TransformScout: Finding Compositions of Transformations for Software Re-Use

by

Mujde Pamuk

Submitted to the Engineering Systems Division
in partial fulfillment of the requirements for the degree of

Master of Science in Technology and Policy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2007

Author .... [illegible]
Engineering Systems Division
August 10, 2007

Certified by .... [illegible]
Michael Stonebraker
Adjunct Professor of Electrical Engineering and Computer Science
Thesis Supervisor

Accepted by .... [illegible]
Dava Newman
Professor of Aeroneutics and Astronautics and Engineering Systems
Director, Technology and Policy Program

TransformScout: Finding Compositions of Transformations for Software Re-Use

by

Mujde Pamuk

Submitted to the Engineering Systems Division
on August 10, 2007, in partial fulfillment of the
requirements for the degree of
Master of Science in Technology and Policy

Abstract

As organizations collect and store more information, data integration is becoming increasingly problematic. For example, nearly 70% of respondents to a recent global survey of IT workers and business users called data integration a high inhibitor of new application implementation.

A number of frameworks and tools have been developed to enable data integration tasks. The most prominent include schema matching, use of ontologies and logic-based techniques. A joint project by UFL and MIT, Morpheus, has attacked the same problem with a unique emphasis on re-use and sharing. In the first part of the thesis, we try to define software re-use and sharing in the context of data integration and contrast this approach with existing integration techniques. We synthesize previous work in the field with our experience demoing Morpheus to an audience of research labs and companies.

At the heart of a system with re-usable components is browsing and searching capabilities. The second part of this thesis describes TransformScout, a transform composition search engine that automates composition of re-usable components. Similarity and quality metrics have been formulated for recommending the users with a ranked collection of composite transforms. In addition, the system learns from user feedback to improve the quality of the query results.

We conducted a user study to both evaluate Morpheus as a system and to assess TransformScout's performance in helping completing programming tasks. Results indicate that software re-use with Morpheus and TransformScout has helped the user perform the programming tasks faster. Moreover, TransformScout was useful in aiding the users with completing the tasks more reliably.

Thesis Supervisor: Michael Stonebraker
Title: Adjunct Professor of Electrical Engineering and Computer Science

Acknowledgments

Thanks to my advisor and mentor Mike Stonebraker. Without his help this thesis would not have been possible.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

When organizations exchange information, three levels of heterogeneity may affect the quality of their communication. On the infrastructure level, underlying hardware and operating systems in corresponding information systems may have incompatibilities that need to be resolved to allow communication. Examples of such incompatibilities include differences in hardware, network and operating systems in the underlying IT infrastructures. On a logical level, different data models may be used to represent the exchanged data. For example, information about a customer may be stored in a variety of data models, from unstructured documents, such as web pages, to structured files, such as entries in a relational database. In addition, discrepancies between representations can occur on a conceptual level. Such discrepancies can be present because data that is being gathered from a variety of sources was not originally intended to be merged into a common schema. Such conflicts can be present even when the exchanged data has a common representation. For example, *"two days"* has a different meaning in the context of *"two business days"* versus *"two Wall Street days"* versus *"two Federal Express"* days. Alternatively, differences in how data is represented may be limited to structural discrepancies. For example, one organization may represent height in British measurement units whereas another organization may represent height in metric units.

As organizations collect and store more information, applying data integration techniques to resolve structural and semantic discrepancies is becoming increasingly problematic. For example, nearly 70% of respondents to a recent global survey of IT professionals and business users called data integration a high inhibitor of new application implementation [1]. The proliferation of service oriented architectures and the Internet will expand the scope of this integration problem from the boundaries of a single organization to between organizations. Furthermore, tolerance for latency in inter-enterprise communication is diminishing. Ideally, organizations would like to interact with one another and resolve their semantic conflicts in real time. This need for faster communication, together with increased desire to share data, will make data integration tasks even more challenging in the future.

Both the research community and industry have responded to the growing problem of reconciling heterogeneous data sources. Information integration has been touted as the "Achilles heel" of computing in all four self assessments of the DBMS community [2-5]. Various approaches to the problem of data integration have been proposed from schema matching [6], to ontology sharing [7] to logic based systems [8]. Industry's response to the problem has resulted in the development of three product categories: EII (Enterprise Information Integration), ETL (Extract-Transform-Load), and EAI (Enterprise Application Integration).

To understand the data integration problem faced by real organizations, we interviewed CTOs and CIOs of many companies, ranging from small companies to large corporations. We also interacted with research labs to understand their perspective on the issue. We synthesized the related work with this feedback and made some observations. First, we noticed that current approaches are somewhat limited in their success. A second observation is that the majority of integration efforts are directed at understanding what source and target representations mean. This leads us to believe that there is a need for better metadata management. A third observation is, writing the actual programs that will map a data source to a target representation is a time-consuming item in integration projects [9], which indicates a need to write mappings to and from different representations of data more efficiently.

Motivated by these insights, a team of researchers from MIT and UFL started a project called Morpheus. Our approach in Morpheus is to allow the user to write transformations between two schemas easily and save those transformations in a central repository that is browsable and searchable. The premise behind having such a repository is that there is a large amount of similarity between many transforms. By encapsulating enough metadata about the transform itself as well as its input and output data types in a central repository, it will be possible for the user to search for objects in the repository that are the most similar to a particular transform. Furthermore, once the repository is large enough, any transform not present in the repository will be a variant of one that is in the repository. The new transform can be easily constructed using our high-level programming environment. Hence, our goals in this project are the following: (1) make it easy to write transforms, and (2) make it easy to find and re-use transforms written by others.

Benefits from re-use are not limited to savings in a programmer's time. There are other advantages to maintaining a centralized repository within an organization including:

1. Lack of redundancy: Users can re-use transformations and metadata created by other people in their organization, eliminating the duplicate work of writing the same transformation multiple times.

2. Consistency: Data and application logic in an organization can be maintained in a uniform and transparent way. A representation of an entity can be used by everyone in the organization and changes to this representation are reflected instantaneously. A centralized approach, where application logic and metadata are overseen by a number of experts in information systems, can greatly reduce inconsistencies in the way organizational processes are described.

3. Efficiency: Similarly, users can speed up development by re-using similar, previously created transforms, rather than writing transformations from scratch.

4. Isolation of organizational logic from the programmer: An important goal of Morpheus is to provide user-friendly tools, which can help even non technical people to easily write and re-use transforms. This way, non technical people with the best knowledge of an organization's rules can write transforms without utilizing IT staff's time and resources.

5. *Maintainability of ETL cycle:* Unlike ETL systems, we carry out integration tasks (applying transforms to data) within the DBMS layer so that one does not need to maintain separate layers for storage and execution. This way we reduce the administrative and maintenance costs that increase with each additional layer added to the system [10].

1.2 TransformScout: A Transform Composition Search Engine

One of the basic goals of Morpheus is to let the user locate re-usable components easily. To this end, we created a sophisticated browser with a number of search modes. One of the feature requests we received from many users was to specify an input data type and an output data type and have Morpheus find either:

1. A ranked ordering of "connected" compositions of transforms that will produce the output type from the input type.

2. If no "connected" path exists between two types, a ranked order of compositions of transforms such that some of the intermediate nodes are unconnected. Hence, the user can get the transform he wants by a composition of existing transforms if he defines the missing one(s).

We built TransformScout, a search mode within the Morpheus project that can produce these results. In effect, TransfromScout presents the user with the information about how transforms in the system can be connected with one another to produce larger software units. TransformScout's value proposition is that a considerable amount of programming efforts can be eliminated by wiring components automatically, especially in a repository with possibly tens of thousands of components.

This thesis' has two goals. The first goal is to explain existing approaches to data integration and compare them to our approach of software re-use. The second goal is to describe the transform composition problem and describe our implementation of TransformScout.

Chapter 2 provides background information on current approaches to dealing with the data integration problem and compares Morpheus' approach of re-use and sharing with other approaches to data integration. We give an overview of our system, Morpheus, with system goals and implementation details in Chapter 3. Chapter 4 gives a background on related work on the transform composition problem while Chapter 5 explains the design and implementation of TransformScout. We discuss experimental evaluation of Morpheus and TransformScout with a user study in Chapter 6. We wrap up with Chapter 7, which states the conclusions of this thesis.

Chapter 2

Background on Data Integration

This chapter focuses on explaining the data integration problem. We give a definition of the problem and the reasons behind its existence. Next, we describe various approaches to solve the problem. We conclude with a comparison of the existing integration approaches to Morpheus' approach of re-use and share.

2.1 Data Integration Problem

The broad definition of data integration is combining data residing at different sources to conform to a target representation. Databases and other data sources such as websites are an integral part of the information system of different groups of users and organizations. These data sources have typically been designed and developed independently to meet local requirements. An important consequence of the local autonomy and heterogeneity of data sources is that semantically similar pieces of information may have different names and different schemas in disparate data sources [11]. Data integration tasks have to be performed when related information necessary for a global application exist in multiple, incompatible data sources.

The desire to merge heterogeneous representations has been around since the inception of computerized data management systems. The proliferation of service oriented architectures and the Internet will expand the scope of the integration problem from the boundaries of a single organization to between organizations. Moreover, tolerance for latency in inter-enterprise communication is diminishing. Ideally, organizations would like to interact with one another and resolve their semantic conflicts in real time. Yet, no proposal had ever come close to providing a comprehensive solution for the data integration problem.

2.2 Related Work

A popular data integration practice involves agreeing on a common, global schema and providing a uniform query interface over this common schema. A substantial body of work on data integration is about the architecture of this kind of integration and answering a mediated query over a set of distributed heterogeneous data sources efficiently. This line of research is called logic-based data integration. It covers the majority of theoretical work on data integration and it deals with classic database theory problems such as query containment and query reformulation. [8]

An integration system needs to know the relationships between the data sources and the target representation in order to answer queries from the mediated schema. Two prominent approaches to establishing these relationships are the Global As View (GAV) and the Local As View (LAV) approaches. In the GAV approach, the mediated schema is designed to be a view over the data sources. The burden of complexity is placed on implementing the mediator code that instructs the data integration system how to retrieve elements from the source databases [12, 13, 14]. In the LAV approach, a query over the mediated schema is transformed into specialized queries over the source databases [15, 16]. The benefit of an LAV modeling is that new sources can be added with far less work than in a GAV system. Hence, the LAV approach is preferred in cases where the mediated schema is not likely to change. The advantage of the GAV approach is that query formulation is easier because it simply consists of replacing each atom of the query by its definition in terms of the relations of the sources' schemas. In the LAV approach, since both queries and the views describing the data sources are expressed in terms of the mediated schema, the reformulation of the queries in terms of the source relations cannot be done as a simple query unfolding as in the GAV approach [17].

A considerable body of work on data integration is involved with resolving semantic conflicts among heterogeneous sources. The most prominent approaches in this work include schema matching and ontology-based techniques.

Schema matching refers to the problem of identifying corresponding elements in different schemas. Important approaches to automatic schema matching include structural matching and learning-based matching techniques.

Structural matching considers only the structure of database objects, e.g., fields, names, integrity constraints, and types of objects [18, 19, 20]. The structural matching approach is relatively inexpensive and unlike learning-based techniques, they do not require training. Structural matching systems do not look at every instance of a database object, i.e. data content, to determine the similarity between two objects. Therefore, they are relatively fast.

Learning-based schema matching leverages both structure information and instances of database objects [21, 22]. They are valuable in situations where it is not possible to formulate explicit rules to create mappings between representations. In addition, they learn from previous matches in the system and improve the quality of their matches over time. Since both structural matching and learning-based techniques have their own disadvantages, a natural response from the research community is to combine matching techniques from both approaches to build robust matching systems [23, 24].

As an alternative to schema-matching techniques, some researchers from the Semantic Web community propose ontology-based approaches. Ontologies explicitly define schema terms. The vision of the Semantic Web is to have common ontologies and languages (RDF, OWL, WSDL) for integration and combination of data drawn from different sources [25].

One of the approaches in ontology-based techniques assumes the presence of shared ontologies. Matching in this scenario involves developers of disparate ontologies defining mappings to a shared, larger ontology. Agreement on common ontologies such as SUMO [26] and Dolce [27] is rather inconclusive [7]. The second approach in ontology-based integration discards the common ontology idea. Instead, this line of research tries to match incompatible ontologies automatically [28, 29]. Automatic matching of ontologies is very similar to automatic schema-matching. The difference is that ontology-based system incorporate formal semantics to represent entities, which makes automatic inference of relationships between ontologies somewhat easier than the task of matching schemas. On the other hand, utilizing formal semantics has a number of drawbacks. First of all, it is difficult to agree on formal semantics to describe entities. Second, the burden of annotating objects with domain knowledge still rests on the user.

2.3 Solutions from the Industry

On the industry side, a number of product categories were developed to deal with the problem of integrating heterogeneous systems. In contrast to the approaches trying to integrate the disparate sources, some of the other offerings, such as ETL tools, transform data from a source representation to a target representation.

Enterprise Application Integration (EAI): EAI is defined as the integration of various applications so that they may share processes and data freely [30]. EAI systems try to enable data propagation and business process execution throughout the various distinct networked applications as if it were a global application. It is a distributed transactional approach and the emphasis is on supporting operational business functions such as taking an order, generating an invoice, and shipping a product [31]. In addition to linking business processes across applications EAI can be used for data integration. EAI systems strive to keep data in multiple systems in the same format and with the same semantics.

The limitation with this kind of integration is two fold (1) Enterprises still need to interact with other organizations with different representations than the EAI applications' proprietary format and semantics, (2) A one-size-fits all approach does not necessarily capture all the details of a business process even within a single enterprise. EAI vendors typically sell companies a suite of off-the-shelf modules that are compatible with each other. These modules are further specialized with a particular industry in mind. For example, a manufacturing company would get a solution that focuses on the needs of a generic manufacturing company. In practice, companies always need substantial amount of customization to suit these modules to their local needs because of the diversity in business practices. The different modules in the system which could be linked with one another cannot do so after the customization. A second problem is that enterprises use a variety of other tools and legacy systems that cannot be fully integrated within the EAI framework. A substantial amount of data integration efforts need to be taken to connect legacy systems to EAI systems.

Data warehousing and ETL tools: Data warehousing systems archive data from operational systems in a centralized repository for business analytics purposes. ETL (Extract-Transform-Load) systems transform large amount of data from operational systems to a data warehouse's global schema. In contrast to other solutions that we are describing in this section, ETL tools do not try to integrate disparate data sources. Instead, they transform the data coming from different sources according to the global schema in the data warehouse.

In contrast to EII (Enterprise Information Integration) tools, which reflect changes from sources in real time, a data warehouse would still contain the older data when a change in a data source occurs. The ETL process would need to be executed again to reflect the changes.

Enterprise Information Integration (EII): EII tools address the data integration problem by considering external sources as materialized views over a virtual mediated schema. Wrappers are designed for each data source that transform the query results from external sources into an easily processed form for the data integration solution. When a user queries the mediated schema, the data integration solution transforms this query into appropriate queries over the respective data sources. Finally, the results of all these queries are merged together to answer the user's query. Compared to data warehousing, query results reflect changes in real time. Querying a distributed set of sources with different data models and varying response times makes EII a more challenging task than the offline transformation phase (ETL) in the data warehousing systems. Handling distributed queries that access sources with different data models makes the optimization of the global query very complex since different data models have different strengths with respect to their data access and query processing patterns.

In the previous sections, we described several proposals that have been developed to address the data integration problem. There is a number of other integration approaches as well as issues such as Quality of Service (QoS) requirements [32], security [33] and data cleansing [34]. We conclude this chapter by comparing Morpheus' approach of re-use and sharing with the other approaches to data integration.

2.4 Comparison of Morpheus with the Other Data Integration Approaches

Shared World vs. a Diverse World

Research on the semantic web, shared languages [35], shared architectures [36] and standards [37] propose to resolve conflicts by sharing languages and dictionaries. We believe that shared languages and data models can help information systems to interoperate seamlessly at the infrastructure layer. Two things limit the success of shared world approach for resolving semantic conflicts. First, there are too many standards even within a single industry or an application domain, e.g., there are at least 5 different XML standards for exchanging financial data [38, 39, 40, 41, 42]. Second, standards and ontologies are unlikely to cover all possible variations of semantics even within a modestly rich application domain.

Match-centric vs. Transformation-centric

Both schema matching and ontology-based integration focus on determining whether two schemas or two ontologies can be matched with one another automatically. In contrast, the goal of both Morpheus and ETL tools is to transform source information into target information. Both rely on human users to resolve the semantic conflicts and to write the mappings between conflicting representations.

A matching approach with a similar emphasis on transformations is the complex matching approach. Complex matching systems try to determine whether a data type is equivalent to an another type that had a transformation applied to it. As an example, consider a "name" data type with "first name, last name" representation. A complex matching system will try to verify that a "name" type is equivalent to a "full name" type with representation "last name, first name" after a re-arranger transformation has been applied to it. A number of complex matching systems have been developed in the past decade [43, 44]. This approach along with approaches such as COIN [45] that aim to find the correct transformation automatically only work for fairly simple transformations. They are unlikely to succeed in complex transformation tasks.

The purpose of the Morpheus is to address the needs of transformation-centric data integration tasks better than ETL systems. Morpheus' key advantage over ETL tools is its emphasis on the management of re-usable and share-able transformations. In addition, Morpheus complements the technologies in match-centric systems. Specifying that a concept matches another concept is simply a lookup transformation that can be easily incorporated into Morpheus. Morpheus provides tools to assist a human in constructing and re-using transformations efficiently, instead of automating the process of resolving heterogeneities.

Chapter 3

Morpheus

This chapter describes the design of Morpheus. We give an overview of the system, followed by sections on each layer of Morpheus. We explain each layer with the system goals, the design decisions made to achieve these goals and the details about the implementation.

3.1 System Goals

Our basic goals for the Morpheus project are two-fold:

1. Make it easy to write transforms

2. Make it easy to find and reuse transforms written by others

We built a high-level, workflow programming environment and a searchable repository to store these transforms.

We made some changes to the system based on the user feedback we received by demoing the system and the workshop experience in SIGMOD2006 [46]. We introduced more automation into the process of locating and re-using transforms, in contrast to labor-intensive, manual construction of transforms in the first version of Morpheus. We added a crawler that can scour the Internet and local networks to find and register new transforms. While automation helps with locating and registering a large number of transforms in a fraction of the time required for manual construction, the growing size of the repository can become a concern for locating a desired transform among a large number of components. Hence, a considerable amount of our effort went into improving the browsing and the search experience.

3.2 System Design

Figure 1 is a diagram of the modules of the system.

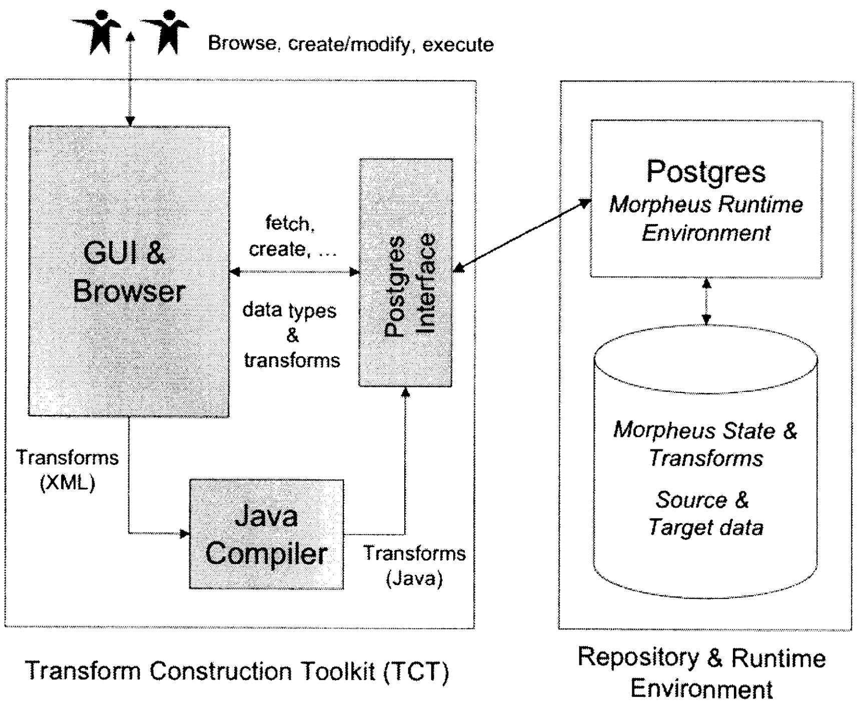

Browse, create/modify, execute
GUI & Browser
fetch, create, ...
data types & transforms
Postgres Interface
Transforms (XML)
Java Compiler
Transforms (Java)
Transform Construction Toolkit (TCT)
Postgres
Morpheus Runtime Environment
Morpheus State & Transforms
Source & Target data
Repository & Runtime Environment

Figure 3-1: System Architecture of Morpheus

A human user interacts with Morpheus through our *GUI*. Interacting with this interface, the user can find existing Morpheus objects, data types and transforms, and modify them or create new ones.

The *Browser* helps the user to locate existing objects. The *GUI &Browser* layer allows the user to perform various types of searches to find objects registered in the system. To create a new object or modify an existing object, the user interacts with

TCT (Transform Construction Tool) and its companion *DTT (Data Type Construction Tool)*. Both TCT and DTT are workflow programming environments where users can build transformations with high level building blocks. Both TCT and DTT produce XML representations of transforms and data types. The *Java Compiler* translates the underlying XML representation of a TCT program to platform independent Java source code. Upon creation of a new object, the system generates some metadata about the object automatically while other metadata must be entered by the user. These metadata form the input to the various searches that can be later performed on Morpheus objects.

A *Postgres DBMS* supplies both the storage and runtime environment for Morpheus. On the storage side, data types are stored as Postgres' user-defined types. Transforms are registered as Java procedures ready to be executed within the DBMS.

Interaction between the user-centric components of Morpheus (GUI, Browser, TCT) and the Repository & Runtime Environment is facilitated by the PostgreSQL interface via a standard JDBC bridge.

3.3 Individual System Layers

3.3.1 GUI and Browser

System Goals and Design

The unifying goal that shaped the design decisions for the Morpheus' browser is to help users locate re-usable components easily. We present the sub-goals, their rationales and how these goals affected our design and implementation.

Goal: To browse nearby objects easily

Browsing nearby components means searching for components that are close to a component with respect to some dimension. For example, if two objects are under the same category in a classification hierarchy, they are close to each other with respect to a category dimension.

Rationale: Searching for software components is not a process with a single, definite answer. Therefore, it is desirable to browse nearby objects once a subset of possibly desirable components is located.

Implementation: In the first version of Morpheus, we let the user enter the repository from a starting point. The user can start with a keyword search or by selecting a category at the root of a classification hierarchy. The result is a list of one or more data types or transforms from which the user can select the current focus for subsequent searching.

If a user clicks on a category, the Morpheus objects, which are nearby in that category, appear on the screen. A user can click on an object to change his current focus, and the cycle repeats. Once an interesting object is identified, Morpheus allows the client to zoom into the object with increasing amounts of detail.

Below are the kinds of search supported for finding transforms and data types in Morpheus 1.0:

Keyword: Users can perform keyword searches on the names and the textual descriptions of data types and transforms. The keyword search in Morpheus is performed by Lucene, a text search engine library [1].

Category: Transforms and data types can be assigned to a number of categories in the category hierarchy of Morpheus. Users can browse this hierarchy to find objects under a specified category.

Uses-relationships: A-Uses-B is a way of search where a user can look for the components in the system being used by other components. For example, with this feature, given a transform, we can find the input and output types used by this transform.

Goal: To provide a multi-faceted search whereby a user can specify a multitude of search criteria at once

Rationale: The first version of Morpheus's browsing model allowed a user to wander around objects nearby to another object in a number of dimensions such as category and textual similarity. However, user feedback suggested that this model was not desirable. Users indicated they already know various pieces of information about

[1]Apache Lucene, http://lucene.apache.org/java/docs/.

the transform they would like to find. For example, they might know it was written by someone in Accounting Department, under the category of Payroll Processing, and contains "salary" in its text.

Design: This search requirement is best supported by allowing a user to specify search criteria in multiple dimensions at once. Hence, we changed our search paradigm to multi-faceted search in Morpheus 2.0. In the new system, users can specify a multitude of criteria to perform a search.

Goal: If a search yields too many candidates, allow the user to refine the search.

Design & Implementation: To allow subsequent refinement to a search, we need to store *result sets* of previous queries. We provided support for union and intersection operations on the results sets so that they can be combined together.

Since every search in Morpheus is converted into a Postgres query, we simply store the query that corresponds to every result set. Then, we can combine the result sets with the AND operator or intersect them with the OR operator.

Goal: To facilitate search on lineage information

Lineage information in a repository describes which objects are derived from one another.

Rationale: One of the feature requests we received from our interaction with industry was to allow the system to search for lineage relationships. Having such a feature helps the user find similar objects for re-use once an object is selected as the focus. Furthermore, re-use is a metric for software practice assessment in many companies. Storing lineage relationships would aid in the assessment of how often re-use happens in an organization.

Implementation: In Morpheus, new types and transforms can be derived from existing types and transforms. When a user modifies a seed transform and registers it as a new transform, the system establishes the newly created transform as a descendant of the seed transform. A similar process applies to the creation of derived data types.

3.3.2 Transform Construction Tool (TCT)

System Goals and Design

Goal: *To allow the non programmer to write and modify transformations easily*

Rationale: Morpheus is designed with common data integration tasks in mind. These tasks typically range from very simple to complicated programming tasks. Yet, often the people who understand the organizational logic are not programmers themselves. Hence, we need to create an environment where even the non-programmer can construct transformations in a way that is easier than using conventional programming languages.

Design & Implementation: The Transform Construction Tool is a graphical workflow programming environment. We supply the user with a number of high level building blocks that can be wired together in a composite workflow to accomplish a given transformation task. We have built a compiler for this workflow language, which generates Java source code.

The workflow primitives in the current Morpheus prototype include the following:

- A *computation* primitive, which performs standard numeric transformations.
- A *control* primitive, which supports if-then-else statements
- A *lookup table* primitive, which maps an object in one field to an object in another field (e.g., letter grade to numeric grade).
- A *macro* primitive which allows a user to group several boxes together (inside a larger box) as part of a high-level transform (macro). This is also a way to hide some of the details of a transformation.
- *Postgres user-defined functions*, so transforms can be composed out of existing ones.
- A *predicate* primitive, which maps sets (ranges) of values to related sets (e.g., University class status such as freshman to units taken, '>0', '>35', and so on) based on a user-defined predicate.

- A *re-arranger*, which supports creation of transforms by allowing the user to visually manipulate characters and strings.

- A *wrapper* primitive for external call-outs (e.g., to web services).

The current prototype allows a user to start either with a blank workspace or with an existing transformation from the repository. He can then edit the workspace, adding, deleting and rewiring objects of interest. When he is satisfied, he can register the transform in the Morpheus repository. In addition, there is a testing environment in the system. Users can test the transforms they created in this environment by entering input values and then they can inspect the results returned by the transform to ascertain that it is working correctly.

Goal: To allow the user to re-use code from various resources

Design & Implementation : Transforms created in Morpheus are stored in two different representations. The first is an XML representation that corresponds to the diagram of a transform. The second representation is a Java function that can be executed efficiently. However, there is no requirement that transforms be written using the TCT. Java code can be written, compiled outside of the system, directly entered into the database, and then included in a Morpheus transform. Moreover, we expect many transforms to be available as web services. This extends the system to any programming language code as long as it is wrapped as a web service.

3.3.3 Postgres as Repository and Execution Environment

Goal: Support storage for objects in a way that enhances re-use

Design & Implementation: The Postgres DBMS has novel features that simplify Morpheus' way of representing re-usable components. First, we can use user-defined complex types in Postgres to represent complex data types in our system. Second, Postgres allows type inheritance. Hence, representation of lineage relationships in Morpheus can be supported by Postgres' typing system. Having both features built-in to the DBMS greatly simplifies the code base for Morpheus.

On the other hand, these features are not standard SQL, which can limit Morpheus' extensibility to other repository structures. To avoid this trap, we store the Posgtres specific functionality redundantly. We store the schema of complex types and inheritance relationships in tables that can be easily transferred to any other relational system.

Goal: To simplify maintenance of the ETL cycle by pushing the application logic to the DBMS

Rationale: In contrast to ETL vendors who keep the transformation task out of the DBMS, we push down the application logic to the internals of the database. This approach benefits us in two important ways. First of all, we reduce the maintenance cost of keeping more layers in the system architecture. Secondly, we can perform bulk transformations inside the DBMS where transaction management and powerful queries can be leveraged.

Design & Implementation: Transforms are registered as stored procedures in Postgres. Input and output types of a transform correspond to tables in Postgres. Running a transform within the DBMS is equivalent to running a query that invokes a stored procedure.

Metadata Stored In Repository

All Morpheus information is stored in Postgres tables. The repository holds all Morpheus objects: transforms and data types, look-up tables and input tables.

Transforms can be Morpheus transforms, created in the Transform Construction Tool. Alternatively, they can be any Java method or a method in any other language as long as the method is wrapped as a web service.

Data types are entities that supply the input and output parameters to transforms. Data types may have more than one representation.

We store the following information for both transforms and data types: the author of the object, the data it was constructed, a textual description of the semantics of the object, the position in a classification hierarchy. For transforms, we store the data type it accepts as input and the data type it produces as output as well the original language of the transform.

In addition to types and transforms, we have auxiliary structures called look-up tables which describe mappings between two values. They can be user-defined or derived from web services. We also store user tables, which hold the actual contents of data to be transformed.

We also store our classification hierarchy, which is a subset of DMOZ [2] hierarchy. Domains within this hierarchy helps Morpheus categorize objects: data types, transforms, and input tables. There is a many-to-many relationship between domains and Morpheus objects; several objects may be classified under a single domain and an object may belong to more than one domain.

Finally, we store information about the users in the system. Users register themselves in the Morpheus system. They may have an additional author role upon creation of a transform/data type. Users can store result sets of their previous searches with named queries, as explained in our discussion of GUI & Browser in section 3.3.1.

3.3.4 Crawler

We end our discussion on Morpheus with an overview of the Crawler module. It has been included in Morpheus version 2.0.

Goal: To reduce the cost of finding and registering transforms

We have designed and built a crawler with the goal of increasing automation in our system. The crawler can look for transforms either inside an enterprise or across the Internet.

In the next section, we talk about the proposed functionality of the crawler. With the exception of the WSDL mode, these features have not been built yet.

Proposed Functionality: The crawler looks for websites with source code, web forms as well as web services. Within the intranet of an enterprise, the crawler looks for the source code files and available web services.

For source code files, the crawler parses the input and output data type specifications, and looks for comments in the code that describe the functionality of a method.

[2]ODP - Open Directory Project, http://www.dmoz.org

The file is downloaded and compilation is attempted. If successful, each method is identified as a candidate registration with Morpheus.

In web services mode, the crawler looks for WSDL files, which describe interfaces to different transformations made available by a web service. Web services publish their input and output types, address information for locating the service and binding information about the transport protocol to be used to interact with them. Since they have comprehensive interfaces, it is relatively easy to extract metadata from them and register them in Morpheus.

For web forms, the crawler tries to extract the functionality of a form from the contents of the HTML document. The input and output types of such a transform are inferred from the input and output fields of forms. This task is diffcult since the input and output fields within HTML forms do not conform to a standard.

After searching the network for these three types of transforms -web services, web forms, and methods in source code-, the users need to figure out if these transforms work correctly and conform to their extracted specification. For source code, we use an automatic unit testing tool such as [47], to create mock values to test methods. For web services, we ask the user to specify values based on the interface description of a service. Then, external calls to the web services are made through the SOAP protocol. The procedure is similar in web forms, with the exception that the external calls are made through the HTTP POST interface. A user can ascertain the usefulness of a transform by inspecting the values returned by a transform. Finally, users can go ahead with the manual registration of these transforms with the help of the metadata output by the crawler.

Chapter 4

Related Work on Transform Composition

Applications in Morpheus are compositions of "black box" processes, i.e., transforms that exchange data across predefined connections, i.e, through data types. These black box processes can be reconnected endlessly to form different applications without having to be changed internally.

Transform composition is the process of combining transforms together to form larger programs. Transform composition search can be defined as searching for compositions of components that form the mapping from a start state, i.e., an input type to a goal state, i.e., an output type. Since transforms are simply software components, transform composition search in TransformScout can be likened to the work on composition for re-usable software components in the software engineering field.

In this chapter, we give a brief overview of research on component search to explain the search capabilities of Morpheus and TransformScout in the context of the related work. Next, we focus on signature matching systems that search for software components in a similar vein to TransformScout. Recent research on component composition came from the database community, with the advent of web services. We conclude this chapter with an overview of research in web services composition.

4.1 Software Component Search: An Overview

A component retrieval mechanism works in the following way: When faced with a programming problem, the user understands the problem in his own way, and then formulates a query through the search interface to find a solution. The complexity of the search interface can range from a keyword-based search to complex specifications in a formal language. In practice, formulating a query may result in loss of information since the user is not always capable of exactly comprehending the programming task or encoding it in the query language. The user has to understand what the component repository is like. Then, he needs to locate which components in the repository are suitable for his needs. This is in contrast to structuring a program from scratch where search and retrieval steps are omitted.

Before it can be retrieved, the information about components must be encoded. The encoding process is called indexing, which can be either manual or automatic. The search itself consists of comparing the query with the indexes and returning the components that match the query.

Component search techniques differ with respect to how components are indexed by the system. With information retrieval (IR) techniques, the goal is to automatically classify components. The key terms from documentation and source code are extracted automatically to construct indexes. The advantage of IR systems is that they are relatively inexpensive. The disadvantage is that variable and method names in source code do not necessarily produce useful indicators of the intent of a program. Furthermore, in real settings, documentation is often missing, incomplete or inaccurate.

In contrast to IR techniques, Artificial Intelligence (AI) based approaches try to understand the semantics of queries and the functionality of components. These systems try to infer what a program does from how it does a task. In addition, they try to estimate the intent of the user from the queries he poses and tailor the results accordingly. Unlike pure IR based techniques, AI systems utilize domain knowledge and rely on human input to understand what a program does. The advantage of this approach is that it can come up with more "clever" answers than IR systems. The obvious drawback is that it is more costly than an IR system where indices are created automatically. Furthermore, domain specific information may limit the generalizability of the approach. New domain knowledge may need to be input each time components from a new domain are added to the system. Finally, scalability may be poor because indexes are not automatically constructed.

Both TransformScout and other search modes in Morpheus are a mix of IR and AI component search systems. Like IR systems, we use automatically extracted features of a document in the context of keyword-based search. We also provide semantically richer search capabilities in the vein of AI systems. For example, in TransformScout, we make inferences based on metadata input from a user to rank components. In addition, TransformScout learns from the user's interactions with the system to improve query results. For example, we give a higher ranking to results that were previously chosen by users.

A second issue in software component search is producing exact versus relaxed component matches. Exact matches conform precisely to constraints in a user query; components returned in a result set may be restricted by input-output types, standards, languages and parameters. With relaxed matches the user is given the option to locate inexact results and see nearby objects that are similar to another object with respect to a dimension such as textual similarity. Literature suggests that exact search is useful only in a limited number of application scenarios, such as in reactive and real time systems [48]. We also note that for some specific programming problems, tools such as Prospector [49] have to produce exact matches. For example, when Prospector searches for possible compositions in the standard Java library, exact compositions of objects are necessary for avoiding compilation errors. Other than the specific contexts where exact matches are necessary, the consensus in the research community is to present the user with relaxed matches [50].

4.2 Signature Matching Systems

Signature matching systems find reusable units such as functions that match a signature, i.e., input-output types of a function. Some of the signature matching systems provide both exact matches and relaxed matches with minor variations such as the order of arguments to a function [51], [52]. A second line of research is specification matching, which further restricts the search space of possible query results with pre-conditions and post-conditions of using a component. Notable examples in this area include Zaremski's [53] specification matching system using Larch provers [54], Inquire [55], and Rollins and Wing's [56] Prolog based component retrieval system.

The majority of the prior work on signature matching systems shares the use of a query language and a system that finds complete units, such as functions or software libraries, that match a query. The difference of TransformScout with respect to other signature matching systems is that we do not have a complex query language. In TransformScout, users specify queries by selecting an input type and an output type, similar to a function signature. In addition, we have an elaborate ranking scheme based on software performance, software quality and user ratings of components and query results.

Signature matching systems focus on helping with difficulties in selection [57], "where the user knows what he wants to do but does not know which component to select". Our system utilizes various browse and search modes in Morpheus to pick the initial types to compose, i.e., an input type and an output type. We then present the user with larger units (collections of composite transforms). We also help with coordination barriers, "where the user knows what units to use, but does not how to make them work together" [57]. These barriers are common in large repositories with several components.

Compositions of components can be likened to program dependence graphs (PDGs), because a composition of transforms is like a single data flow path from a PDG. In TransformScout compositions of transforms have a linear shape whereas PDGs can be general graphs. Hence, PDGs can express some data flow structures that linear compositions cannot. For example, PDGs can express data flow paths where both the input and the output type are the same. In contrast, TransformScout does not support search that will cause cycles. Consider the set of solution compositions for the query (input, output). It corresponds exactly to the set of paths from the input type to output type. Thus, solution compositions can be enumerated by standard graph search algorithms. Note that there may be infinitely many paths from the input type to output type if we allowed a transform to produce the same type for both the input and the output. We limit our search to acyclic paths, which are finite in number. This is a limitation of our system that we plan to address as future work.

4.3 Composition of Web Services

A key promise of web services to provide software components within the flexible interface of the Internet. One of the issues in the web services research is locating desired web services. Once we locate a number of different web services though, we see that most of these components are simple tasks that need to be composed with other components to make any non-trivial task possible.

In the area of composition of web services, two approaches are prevalent. The first one is the workflow approach where a human assists in composing components. In the workflow approach, participants in a web service agree on the interfaces that describe the semantics of a web service. The process description language BPEL4WS [58] is a widely-adopted language for specifying such a workflow. After agreeing on semantics, users at each side of communication can manually wire together individual services in a high-level programming environment, similar to Morpheus' TCT.

A second approach in compositions of web services is an AI planning approach where the process of composing components is automated as much as possible. Because services can diverge from the initial agreements and composition by hand may be too expensive from the user's perspective, it is desirable to automatically detect changes in service descriptions and to automatically compose services together. Several proposals have been presented to describe the semantics of a service in a formal, machine interpretable fashion so that automation is possible.

An important body of the research in formalizing semantics of web services comes from the Semantic Web community. The majority of their techniques use OWL-S [59], the standard framework for specifying process models in the Semantic Web. The process model describes a service in terms of its inputs, outputs, pre-conditions and post-conditions. Researchers have proposed methods for transferring OWL-S descriptions to Prolog [60] and Petri-nets [60] for automating composition. In the Prolog approach, the programmer has to manually translate an OWL-S description to Prolog, which makes it possible to find a plan for composing Web services to arrive at a target description. With the Petri-net approach, an OWL-S description can be automatically translated into Petri-nets. Then, programmers use this specification to automate composition.

Other approaches for automating web service composition include modeling service composition as Mealy machines and automatic composition of finite-state machines (FSMs). Mealy machines [61] are FSMs with input and output. Web services are modeled as Mealy machines (equipped with a queue) and exchange a sequence of messages of given types (called conversations) according to a predefined set of channels [62].

Berardi et. al. [63] present a framework that describes a Web service's behavior as an execution tree and then translates it into an FSM. They propose an algorithm that checks the existence of a composition and returns one if it exists. In the process, the composition is proved correct. In addition, the system will determine the algorithm's computational complexity, ensuring that the automatic composition will finish in a finite number of steps. So far, modeling services as FSMs is the most promising approach for automatic composition of web services.

Another line of research related to composition search in web services is Woogle by Halevy et al. [64]. In Woogle, composition search is limited to understand if two components can be composed together, for example Woogle can determine if the output of a web service is a valid input to another web service. Thus, Woogle provides only a subset of the search capabilities of TransformScout, which can search for full as well as partial compositions between input-output pairs.

We conclude with our observation that transform composition is a hard problem and it is not entirely clear which techniques serve the problem best. Our approach in TransformScout is a semi-automatic one: We assist the users with constructing and reusing transformations by recommending them a ranked collection of composite transforms.

Chapter 5

TransformScout

This chapter presents an overview of TransformScout. We explain the design and the implementation of the system. Next, we describe each subsystem with the goals of the subsystem and the implementation details.

5.1 System Goals

The value proposition of Morpheus is two-fold. Its first benefit is making writing, sharing, and re-using transformations easier. Its second feature is helpful for large repositories containing on the order of hundreds of thousands objects, where it is crucial for users to easily find what they are looking for. Therefore, searching and browsing are fundamental elements of the system along with the Transform Construction Tool.

To support the goals of re-use and sharing, we created a sophisticated browser and a search tool. One of the feature requests we have received by many users was specifying an input and an output data type and letting the system find a composition of transforms that produce the mapping between the two types. To this end, we designed and implemented TransformScout. TransformScout produces mappings between two types. In addition, it provides the user with a helpful building block for constructing the full mapping between two types even when a full composition does not exist.

In its basic form, TransformScout could have been merely another mode of searching and re-using components in the Morpheus repository. However, TransformScout enhances the search capabilities of the whole system with the following goals:

1) Formulating a similarity metric for the components in the repository

Often, there is not a single answer to a user query that tries to solve a programming task. Instead, there may be a number of components that may be approximately suitable for a user's needs. Therefore, an explicit or an implicit "Find similar objects" feature to perform a relaxed search may be very useful. In a relaxed search, the users would be presented with a group of objects that are similar to one another with respect to some dimension such as textual similarity.

2) Formulating a ranking metric for the components in the repository

An important part of information retrieval systems is having a ranking module that will sort all possible results in response to a query. There are multiple rationales for incorporating such a module. Some of the components that satisfy the user's query may have more desirable features than others. For example, some transforms may take longer to execute than other transforms. In addition, the quality of components within a result set may vary. For example, some transforms may have fewer bugs and may conform to the user's intent better than others. For these reasons, we need to match the intent of a query to the intrinsic quality of the individual components.

5.2 System Overview

TransformScout is a component search system within Morpheus. It helps users produce a *target type* (output type) from a *source type* (input type), both already registered in the system.

When an input, output pair is specified, there are a number of different types of compositions that will result in a complete or a partial mapping between the two types:

1. Complete compositions: A collection of compositions of transforms that map *input* to *output* through some collection of intermediate data types.

2. Incomplete compositions: "Partial" compositions of transforms that map from *input* to *output* through a collection of intermediate nodes as above. However, one or more of the transforms along this path has not been defined. Hence, the user can get the transform he wants by a composition of existing transforms if he creates the missing one(s).

TransformScout (TS) produces both complete and incomplete compositions. TS generates these results by searching a graph whose nodes are data types and whose edges are transforms. Each transform has a cost that is determined by the average running time of the associated transform. Finding complete paths is supported by finding all the connected paths from a given input data type to an output data type. If multiple complete paths are found, we first compose stored (input, output) pairs to determine if the multiple paths are semantically identical. If none exist, then we execute the functions to produce (input, output) pairs that can be composed.

The result is a collection of semantically distinct composite transforms. Each composite transform may have multiple implementations (paths from input to output) which Morpheus must rank. This is accomplished by the ranking module which takes into account computational performance and software quality. The first portion of the metric is obtainable using costs (running times) annotated within the transforms' metadata. The second is supported by allowing users to review a transform and rate its quality, as well as report bugs. Searching for an incomplete path uses the same logic discussed for connected paths with the requirement that we also need to take into account the similarity between two ends of a missing link when we rank these compositions.

5.3 System Design

Given the goals and constraints we outlined in 5.1 and 5.2, we have designed the architecture of TransformScout with three main modules: search, similarity, and ranking.

Figure 5.1 shows the architecture of TransformScout. A user interacts with the

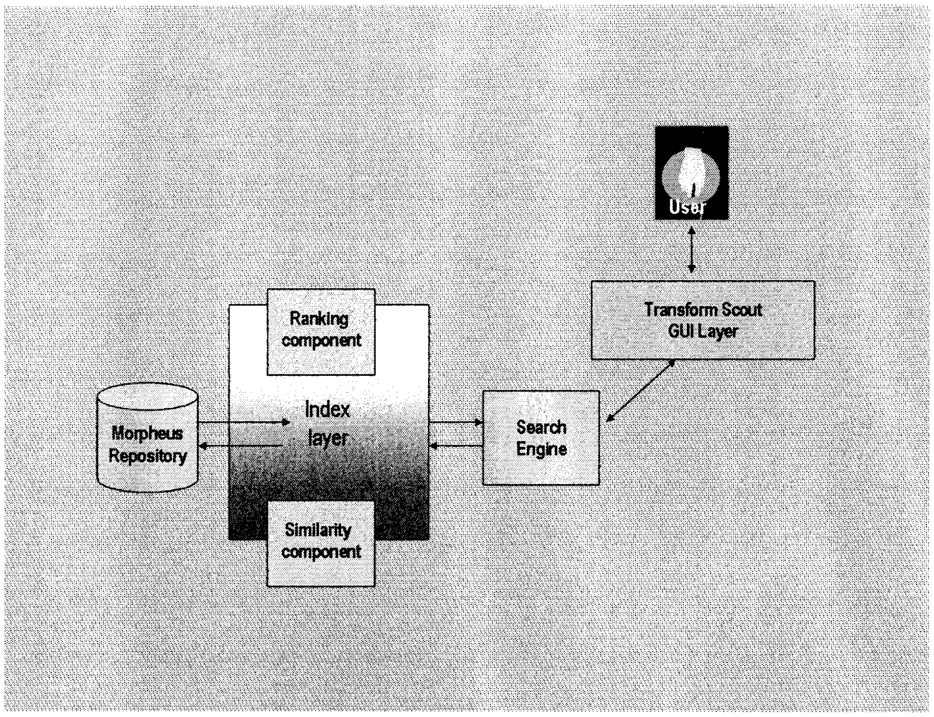


Figure 5-1: System Architecture of TransformScout

GUI Layer of TransformScout to issue a query, specifying an input and an output type as parameters. The Search Engine module interacts with the *Indexer layer* to perform the search. The Indexer layer accesses the *Morpheus Repository*, which stores the types that are used by a transform as inputs and outputs. The Indexer layer employs the *Similarity module* to retrieve values for similarity between two types, and interacts with the *Ranking module* to retrieve ranking scores of individual components.

The *Similarity module* computes and stores the similarity between two types based on semantic, category and type similarity. Since computing similarities between any two types is too computationally expensive to be handled during runtime, we use pre-computed indices.

Users can rate quality of transforms at any point in their interaction with Morpheus. The user's feedback to the query results is the first input to the *Ranking module.* The second input comes from the *log reader*, which computes the average run times of transforms from the log files of Postgres. The third input comes from the popularity of the components used in *compositions* chosen by users in response to a query.

5.4 Similarity Module

The Similarity module estimates the similarity between two data types. We use a similarity metric in TransformScout to support the following features:

1. *As a heuristic for pruning the search space of the graph search algorithm:* We use an A* search algorithm. The A* search algorithm utilizes a heuristic function to prune the search space. A heuristic function estimates the distance from the current search state to the goal state. In this way, the search can prefer nodes that will arrive at the goal state in a shorter distance. In TransformScout, our heuristic function is the similarity between a source and a target data type.

2. *A criteria for ranking incomplete composition results:* Similarity helps us rank compositions of transforms that do not produce a complete mapping from an input type to an output type. We compute the similarity between the target type of the partial composition from the left hand side and the source type of the partial composition from the right hand side. In other words, we compute the similarity between the unconnected nodes in a collection of partial compositions. Our premise is that the more similar the unconnected nodes (data types) are, the easier it will be to construct the missing portions of the path between the source and target types.

Similarity between two data types is estimated with respect to three dimensions: semantic similarity, user-defined category similarity and type similarity.

5.4.1 Semantic Similarity

In our system, semantic similarity specifies if words that make up a data type name and names of fields of data types (for composite types) are similar in meaning.

A widely-used semantic similarity metric between single words is Resnik's [65] semantic similarity metric that utilizes both an ontology and a corpus. The idea underlying Resnik metric is that the similarity between a pair of concepts may be determined by the extent to which they share information. Resnik defined the similarity between two concepts lexicalized in WordNet [66] to be the *information content* of their lowest common denominator (the most specific common subsumer), lcd($c1$; $c2$):

$$simResnik(c1; c2) = -logp(lcd(c1; c2))$$

where $p(c)$ is the probability of encountering an instance of a synset (concept) c in some specific corpus.

Jiang and Conrath's [67] approach also uses the notion of information content, but in the form of the conditional probability of encountering an instance of a child-concept given an instance of a parent-concept. Thus, both the information content of the two nodes, as well as that of their most specific subsumer, are taken into account. Notice that Jiang and Conrath's semantic metric measures semantic distance rather than similarity:

$$distJC(c1; c2) = 2log(p(lcd(c1; c2))) - (log(p(c1)) + log(p(c2)))$$

Lin's [68] similarity measure follows from his theory of similarity between arbitrary objects. It uses the same elements as Jiang and Conrath but with a different configuration. The semantic similarity between two concepts is defined as:

$$simLin(c1; c2) = \frac{2log(p(lcd(c1; c2)))}{(log(p(c1)) + log(p(c2)))}$$

All of these measures are based on an information theory treatment of distances between the categories that a concept falls under. The intuition is as follows. In an ontology like WordNet, if the common most specific category of two concepts (or words) is specific enough, the two concepts are likely to be closely related. For example, a car and a cat fall under the same concept "entity". A dog also falls under "entity" but the lowest common concept a cat and a dog share is "mammals". Since "mammals" is a more specific concept compared to "entity", we view cats and dogs to have higher similarity than cars and cats.

We chose Jiang and Conrath's measure because it gave the best experimental results among the five most prominent similarity measures utilizing WordNet [69]. Similarity between two concepts is the inverse of the semantic distance. Therefore, we can define JCSim as:

$$JCSim(c1;c2) = 1/(distJC(c1;c2) + 1)$$

1 is added to distJC to avoid infinity values, since distJC(c, c) = 0.

We use the average best similarity calculation in [23] to compute the semantic similarity of two data types:

$$ss(T_1, T_2) = \frac{\sum\limits_{t_1 \epsilon T_1} [\max\limits_{t_2 \epsilon T_2} JCSim(t_1, t_2)] + \sum\limits_{t_2 \epsilon T_2} [\max\limits_{t_1 \epsilon T_1} JCSim(t_1, t_2)]}{|T_1| + |T_2|}$$

The semantic similarity (ss) of two data types with two sets of name tokens T1 and T2 is the average of the best similarity of each token with a token in the other set.

In addition to WordNet, the system is open to a domain synonym dictionary. We can specify our custom, domain-specific subsumption relationships in this dictionary. This is necessary to compute similarities between words that are not included in WordNet. Examples of such words are abbreviations and proper words.

5.4.2 User-Defined Category Similarity

The user-defined category similarity specifies whether two data types fall under similar categories in a user-defined concept hierarchy.

In the previous section, we described a WordNet-based semantic similarity of data types. We introduce user-defined category similarity metric to make a robust similarity module. This way, users have the flexibility to increase the similarity of two data types even if the semantic similarity between them is weak or vice versa. For example, two data types may be semantically different yet the individual words within the name of the data types may be very similar to one another. Consider a data type named "produce yield" that is about agriculture and a second data type "production capacity" that describes the production capacity of a plant. WordNet based similarity would be high between these data types because the similarity of tokens that make up the names of the data types is high. In such a case, we can decrease the overall similarity between these two types by assigning them to distinct categories in the classification hierarchy.

The similarity between two categories, CatSim, is defined as the inverse of the length of the path, LenCat, between the two categories.

$$CatSim(c1; c2) = 1/(LenCat(c1; c2) + 1)$$

1 is added to LenCat to avoid infinity values, since CatSim(c, c) = 0.

The user-defined categorical similarity between two types is defined as:

$$us(C_1, C_2) = \frac{\overset{\sum}{c_1 \epsilon C_1} [\overset{\max}{c_2 \epsilon C_2} CatSim(c_1, c_2)] + \overset{\sum}{c_2 \epsilon C_2} [\overset{\max}{c_1 \epsilon C_1} CatSim(c_1, c_2)]}{|C_1| + |C_2|}$$

The user-defined category similarity (us) of two data types with two sets of categories C1 and C2 is the average of the best similarity of each category with a category in the other set.

The user-defined category similarity computes similarity of two concepts based on the distance between two categories in the classification hierarchy instead of the concept similarity metric of Jiang and Conrath. Although we could have used the measure we have previously used for computing the semantic similarity, we created a distance-based metric for the following reasons. Unlike complete classification for English words under WordNet, users are not required to register objects under our category hierarchy. Furthermore, our classification hierarchy is quite shallow, with average depth less than 6. Researchers suggest [70] that information-theoretic distance between two concepts relies on "the existence of sufficient annotation with ontological terms and sufficient depth of the term for a significant correlation". In other words, a shallow ontology or shallow labeling may not provide enough discrimination. Therefore these metrics are not suitable for shallow ontologies like our user-defined category hierarchy.

5.4.3 Type Similarity

Type similarity specifies whether two types share similar subtypes (if compared types are composite types) or base types. If both data types are simple data types or composite data types with a single field, the system looks up the base type similarity, i.e., Bts, from a similarity table that is prepared by a human expert. In this table, base data types are further categorized into more specific categories such as String types, Double Numerical types, Discrete Numerical types, Date types and Boolean types. For example, a base type under the Double Numerical Type category is more similar to Discrete Numerical types than Date types.

If one or more of the data types are composite data types with a multitude of subfields and one is not contained in the other, type similarity is defined as the average type similarity of subfields of data types:

The user-defined categorical similarity between two types is defined as:

$$ts(S_1, S_2) = \frac{\sum_{s_1 \epsilon S_1} [\max_{s_2 \epsilon S_2} Bts(s_1, s_2)] + \sum_{s_2 \epsilon S_2} [\max_{s_1 \epsilon S_1} Bts(s_1, s_2)]}{|S_1| + |S_2|}$$

The type similarity (ts) of two data types with two sets of subfields' types S1 and S2 is the average of the best similarity of each subfield type with a subfield type in the other set.

5.4.4 Composite Similarity

Given the semantic, categorical and type similarities outlined above, our composite metric of similarity is simply a linear combination of the similarity scores across these dimensions. Given types d_1 and d_2 , their composite similarity score is computed as follows:

$$CompSim(d_1, d_2) = (w_{1*}ssc(d_1, d_2)) + (w_2 * csc(d1, d2)) + (w_3 * ts(d_1, d_2))$$

where ssc(d1,d2) is the semantic similarity score, csc (d_1, d_2) is the categorical similarity score and ts (d_1, d_2) is type similarity score for types d_1, d_2. The sum of the constant weights w_i in the formula is equal to 1 and the weights are the same, 0.33.

5.5 Ranking Module

The ranking modules of search engines score results along two dimensions. The first dimension is the relevance to the query at hand. The second dimension is the query-independent goodness of the search result.

In TransformScout, system relevance to the query at hand is assessed as follows:

If there is more than one complete path, a linear combination of user rating and runtime performance scores is computed to rank these compositions of transforms. The rank of a composition c is computed as follows:

$$r(c) = (w_1 \quad * \quad rp(c)) + (w_2 \quad * \quad u(c)) + (w_3 \quad * \quad qr(c))$$

where rp(c) is the average performance score of transformations in the composition c, u(c) = average user ratings of transforms in the composition, and qr(c) is the average of the user ratings of this composition with respect to the query. The sum of the constant weights w_i in the formula is equal to 1 and the weights are the same, 0.33.

5.5.1 Runtime Performance

Runtime performance of a transform t is computed as:

$$rt(t) = 1/art(t)$$

where art(t) is the average runtime of a transform t.

Transforms are stored procedures within Postgres. Therefore, we can easily retrieve from Postgres log files which transforms were run and how long they took to run. We have a log reader that updates the runtime of each transform periodically.

5.5.2 User Rating of Quality of Transforms:

Users can rate the quality of transforms within any part of Morpheus by choosing a transform and selecting "Rate This Transform" option in the main toolbar. User ratings are limited to a scale of 1-4. The system keeps track of average ratings for each transform which constitutes a user-defined quality metric.

5.5.3 User Rating of Relevance to Query:

TransformScout stores the results picked by the user in response to their queries (input/output pairs they specified). When the same query is requested, preference is given to the results that were chosen by users before.

5.6 Search Module

The search module performs the actual search between two types. We use a bi-directional A* search. A* search is a graph search algorithm which attempts to find the optimal path between a given start state and a goal state. The graph being searched is defined by a function which returns the set of states directly reachable from a state s. The algorithm maintains a set OPEN of states that might lie on a path to some goal state. At the start of the search, the OPEN set contains only the start state. At each subsequent step of the algorithm, a single state is removed from the OPEN set; in particular, the state s that is "best" according to a heuristic function h(s) is removed from OPEN. If s is a goal state, then this state is output. Otherwise all children of s are added to the OPEN set. The search continues until all goal states have been output or the search space is exhausted.

For a forward A* search, at each step of search, priority is assigned to paths that will maximize h(x) + g(x). The term g(x) is the cost of the path so far, h(x) is the heuristic estimate of the minimal cost to reach the goal from x.

A bi-directional A* search is different from a forward A * search because it runs two simultaneous searches: one forward from the initial state, and one backward from the goal, and the goal state is the meeting of the two searches in the middle. Hence, at each step, we prefer nodes that will maximize g(x) + h(x ,y) + g(y) instead of the sum of h(x) and g(x). Our priority is determined by (1) how similar two data types at each side of the search, i.e., h(x,y) (2) the actual cost of getting to a node x from the start node, i.e., the input type, g(x) (3) the actual cost of getting to a node y from the output type, i.e., g(y).

We use a front-to-front variation of a bi-directional A* search. At each step we look at edges emanating from the left hand side of the search (from the source data type) and then switch to looking edges emanating from the right (from the target data type). We chose a bi-directional search because of the nature of incomplete compositions. If we chose forward search, we would miss possible compositions emanating from the output type if a complete composition does not exist. A similar argument applies to backward A* search.

5.7 Indexer Layer

At each registration of a data type, a background process runs behind Morpheus that computes the similarities between the newly-registered data type to all other data types in the system. In addition, similarity metrics may change over time. For example, users may assign a data type to a new category. Alternatively, they can modify categories or field names of data types. For these reasons, we need the Indexer Layer to make a periodic update of metadata indexed by TransformScout.

5.8 GUI

TransformScout has a simple user interface with a list of input and output types as the query interface. Figure 5.2 shows the opening screen of TransformScout. The user can specify an input-output pair by selecting each from the combo boxes on the right side of the screen. Then, he can press the "Go" button, which will populate the empty space below the combo boxes with the results of the query.

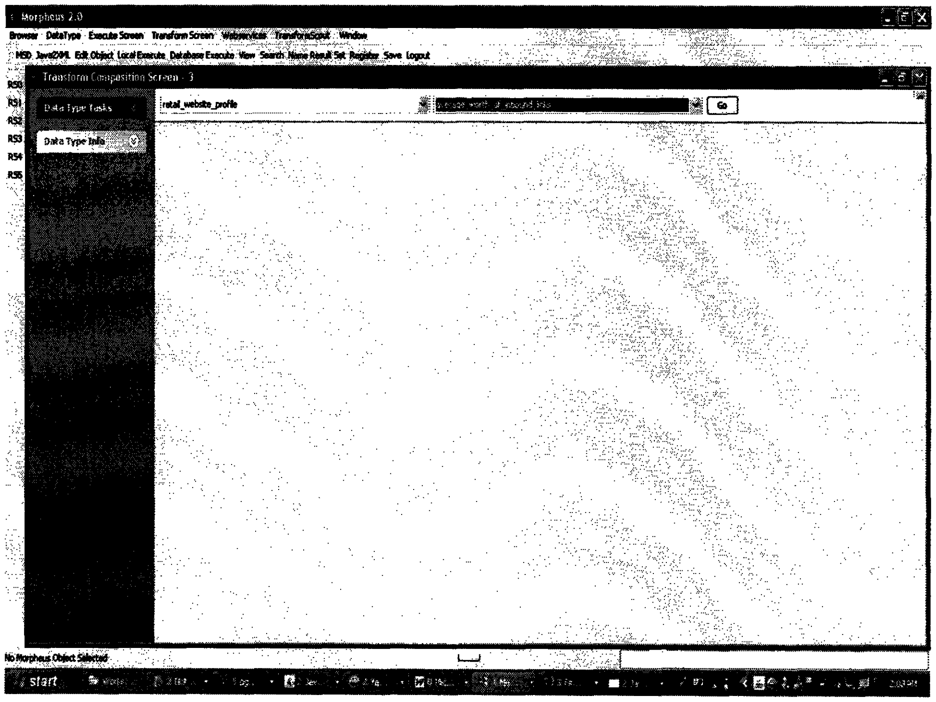


Figure 5-2: Query Interface of TransformScout

Once the user selects an input and output type, he is presented with a number of possible compositions, ranked by relevance (Figure 5.3). The highest ranked compositions appear at the top of the results screen.

After the user picks one of the results presented by TransformScout, he can import the compositions (collection of transforms and types) into the Transform Construction Tool (TCT) by clicking on the Import button next to the result.

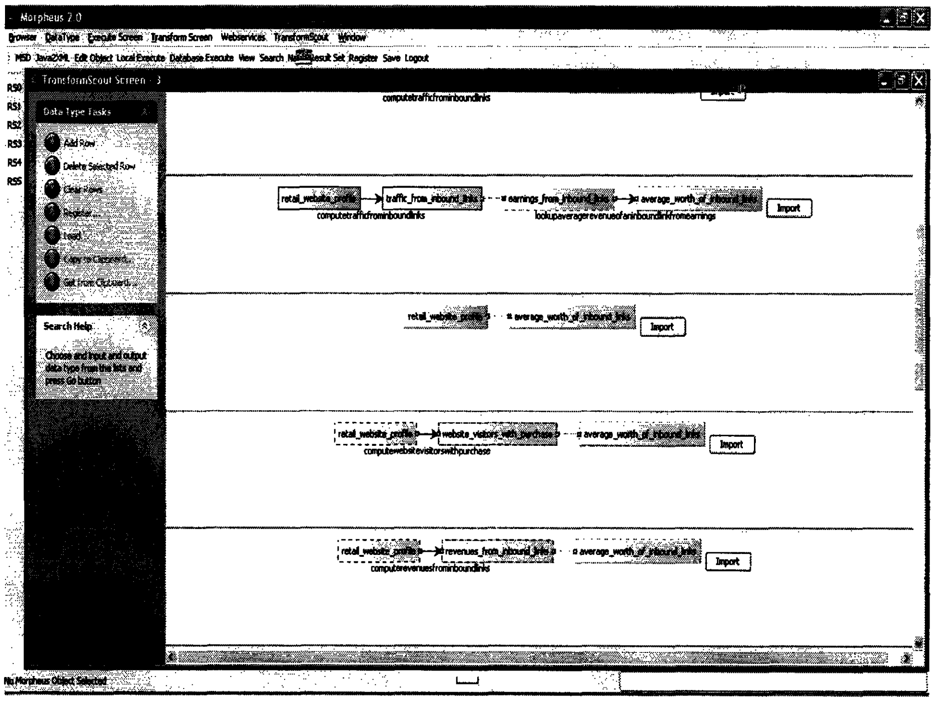
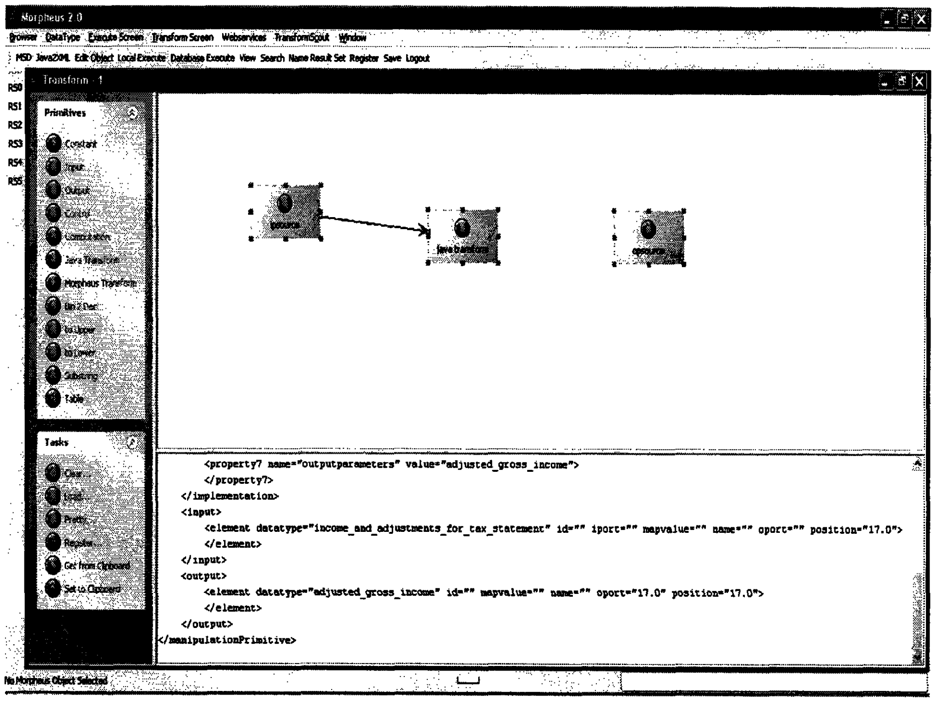


Figure 5-3: Results of a User Query in TransformScout

Figure 5.4 shows the state of the system after an import from TransformScout has been made. On the right hand side of the screen, we have a canvas with boxes and arrows. These boxes and arrows represent the compositions that were imported from TransformScout. Boxes in the TCT environment denote either the data types or the transforms in the composition. The arrows indicate the connection points between types and transforms. Below this canvas is an XML representation of a selected object within the workspace. The XML representation describes the structure and metadata of a data type or a transform. On the left hand side of the TCT screen, there are a number of primitives (building blocks) for writing programs in TCT such as "Computation", "Control" primitives. The user can now wire together the missing link between compositions with the various primitives provided by TCT.

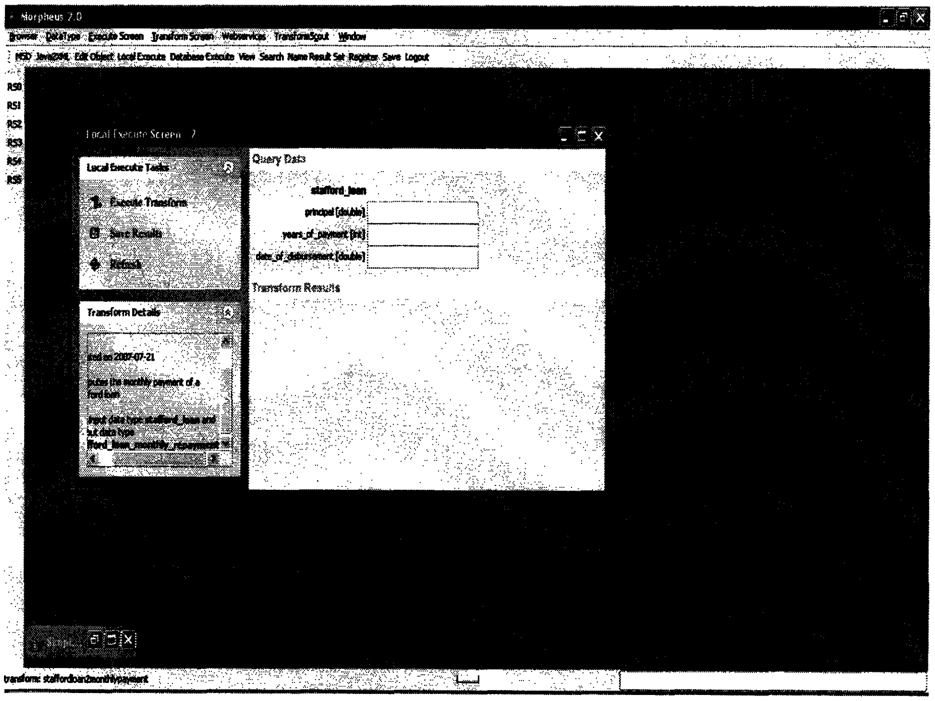
Alternatively, if there is a full composition, the user can use "Import" option to do unit testing on the composition by specifying input values and inspecting values returned by the composition. Figure 5.5 shows the state of Morpheus after the unit testing option is chosen. The user is taken to the "Local Execute" screen with the full composition he chose in TransformScout. On the right side of the screen, we



```
        <property7 name="outputparameters" value="adjusted_gross_income">
        </property7>
    </implementation>
    <input>
        <element datatype="income_and_adjustments_for_tax_statement" id="" iport="" mapvalue="" name="" oport="" position="17.0">
        </element>
    </input>
    <output>
        <element datatype="adjusted_gross_income" id="" mapvalue="" name="" oport="17.0" position="17.0">
        </element>
    </output>
</manipulationPrimitive>
```

Figure 5-4: TCT After a Composition is Imported from TransformScout

have a form window where the user can enter values to the composition he picked in TransformScout. Then, he can choose "Execute Transform" option from the "Local Execute Tasks" tab on the left corner of the screen, which will execute the transform with the values input by the user. Finally, the user can inspect the returned values and determine whether the composition works as expected.


Morpheus 2.0
Browser DataType Execute Screen Transform Screen Webservices TransformScout Window
MSD Java2XML Edit Object Local Execute Database Execute View Search Name Result Set Register Save Logout
RS0
RS1
RS2
RS3
RS4
RS5
Local Execute Screen 2
Local Execute Tasks
Execute Transform
Save Results
Refresh
Transform Details
Query Data
stafford_loan
principal [double]
years_of_payment [int]
date_of_disbursement [double]
Transform Results
transform: staffordloan2monthlypayment

ced on 2007-07-21

putes the monthly payment of a
ford loan

input data type stafford_loan and
ut data type
fford_loan_monthly_repayment

Figure 5-5: Unit Testing for a Full Composition

Chapter 6

Experimental Results

6.1 User Study Overview

We conducted a user study to evaluate Morpheus and TransformScout. The user study measures whether developers complete programming tasks involving re-use more reliably and quickly when they use Morpheus and TransformScout. To measure this, we designed a set of three programming problems that can be solved with re-use and assigned them to a group of test users. (Table 6.1 and Appendix A)

6.2 Experimental Set-Up

Six users volunteered to participate in the user study. The users were a mixture of undergraduates and graduate students. The undergraduates had taken at least one course involving object oriented programming. Graduate students are PhD candidates from the Electrical Engineering and Computer Science department. Each user attempted one problem with each of the following configurations:

1. From scratch

2. All features of Morpheus except TransformScout

3. All features of Morpheus including TransformScout

We provided each user with a hands-on training session where we explained to them how to use Morpheus and TransformScout. No other training was provided, and none of the users had ever used Morpheus before the study. We assigned the problems to be solved and the configurations for completing a task randomly to each user. We recorded the users' answers, the time spent on each problem and the components that the users picked for re-use.

Table 6.1: Programming Tasks in the User Study

Problem 1	Write a program that determines whether rate of return on an investment is higher than 9%
Problem 2	Write a program that receives terms of a Canadian mortgage as input and produces an equivalent American mortgage payment
Problem 3	Write a program that computes annual payment for a Stafford loan

For programming tasks that involved Morpheus and TransformScout, the user would search for relevant components in the system. Once he was satisfied with locating all the relevant components, he would copy them to the high-level programming environment of TCT. In contrast to writing a transformation from scratch, Morpheus would provide some of the components to the user. However, as with other re-use systems, the user would incur the extra cost of searching for these components.

6.3 Results

6.3.1 Relevancy of Top-K Results of TransformScout

We wanted to determine whether the results that were chosen by the users for re-use were also ranked highly by TransformScout. To determine this, we recorded which results were chosen by users for re-use and compared them to the top-k results returned by TransformScout.

Table 6.2 presents the results for TransformScout's ranking of the results that were chosen by the users. We presented the users with 10-16 possible results for each programming task. In each case, users had to inspect fewer than top-7 results to find the answer that they chose for re-use. In two of the cases, TransformScout returned the desired result in the top-2 results.

Table 6.2: Ranking of User Picks in TransformScout

Problem #	Rank in TS	Possible Results
1	6	10
2	2	16
3	1	12

The query for Problem 1, for which TransformScout did not find the desired answer in the first half of possible results, reveals a limitation of TransformScout. Intermediate types that are similar to either the input type or the output type are ranked highly. These types may not be useful in completing a task. Such a situation occurs when programming tasks do not necessarily follow a linear data flow with consecutively similar intermediate types. Hence, with queries producing a large number of results, the desired composition may be at the bottom of a long results list.

6.3.2 Speed

Table 6.3 shows the time spent by each user on each configuration. For programming tasks that involve Morpheus only or Morpheus with TransformScout, total programming time is the sum of the component retrieval time and the time spent programming in TCT. The study is not large enough to measure differences with statistical significance, but the averages and the distributions shown in the chart suggest that Morpheus and TransformScout often helps users solve problems faster.

We found that 4 of the users were faster using Morpheus only, and 5 of the users were faster using TransformScout and Morpheus together. Only one user took slightly longer with both of the Morpheus and TransformScout configurations. For users completing a task with Morpheus alone, the average speed-up over writing from

Table 6.3: Time Spent by Each User to Complete the Programming Tasks

User #	From Scratch	Morpheus	Transform Scout	Mean	Std. Deviation
1	13	12	7	10.667	3.215
2	18	13	13	14.667	2.888
3	20	6	7	11	7.810
4	9	10	11	10	1
5	20	21	10	17	6.083
6	15	14	11	13.333	2.082

scratch was 0.613. For users with Morpheus and TransformScout configuration, the average speed-up was 1.44. Average speed-up is calculated as

$$AvgSpeedUp = \frac{\sum t_{scratch\,i}/t_{ci}}{n}$$

where $t_{scratch\,i}$ is the time spent by the ith user to complete a task from scratch (in minutes), t_{ci} is the time spent by the ith user to complete a task with a configuration c (in minutes), and n is the number of users.

For each user, we divide the time spent from scratch by the time spent with one of the other configurations. If there was not a speed-up but a slow-down with either of the configurations, we subtract the ratio (t_c / $t_{scratch}$) instead of adding. We sum speed-up ratios for each user and divide the sum by the number of users to compute the average speed-up.

6.3.3 Reliability

In our user study setup, none of the users tested their answers to the programming tasks. 5 of the users found the correct solutions writing from scratch. 3 of the users found correct solutions with Morpheus alone. 5 of the users found correct solutions when they used Morpheus and TransformScout together. A correct solution in our assessment means that it conforms to the specifications in the programming task provided.

In both Morpheus only and Morpheus with TransformScout configurations, the users constructed the tasks in the programming environment of TCT. Therefore, at a first glance, it is not obvious why users completed tasks more reliably with TransformScout than they did in the Morpheus only configuration. We observe that all the users who finished the tasks with Morpheus incorrectly had errors in making connections between types and transforms. We guess that users had trouble connecting components correctly because of the naming conventions of types and transforms. For example, all input types in the TCT environment are called "ipsource" instead of the specific name of the data type. Similarly, all transforms are named "java transforms" instead of the actual name of the transform, e.g., "dollar2euroconverter". Our guess is that users made more mistakes with Morpheus only configuration, because they had to make a greater number of connections between types and transforms compared to the TransformScout configuration, where a portion of the components had been connected already.

In our tests, users noted a number of areas of TCT that caused confusion. In particular, the naming convention of types in TCT, the naming conventions of the programming primitives, and the obscuring of connections between fields of types. Given the feedback in the post-study interview, these problems do not appear to be flaws in the design of Morpheus, but implementation errors not intrinsic to the model upon which the functionality is based.

Chapter 7

Conclusions

Data integration is the process of combining data residing at different sources to conform to some target representation. The need for data integration arises because data sources in different organizations have been designed and developed independently to meet local requirements. The process of integration brings together data that is required for communication among organizations that have disparate representations of the shared data.

Both the research community and the industry offer techniques and systems to support data integration tasks. In this thesis, we talked about Morpheus, a novel Transformation Management System for data integration. Morpheus' key advantage over existing approaches is its emphasis on the management of re-usable and shareable transformations. Morpheus allows users to write transformations easily and to share and re-use these transformations.

The development of Morpheus has uncovered a number of issues within the area of integration of information. These issues are the basis for the following recommendations to the data management community:

Store transformations for efficiency and better governance: Our observation from industry is that many transformations are written over and over again. To eliminate the duplicate work of re-implementing the same logic, we need a way to store transformations and locate them efficiently for later re-use. Furthermore, metadata is not consistent even within a single organization. Better governance for metadata is needed to resolve semantic conflicts in an organization, which can be achieved through a centralized repository of metadata.

*Abandon the hope that we will agree on shared semantics:*Based on the limited success of standards and ontologies, we disagree with the belief that that shared ontologies are the panacea for the data integration problem. Instead, we believe that information technologies will be as diverse as they are now, if not more so, in the next century. A transformation-centric approach is more realistic instead of a wide-spread adoption of federated metadata with shared semantics, at least in the near term.

*Balance human input with automation:*Several approaches in the data integration world try to automate integration tasks aggressively. Likewise, some of the approaches heavily rely on the domain knowledge supplied by human users. The techniques at both sides of the extreme have been useful to a certain extent. Yet, they often require the lacking element -more automation or more human input- to allow the solution of any non-trivial data integration task. Therefore, we propose a balanced approach with varying levels of human labor and automation according to the domain, size and latency characteristics of a data integration task.

In the second half of this thesis, we described the transform composition problem, which involves automatic composition of transformations to complete a programming task. Our implementation of TransformScout can find collections of compositions between an input and output pair. In doing so, TS helps with difficulties in selection, where the user knows what he wants to do but does not know which component to select in a transformation repository. TS also helps with coordination barriers, where the user knows which units to use, but not how to make them work together. These barriers are common in large repositories with many components.

We evaluated Morpheus and TransformScout with a user study. While our user study was very small in sample size, the improved speed-up of completion of programming tasks using Morpheus and TransformScout demonstrates the benefits of using the transformation management technologies.

TS was able to successfully meet the design criterion of utilizing composition search as a re-use option. It also laid the groundwork for formulating the ranking and the similarity metric of components in Morpheus. While informal testing of TS for reliability and speed-up has been positive (1) scaling the composition system to large numbers of components (2) automating more complex composition tasks have not been thoroughly tested. Nonetheless, we believe TransformScout will be an important part of the Morpheus project, which has the potential of being a milestone in the evolution of data integration technology in this century.

Appendix A

Programming Tasks In The User Study

A.1 Programming Task 1: Determining Rate of Return

Rate of return (ROR) is the ratio of money gained or lost on an investment relative to the amount of money invested. In this programming task, we are interested in determining whether the rate of return on a purchase is greater than some other value (i.e., 9%).

To find rate of return for a purchase, we would need four pieces of information: (1) Date we purchased an item (2) Amount of money we paid for the item (3) Date we are evaluating the value of our purchase (a date later than the date of purchase), and (4) the market price we can get for selling the item.

Once we have this information, we need to divide difference of the selling price and the original price paid for the item by original price paid for the item. We multiply this figure with the difference between date we re-sale the item and the date we purchase the item (in years) to arrive at ROR.

After we calculate the rate of return, we need to decide whether it is greater than 9%. The program should return true if the rate of return is greater than 9% and false if rate of return is less than 9%.

Write a program that determines whether rate of return on an investment is higher than 9%.

A.2 Programming Task 2: Converting Canadian Mortgage to American Mortgage

A mortgage is a type of a loan that a person takes out to buy property.

If we are given principal, annual interest rate and number of years for an American mortgage as inputs, we can compute monthly payment of this mortgage with the following formula:

$$MonthlyPayment = \frac{P*J}{(1-(1+J)^{-N})}$$

where

- **P** = principal, the initial amount of the loan
- **I** = the annual interest rate (from 1 to 100 percent)
- **L** = length, the length (in years) of the loan
- **J** = monthly interest in decimal form = I / (12 x 100)
- **N** = number of months over which loan is amortized = L x 12

If we know principal outstanding, annual interest rate percentage and number of years for paying mortgage, we can compute the monthly payment of a Canadian mortgage with the following formula:

$$MonthlyPayment = P * \frac{200}{1-(1+\frac{i}{200})^{1/6*N}}$$

where:

P = principal

i = annual interest rate percentage

L = number of years of payment

N = number of months over which loan is amortized = L x 12

In United States, mortgages are compounded annually. In Canada, mortgages are compounded semi-annually; hence the difference in the formulae. To convert Canadian interest rate for a mortgage to an equivalent US interest rate, we can use the following formula:

US Interest Rate = 1200 * ((1 + Can. Interest Rate/200)^(1/6) - 1)

Given terms of a Canadian mortgage, to determine an equivalent monthly payment of an American mortgage, we need to do the following:

1. We first need to convert the Canadian interest rate to the equivalent American interest rate.

2. We need to plug in the interest rate we found in the previous step and rest of the terms for Canadian mortgage (payment, principal) to the formula that computes American mortgage.

Write a program that receives terms of a Canadian mortgage as input and produces an equivalent American mortgage payment.

A.3 Programming Task 3: Calculating Annual Payment for Stafford Loans

Stafford loans are loans for higher education, granted by U.S. Department of Education. To calculate annual loan payment for Stafford loans, we need to input the following information:

Stafford Loan Information

Principal	Total amount of loan borrowed	Double
Years	Number of years to pay for the loan	Int
Date of Disbursement	Date the loan was disbursed	Date

We also need to find out the interest rate for the Stafford loan. Interest rates for Stafford loans depend on the date they were disbursed (lent). Typically, the earlier the date, the higher the interest rate is. We are given a Stafford Loan Interest Rate table, which provides interest rate given a data of disbursement:

Date of Disbursement	Annual Interest Rate
7/2006-6/2012	6.80 %
7/1998-6/2006	6.62 %
7/1995-6/1998	7.42 %
7/1994-6/1995	8.02 %
10/1992-6/1994	8.02 %

Stafford Loan Interest Rates

Now, we have both principal, number of years to pay for the loan and the interest rate. Given an interest rate, loan payment per month is computed with the following formula:

monthly loan payment = (interest rate * principal) /(1-b^e)

where e = - 12 *(number of years of payment) and

b =(interest rate/12) + 1.0

Finally we can compute the annual payment for Stafford loans with the following formula:

annual loan payment = monthly loan payment * 12.

Write a program that computes annual payment for a Stafford loan.

Bibliography

[1] C. White, Data Integration: Using ETL, EAI and EII Tools to create an Integrated Enterprise, Technical Report, BI Research, The Data Warehousing Institute, 2006.

[2] S. Abiteboul, R. Agrawal, P. A. Bernstein, M. J. Carey, S. Ceri, W. B. Croft, D. J. DeWitt, M. J. Franklin, H. Garcia-Molina, D. Gawlick, J. Gray, L. M. Haas, A. Y. Halevy, J. M. Hellerstein, Y. E. Ioannidis, M. L. Kersten, M. J. Pazzani, M. Lesk, D. Maier, J. F. Naughton, H.-J. Schek, T. K. Sellis, A. Silberschatz, M. Stonebraker, R. T. Snodgrass, J. D. Ullman, G. Weikum, J. Widom, and S. B. Zdonik, "The LOWELL Database Research Self Assessment," *The Computing Research Repository (CoRR)*, vol. cs.DB/0310006, 2003.

[3] P. A. Bernstein, U. Dayal, D. J. DeWitt, D. Gawlick, J. Gray, M. Jarke, B. G. Lindsay, P. C. Lockemann, D. Maier, E. J. Neuhold, A. Reuter, L. A. Rowe, H. J. Schek, J. W. Schmidt, M. Schrefl, and M. Stonebraker, "Future Directions in DBMS Research - The Laguna Beach Participants," *SIGMOD Record (ACM Special Interest Group on Management of Data)*, vol. 18, pp. 17-26, 1989.

[4] P. A. Bernstein, M. Brodie, S. Ceri, D. DeWitt, M. Franklin, H. Garcia-Molina, J. Gray, J. Held, J. Hellerstein, H. V. Jagadish, M. Lesk, D. Maier, J. Naughton, H. Pirahesh, M. Stonebraker, and J. Ullman, "The Asilomar Report on Database Research," *SIGMOD Record*, vol. 27, pp. 74-80, 1998.

[5] A. Silberschatz, M. Stonebraker, and J. Ullman, "Database Systems: Achievements and Opportunities," *Communications of the ACM*, vol. 34, pp. 110-120, 1991.

[6] E. Rahm , P. A. Bernstein, A survey of approaches to automatic schema matching, *The VLDB Journal — The International Journal on Very Large Data Bases*, v.10 n.4, p.334-350, December 2001

[7] N. F. Noy, Semantic integration: a survey of ontology-based approaches, *ACM SIGMOD Record*, v.33 n.4, December 2004

[8] A. Y. Levy, Logic-based techniques in data integration. In *Logic-Based Artificial intelligence*, J. Minker, Ed. Kluwer, International Series In Engineering And Computer Science, vol. 597. Kluwer Academic Publishers, Norwell, MA, 575-595, 2000

[9] L. Seligman, A. Rosenthal, P. Lehner, A. Smith, Data integration: Where does the time go?, In *IEEE Data Engineering Bulletin*, 2002.

[10] M. Stonebraker, Too much middleware, *SIGMOD Rec.* 31, 1 (Mar. 2002), 97-106, 2002.

[11] M. W. Bright, A. R. Hurson, S. and Pakzad, Automated resolution of semantic heterogeneity in multidatabases. In *ACM Trans. Database Syst.* 19, 2 (Jun. 1994), 212-253, 1994.

[12] H. Garcia-Molina, Y. Papakonstantinou, D. Quass, A. Rajaraman, Y. Sagiv, J. Ullman, and J.Widom, The TSIMMIS project: Integration of heterogeneous information sources. *Journal of Intelligent Inf. Systems* 8(2), 1997.

[13] M. J. Carey, L. M. Haas, P. M. Schwarz, M. Arya, W. F. Cody, R. Fagin, M. Flickner, A. Luniewski, W. Niblack, D. Petkovic, J. Thomas, J. H. Williams, and E. L. Wimmers, Towards heterogeneous multimedia information systems: The Garlic approach, In *Proc. of the 5th Int. Workshop on Research Issues in Data Engineering – Distributed Object Management (RIDE-DOM'95)*, pages 124–131, IEEE Computer Society Press, 1995.

[14] C. H. Goh, S. Bressan, S. E. Madnick, and M. D. Siegel.Context interchange: New features and formalisms for the intelligent integration of information. *ACM Trans.on Information Systems*, 17(3):270–293, 1999.

[15] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava. The Information Manifold. In *Proceedings of the AAAI 1995 Spring Symp. on Information Gathering from Heterogeneous, Distributed Enviroments*, pages 85–91, 1995.

[16] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld, An adaptive query execution system for data integration. In *Proc. of SIGMOD*, 1999.

[17] M. Lenzerini, Data integration: a theoretical perspective, In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (Madison, Wisconsin, June 03 - 05, 2002). PODS '02, 2002.

[18] P. Mitra , G. Wiederhold, J. Jannink, Semi automatic integration of knowledge sources. In: *Proc of Fusion '99*, Sunnyvale, USA, 1999.

[19] S. Bergamaschi, S. Castano, M. Vincini, Semantic integration of semi-structured and structured data sources. *ACM SIGMOD* Record 28(1):54–59, 1999.

[20] L. Palopoli, D. Sacca, D. Ursino, Semi-automatic, semantic discovery of properties from database schemas. In *Proc Int. Database Engineering and Applications Symp. (IDEAS)*, IEEE Comput, pp. 244–253, 1998.

[21] W. Li, C. Clifton, S. Liu, Database integration using neural network: implementation and experiences. In *Knowl Inf Syst* 2(1):73–96, 2000.

[22] AH. Doan, P. Domingos, A. Levy, Learning source descriptions for data integration. In *ProcWebDBWorkshop*, pp. 81–92, 2000.

[23] J. Madhavan , P. A. Bernstein, E. Rahm, Generic schema matching with Cupid. In *Proc 27th Int Conf On Very Large Data Bases*, pp. 49–58, 2001.

[24] DW. Embley, D. Jackman, L. Xu, Multifaceted exploitation of metadata for attribute match discovery in information integration. In *Proc Int Workshop on Information Integration on the Web*, pp. 110–117, 2001.

[25] J. Davies, D. Fensel, and F. v. Harmelen, *Towards the Semantic Web – Ontology-Driven Knowledge Management*: Wiley, 2002.

[26] I. Niles and A. Pease, Towards a standard upper ontology, In *The 2nd International Conference on Formal Ontology in Information Systems (FOIS-2001)*, Ogunquit, Maine, 2001.

[27] A. Gangemi, N. Guarino, C. Masolo, and A. Oltramari. Sweetening wordnet with DOLCE. *AI Magazine*, 24(3):13–24, 2003.

[28] Y. Kalfoglou and M. Schorlemmer. IF-Map: an ontology mapping method based on information flow theory. *Journal on Data Semantics*, 1(1):98–127, Oct. 2003.

[29] N. F. Noy and M. A. Musen. The PROMPT suite: Interactive tools for ontology merging and mapping. *International Journal of Human-Computer Studies*, 59(6):983–

1024, 2003.

[30] D. S. Linthicum, Enterpise Application Integration, Addison-Wesley, 2000.

[31]Informatica, Glossary of Data Integration Terms, http://www.informatica.com-/solutions/resource$_c$*enter/glossary/default.htm*

[32] V. Peralta, R. Ruggia, Z. Kedad, M. Bouzeghoub , A Framework for Data Quality Evaluation in a Data Integration System, In *19º Simposio Brasileiro de Banco de Dados (SBBD'2004)*, Brasil, 2004.

[33] C. Clifton, M. Kantarcioglu, A. Doan, G. Schadow, J. Vaidya, A. Elmagarmid, and D. Suciu, Privacy-preserving data integration and sharing. In *Proceedings of the 9th ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery* (Paris, France), DMKD '04, 2004.

[34] E. Rahm, H. Do. Data Cleaning: Problems and Current Approaches, In *Bulletin of the Technical Committee on Data Engineering*, 23, 4, 2000.

[35] L. V. S. Lakshmanan, F. Sadri, and I. N. Subramanian, SchemaSQL - A Language for Interoperability in Relational Multi-database Systems, presented at Twenty-Second International Conference on Very Large Databases, Mumbai, India, 1996.

[36] A. Sheth and J. A. Larson, "Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases," *ACM Computing Surveys*, vol. 22, pp. 183-236, 1990.

[37] D. L. Goodhue, M. D. Wybo; L. J. Kirsch, The Impact of Data Integration on the Costs and Benefits of Information Systems, In *MIS Quarterly*, Vol. 16, No. 3. (Sep., 1992), pp. 293-311, 1992.

[38] Open Financial Exchange (OFX), http://www.ofx.net/.

[39] Interactive Financial Exchange (IFX), http://www.ifxforum.org/.

[40] XBRL (Extensible Business Reporting Language), http://www.xbrl.org/Home/.

[41] Financial Information eXchange (FIX) Protocol, http://www.fixprotocol.org/.

[42] FpML® (Financial products Markup Language), http://www.fpml.org/.

[43] T. Milo, and S. Zohar, Using schema matching to simplify heterogeneous data translation. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1998.

[44] R. Dhamankar, Y. Lee, A. Doan, A. Halevy, and P. Domingos, iMAP: Discovering complex matches between database schemas. In *Proc. of the ACM SIGMOD Conf.*, 2004.

[45] T. Gannon, S. Madnick, A. Moulton, M. Siegel, M. Sabbouh, and H. Zhu, Semantic Information Integration in the Large: Adaptability, Extensibility, and Scalability of the Context Mediation Approach, MIT, Cambridge, MA, Research Report CISL# 2005-04, 2005.

[46] T. Dohzen, M. Pamuk, S. Seong, J. Hammer, and M. Stonebraker, Data integration through transform reuse in the Morpheus project, In *Proceedings of the 2006 ACM SIGMOD international Conference on Management of Data* (Chicago, IL, USA, June 27 - 29, 2006), 2006.

[47] jMock - A Lightweight Mock Object Library for Java, http://www.jmock.org/.

[48] H. Mili, F. Mili, and A. Mili. Reusing software: Issues and research directions. *IEEE Transactions on Software Engineering*, 21(6):528–562, 1995.

[49] The Prospector Project, http://snobol.cs.berkeley.edu/prospector/index.jsp/.

[48] D. Lucredio, A. F. do Prado, E. Santana de Almeida, A Survey on Software Components Search and Retrieval, *euromicro*, pp. 152-159, 30th EUROMICRO Conference (EUROMICRO'04), 2004.

[49] A. M. Zaremski, J. M. Wing, Signature matching: a key to reuse. *SIGSOFT Softw. Eng. Notes* 18, 5 (Dec. 1993), 182-190, 1993.

[50] Y. Ye and G. Fischer, Supporting reuse by delivering task-relevant and personalized information, In *Proceedings of the 24th International Conference on Software Engineering (ICSE-02)*, pages 513–523, 2002.

[51] M. Rittri. Retrieving library identifiers via equational matching of types, Jan. 24 1997.

[52] A. M. Zaremski, and J. M. Wing, Signature matching: a key to reuse. In *SIGSOFT Softw. Eng. Notes* 18, 5 (Dec. 1993), 1993.

[53] A. M. Zaremski, and J. M. Wing, Specification matching of software components. In *ACM Trans. Softw. Eng. Methodol.* 6, 4 (Oct. 1997), 333-369, 1997.

[54] J. V. Guttag, J. J. Horning, Larch: languages and tools for formal specification, Springer, 2003.

[55] D. E. Perry, S. S. Popovich, Inquire: Predicate-Based Use and Reuse, In *Proceedings of the 8th Knowledge-Based Software Engineering Conference*, 1993.

[56] E. J. Rollins and J. M. Wing. Specifications as search keys for software libraries, In *Conference on Functional Programming Languages* and *Computer Architectures,* September 1989, pp 174-183, 1989.

[57] D. Mandelin, L. Xu, R. Bodk, and D. Kimelman, Jungloid mining: helping to navigate the API jungle. In P*roceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation* (Chicago, IL, USA, June 12 - 15, 2005). PLDI '05, 2005.

[58] F. Curbera, Y. Goland, J. Klein, F. Leyman, D. Roller, S. Thatte, and S. Weerawarana, Business Process Execution Language for Web Services (BPEL4WS) 1.0, http://www.ibm. com/developerworks/library/ws-bpel.

[59] OWL-S, http://www.daml.org/services/owl-s/1.0/.

[60] N. Milanovic, M. Malek, Current solutions for Web service composition, *Internet Computing, IEEE* , vol.8, no.6, pp. 51-59, Nov.-Dec, 2004.

[61] G. H. Mealy, A Method to Synthesizing Sequential Circuits. Bell System Technical J, 1045-1079, 1955.

[62] T. Bultan, X. Fu, R. Hull, J. Su, Conversation specification: a new approach to design and analysis of e-service composition. In *Proceedings of the 12th international Conference on World Wide Web* (Budapest, Hungary, May 20 - 24, 2003). WWW '03, 2003.

[63] D. Berardi, D. Calvanese, G. De Giacomo, R. Hull, and M. Mecella, Automatic composition of transition-based semantic web services with messaging. In *Proceedings of the 31st international Conference on Very Large Data Bases* (Trondheim, Norway, August 30 - September 02, 2005). Very Large Data Bases. VLDB Endowment, 613-624, 2005

[64] X. Dong, A. Halevy, J. Madhavan, E. Nemes, and J. Zhang. Similarity Search for Web Services. In *VLDB*, 2004.

[65] P. Resnik,Using information content to evaluate semantic similarity, In *Proceedings of the 14th International Joint Conference on Artificial Intelligence*, pages 448–453, Montreal, 1995.

[66] WordNet, http://wordnet.princeton.edu/.

[67] J. J. Jiang and D. W. Conrath, Semantic similarity based on corpus statistics and lexical taxonomy. In *Proceedings of International Conference on Research in Computational Linguistics*, Taiwan, 1997.

[68] D. Lin, An information-theoretic definition of similarity. In *Proceedings of the 15th International Conference on Machine Learning*,Madison, WI, 1998.

[69] A. Budanitsky, G. Hirst, Semantic distance in WordNet: An experimental, application-oriented evaluation of five measures, 2001.

[70] J. L. Chen, Y. Liu, L. T. Sam, J. Li, Y. A. Lussier , Evaluation of high-throughput functional categorization of human disease genes, In *BMC Bioinformatics*, 2007 May 9;8 Suppl 3:S7, 2007.