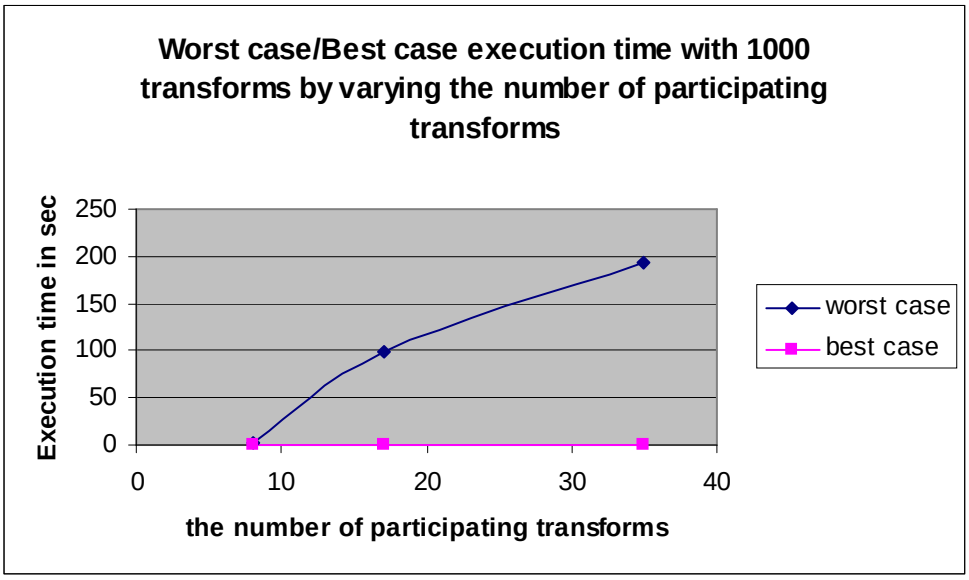

Figure 6-7. Worst case/best case execution time as a result of varying the number of participating transforms

data flow or not in the search for a path to a goal. We can claim that pruning with our matching algorithm keeps the possible data flow that can generate a goal transform, and reduce the search space effectively. As a result, we separate the transform composition phase from the transform execution phase, which makes our approach more realistic because we assume that our transforms can be remote Web services.

The number of participating transforms vs. execution time. By using ExampleA, we show how the worst case/best case execution time is changed as we change the number of participating transforms in 1000 transforms. Fig. 6-7 shows the worst/best case execution times for finding a validated answer. Basically, we use the A* algorithm, which finds the best one first (i.e., the best path in terms of our distance measures). Therefore, the first answer appears quickly, but finding all answers takes time similar to a brute-force search. If there is no answer, we can search the entire space like a brute-force search. In our algorithm, we terminate a search when it reaches the execution time constraint, and then move on to find partial compositions. However, if there is an answer, we can stop the search as soon as the first validated answer appears. In the case

of ExampleA, the first validated answer appears quite quickly, as shown in Fig. 6-7 even as we increase the number of participating transforms.

6.2.4 Experiment Case 3: Partial Composition

We use ExampleB in the previous section to show how we provide partial compositions. We assume in Fig. 6-1 that we do not have exact schema matchings between S and T. Our goal transform is d_3, the inputs of which are *USsalary* and *zipcode* attributes in S and an output is the *wage* attribute in T. Let us assume that the input attributes in S are described by a user as follows:

{*USsalary*}[*zipcode*]

As in Definition 4 in Sec. 4.1, a user specifies that *USsalary* in S is required to find a semantic mapping for *wage* in T and *zipcode* might or might not be used. With the inputs, our algorithm tries to find complete compositions that can generate the output *wage* attribute using *USsalary* and *zipcode*. With 160 transforms in the repository, our search is terminated without a complete composition.

Among the paths of state nodes generated during the search for the complete composition, we filter out the following sets: The set A means bottom-10 paths based on f(x) in (4-1) and the set B means bottom-10 paths based on h(x) in (4-1), respectively. The numbers below are the identification of a transform.

A= {23-6, 157-158, 23-6-31, 165-6, 157-158-103, 23-6-103, 161-130-6, 161-130-6-9, 161-130-53-6, 161-130-6-148-27}

B = {23-6, 157-116, 157-158-103, 23-6-31, 165-6, 165-123, 165-125, 23-6-103, 161-130-6, 161-130-6-3}

As a result, our partial compositions is A∪B.

A∪B = {23-6, 157-158, 157-116, 157-158-103, 23-6-31, 157-158-103, 165-6, 165-123, 165-125, 23-6-103, 161-130-6, 161-130-6-9, 161-130-53-6, 161-130-6-3, 161-130-6-148-27}

Above partial compositions are helpful to the user to catch out about useful transforms in repository. The part of partial compositions (e.g., 23-6) cannot be related

to the desired transform, since we connect mathematical functions (e.g., addition) by just matching the primitive data type. We found that the paths including that 157, 158, 116, 123, and 125 can be helpful for a user to construct the desired transform.

CHAPTER 7
CONCLUSION

In this chapter, we summarize our research and describe future works.

7.1 Summary and Contributions

Data transformation resolves heterogeneities between disparate schemas and is indispensable in many applications where data sharing and exchange happen. Manually performing two main steps (i.e., schema matching and semantic mapping) in data transformation is extremely time-consuming and labor-intensive. Recent research has been done in an automatic data mapping and semantic mapping, but those work concentrate mostly on the structured data mapping or applying a restricted set of transforms in composition.

Unlike previous works, our work in this dissertation focuses on the semi-automatic data transform composition that reuses transforms in repository to construct users' desired transforms. Finding semantic mappings by composing transforms needs massive search space, therefore we need a sophisticated solution. In addition, we cannot guarantee that transforms in repository are complete for composing any new transform. We have following challenges: how to formally represent transforms, how to efficiently find or compose transforms, and how to provide partial solutions in case there is no complete solutions.

We create the transform meta model and the transform in our model can be represented in a graph. We can compare two transforms semantically using graphs. The meta data in our model also can be represented in Resource Description Framework(RDF) [35] triples semantically and those RDF triples can construct structural RDF graphs. By using the RDF framework, our transform meta model gain merits such as new semantic meta data can be easily expandable by adding RDF triples when we add more meta data to accommodate new requirements to describe a data transform and data transforms in our meta model are interchangeable and can be compared with

other resources on Web since the RDF is widely accepted as a standard for semantic representation of information on the Web and it is represented in an XML.

Our model includes not only the primitive data type and semantic meaning of a parameter as in existing standards in Web services but also the data format of a parameter. In addition, meta data on operations describe the relationships between inputs and outputs of a transform. Our experiment shows that our meta model greatly speed up searching for complete compositions and provide high precision.

Based on our transform meta model, we model a data transform composition problem as a graph search problem and design our sophisticated distance measures (i.e., similarity measures) to progress a search using the A* algorithm. Our sophisticated distance measures enable our search to progress to a complete composition correctly and to reduce exponential search space by pruning. In composing transforms, our algorithm keeps the behavior of transforms as much as possible using our transform meta model, thus we can find a goal transform correctly. When there is no complete composition, our algorithm provides partial compositions that are useful to construct a goal transform among the transforms in repository.

We design and implement our prototype system for semi-automatic data transform composition using transforms in our model in a repository. Our system provides an automatic search of a large repository to find or compose a desired transform. Our semantic annotation tool is used to convert crawled Web services in WSDL to our model semi-automatically. Using our system, users can reduce their efforts in time-consuming and error-prune semantic mapping steps of data transformation process, thereby reducing efforts in information integration that requires in many applications.

7.2 Future Work

The automated data transform composition problem is a challenging problem. The work presented in this dissertation can be extended in the following directions:

The *Deep Web* [29, 10, 34] refers to content hidden behind HTML forms. In order to get to such content, a user has to perform a form submission with valid input values. Search engines like Google try to retrieve the hidden contents along with general HTML pages. Our framework for data transform composition can be applied to automatic Deep Web query processing.

For example, a user can frame a query like "How long does it take to go from Orlando International Airport to Disney World by car?" In order to answer the question, we need to extract Deep Web content step by step, such as getting the addresses of Orlando International Airport and Disney World, getting the distance between them, and calculating the time to drive the distance by car. The query can be answered using the Deep Web content of the Google Map site (e.g., `http://maps.google.com/maps?hl=en&tab=wl`) and other web sites. Our framework for automatic data transform composition research can be applied to Deep Web query processing using a transform repository that stores Web forms as transforms. Our transform can be represented in RDF triples, our transform can easily compared with Web resources represented in RDF.

Second, we can extend our work on optimizing the execution of a composed data transform. The composed data transform is compiled into a Java class file, registered in the Postgres DBMS as a UDF, and invoked on the source data set to be transformed using an SQL query for actual transformation. Currently, the registered transform in UDF is treated as a black box during query optimization [13, 12], so there are limitations to optimizing a query invoking the composed data transform. It would be beneficial to find an opportunity for more optimization by looking inside the composed data transform description.

Third, we show that our transform meta model makes a huge contribution to decrease the exponential search space of our data transform composition. We can work on gathering other valuable meta data as automatically as possible [57]. As the kinds of semantic meta data are increased, to design effective distance measures can be challenging.

REFERENCES

[1] R. Akkiraju, J. Farrell, J. Miller, M. Nagarajan, M.-T. Schmidt, A. Sheth, and K. Verma. Web service semantics wsdl-s. http://www.w3.org/Submission/WSDL-S/.

[2] R. Akkiraju, A. Ivan, R. Goodwin, B. Srivastava, and T. Syeda-Mahmood. Semantic matching to achieve web service discovery and composition. In *CEC-EEE '06: Proceedings of the The 8th IEEE International Conference on E-Commerce Technology and The 3rd IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services*, page 70, Washington, DC, USA, 2006. IEEE Computer Society.

[3] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services: Concepts, Architectures and Applications*. Springer-Verlag, Berlin, Germany, 2003.

[4] D. Berardi, D. Calvanese, G. D. Giacomo, M. lenzerini, and M. Mecella. Automatic composition of e-services that export their behavior. In *Proc. of the 1st International Conference on Service Oriented Computing*, 2003.

[5] P. A. Bernstein and T. Bergstraesser. Meta-data support for data transformations using microsoft repository. *IEEE Data Eng. Bull.*, 22(1):9–14, 1999.

[6] P. A. Bernstein and S. Melnik. Model management 2.0: manipulating richer mappings. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1–12, New York, NY, USA, 2007. ACM.

[7] P. Carreira and H. Galhardas. Execution of data mappers. In *IQIS '04: Proceedings of the 2004 international workshop on Information quality in information systems*, pages 2–9, New York, NY, USA, 2004. ACM.

[8] F. Casati, S. Ilnicki, L.J.Jin, V. Krishnamoorthy, and M. Shan. Adaptive and dynamic service composition in eflow. In *Proc. of the International Conference on Adv. Info. aSystems Engineering*, 2000.

[9] G. Chafle, S. Chandra, V. Mann, and M. G. Nanda. Decentralized orchestration of composite web services. In *WWW '04: Proceedings of the 13th International World Wide Web Conference*. ACM, May 2004.

[10] K. C.-C. Chang and J. Cho. Accessing the web: from search to integration. In *SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 804–805, New York, NY, USA, 2006. ACM.

[11] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. *SIGMOD Rec.*, 26(1):65–74, 1997.

[12] S. Chaudhuri and K. Shim. Query optimization in the presence of foreign functions. In *VLDB '93: Proceedings of the 19th International Conference on Very Large Data Bases*, pages 529–542, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[13] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. *ACM Trans. Database Syst.*, 24(2):177–228, 1999.

[14] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. Web services description language (WSDL) 1.1. http://www.w3.org/TR/wsdl.

[15] W. M. Coalition. Process exchange specification language. http://www.wfmc.org/standards/docs/TC-1025_xpdl_2_2005-10-03.pdf.

[16] B. Coppin. *Artificial Intelligence Illuminated*. Jones and Bartlett Publishers, Sudbury, Massachusetts, 2004.

[17] S. B. Davidson and A. Kosky. Specifying database transformations in wol. *IEEE Data Eng. Bull.*, 22(1):25–30, 1999.

[18] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos. iMap: discovering complex semantic matches between database schemas. In *SIGMOD '04*, pages 383–394, 2004.

[19] P. Dobbins, T. Dohzen, C. Grant, J. Hammer, M. Jones, D. Oliver, M. Pamuk, J. Shin, and M. Stonebraker. Morpheus 2.0: A data transformation management system". In *InterDB, VLDB workshop*, 2007.

[20] T. Dohzen, M. Pamuk, S.-W. Seong, J. Hammer, and M. Stonebraker. Data integration through transform reuse in the Morpheus project. In *SIGMOD '06*, pages 736–738, 2006.

[21] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity search for web services. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 372–383. VLDB Endowment, 2004.

[22] B. A.-M. et al. Template-based semantic similarity for security applications. In *Technical Report, LSDIS Lab, Computer Science Department, University of Georgia*, Jan. 2005.

[23] D. M. et al. Owl-s: Semantic markup for web services. W3C, 2004.

[24] J. Fan and S. Kambhampati. A snapshot of public web services. *SIGMOD Rec.*, 34(1):24–32, 2005.

[25] G. H. Fletcher and C. M. Wyss. Data mapping as search. In *EDBT 2006:Advances in Database Technology*. Springer LNCS 3896, 2006.

[26] P. G. D. Group. Postgresql. http://www.postgresql.org/.

[27] A. Y. Halevy, Z. G. Ives, P. Mork, and I. Tatarinov. Piazza: data management infrastructure for semantic web applications. In *WWW '03: Proceedings of the 12th international conference on World Wide Web*, pages 556–567, New York, NY, USA, 2003. ACM.

[28] B. He, K. C.-C. Chang, and J. Han. Discovering complex matchings across web query interfaces: a correlation mining approach. In *KDD '04: Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 148–157, New York, NY, USA, 2004. ACM.

[29] B. He, M. Patel, Z. Zhang, and K. C.-C. Chang. Accessing the deep web. *Commun. ACM*, 50(5):94–101, 2007.

[30] J. M. Hellerstein. Optimization techniques for queries with expensive methods. *ACM Trans. Database Syst.*, 23(2):113–157, 1998.

[31] J. M. Hellerstein and M. Stonebraker. Predicate migration: optimizing queries with expensive predicates. In *SIGMOD '93: Proceedings of the 1993 ACM SIGMOD international conference on Management of data*, pages 267–276, New York, NY, USA, 1993. ACM Press.

[32] IBM. Semantic tools for web services. http://www.alphaworks.ibm.com/tech/wssem.

[33] IBM. Workflow management coalition. http://www.wfmc.org/.

[34] L. A. Jayant Madhavan, Loredana Afanasiev and A. Halevy. Harnessing the deep web: Present and future. In *CIDR '09: 4th Biennial Conference on Innovative Data Systems Research*, 2009.

[35] G. Klyne and J. J. Carroll. Resource Description Framework (RDF): Concepts and abstract syntax. W3C, 2004.

[36] P. Kungas and M. Matskin. Detection of missing web services: The partial deduction approach. In *NWESP '05: Proceedings of the International Conference on Next Generation Web Services Practices*, page 339, Washington, DC, USA, 2005. IEEE Computer Society.

[37] P. Küngas and M. Matskin. From web services annotation and composition to web services domain analysis. *IJMSO*, 2(3):157–178, 2007.

[38] D. Lin. An information-theoretic definition of similarity. In *ICML 98: Proceedings of the Fifteenth International Conference on Machine Learning*, pages 296–304. Morgan Kaufmann Publishers, 1998.

[39] L. Lin and I. B. Arpinar. Discovery of semantic relations between web services. In *ICWS '06. International Conference on Web Services*, pages 357–364, Sep. 2006.

[40] J. Madhavan and A. Y. Halevy. Composing mappings among data sources. In *VLDB '2003: Proceedings of the 29th international conference on Very large data bases*, pages 572–583. VLDB Endowment, 2003.

[41] G. A. Miller. Wordnet: a lexical database for english. *Commun. ACM*, 38(11):39–41, 1995.

[42] J. Myerson. Work with Web services in enterprise-wide SOAs, part 5: Optimize web service applications with websphere business integration tools. *IBM developerWorks*, 2005.

[43] S.-C. Oh, B.-W. On, E. J. Larson, and D. Lee. BF*: Web services discovery and composition as graph search problem. In *EEE '05: Proceedings of the 2005 IEEE International Conference on e-Technology, e-Commerce and e-Service*.

[44] N. I. S. Organization. Understanding metadata. *NISO Press*, 2004.

[45] S. R. Ponnekanti and A. Fox. Sword: A developer toolkit for web service composition. In *Proc. of the 11th International Conference on WWW*, 2002.

[46] M. P.Singh. The pragmatic web. In *IEEE Internet Computing*, pages 4–5, 2002.

[47] G. Qian and Y. Dong. A step towards incremental maintenance of the composed schema mapping. In *CIKM '08: Proceeding of the 17th ACM conference on Information and knowledge management*, pages 173–182, New York, NY, USA, 2008. ACM.

[48] E. Rahm and P. A. Bernstein. On matching schemas automatically. *VLDB Journal*, (4), 2001.

[49] E. Rahm and P. A. Bernstein. A survey of approaches to automatic schema matching. *The VLDB Journal*, 10(4):334–350, 2001.

[50] E. Rahm and H. H. Do. Data cleaning: Problems and current approaches. *IEEE Data Eng. Bull.*, 23(4):3–13, 2000.

[51] E. A. S. Ghandeharizadeh and S. Manjunath. Proteus RTI: A framework for on-the-fly integration of biomedical web services. In *USC Databse Laboratory Technical Report Number 2006-05*, 2006.

[52] J. Shin, J. Hammer, and H. Lam. RDF-based approach to data transform composition. In *7th IEEE/ACIS International Conference on Computer and Information Science, IEEE/ACIS ICIS 2008, 14-16 May 2008, Portland, Oregon, USA*, pages 645–648, 2008.

[53] J. Shin, J. Hammer, and W. J. O'Brien. Distributed process integration: Experiences and opportunities for future research. In *IEEE International Workshop on Web and Mobile Information Systems (WAMIS)*, 2006.

[54] A. Simitsis, P. Vassiliadis, and T. Sellis. Optimizing etl processes in data warehouses. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering*, pages 564–575, Washington, DC, USA, 2005. IEEE Computer Society.

[55] A. Simitsis, P. Vassiliadis, and T. Sellis. Optimizing ETL processes in data warehouses. *ICDE*, pages 564–575, 2005.

[56] A. Sirbu and J. Hoffmann. Towards scalable web service composition with partial matches. In *ICWS '08: Proceedings of the 2008 IEEE International Conference on Web Services*, pages 29–36, Washington, DC, USA, 2008. IEEE Computer Society.

[57] R. Sumra and A. D. Quality of service for web services-demystification, limitations, and best practices. *Developer.com*, 1999.

[58] T. Syeda-Mahmood, G. Shah, R. Akkiraju, A.-A. Ivan, and R. Goodwin. Searching service repositories by combining semantic and ontological matching. In *2005 IEEE International Conference on Web Services*, pages 13– 20, 2005.

[59] I. Tatarinov and A. Halevy. Efficient query reformulation in peer data management systems. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 539–550, New York, NY, USA, 2004. ACM.

[60] P. S. M. Tsai and A. L. P. Chen. Optimizing queries with foreign functions in a distributed environment. *IEEE Transactions on Knowledge and Data Engineering*, 14(4):809–824, 2002.

[61] K. Verma. Configuration and adaptation of semantic web processes. In *Ph.D Thesis, Computer Science, Univ. of Georgia*, June 2006.

[62] W3C. Web Services Architecture Requirement. Technical report, W3C, 2002.

[63] D. Wu, B. Parsia, E. Sirin, J. Hendler, and D. Nau. Automating daml-s web services composition using shop2. In *Poc. of 2nd International Semantic Web Conference*, Oct. 2003.

[64] L. Xu and D. W. Embley. Discovering direct and indirect matches for schema elements. In *DASFAA*, pages 39–46, 2003.

[65] C. Yu and L. Popa. Semantic adaptation of schema mappings when schemas evolve. In *VLDB '05: Proceedings of the 31st international conference on Very large data bases*, pages 1006–1017. VLDB Endowment, 2005.

[66] H. Zhu, J. Zhong, J. Li, and Y. Yu. An approach for semantic search by matching rdf graphs. In *Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference*, pages 450–454. AAAI Press, 2002.

BIOGRAPHICAL SKETCH

Jungmin Shin received her B.S and M.S at the Department of Computer Science and Engineering from the Ewha Womans University in South Korea in 1993 and 1995, respectively. In 1995, she joined at Media Communications Lab of LG Electronics in Seoul, South Korea and worked on intelligent user interface as a researcher. In 1997, she joined SK Telecom in Seoul, South Korea. She worked on a large scale database management system such as the membership management system and bulletin board system of a commercial on-line portal service. Also, she involved in developing a video on demand (VOD) service in a commercial wireless internet service. Since 2001, she has been working on process integration and data transform composition at Database Systems Research and Development Center in University of Florida. She received her Ph.D. from the University of Florida in the fall of 2009.