SPIDER: a Schema mapPIng DEbuggeR[*]

Bogdan Alexe
UC Santa Cruz
abogdan@cs.ucsc.edu

Laura Chiticariu
UC Santa Cruz
laura@cs.ucsc.edu

Wang-Chiew Tan
UC Santa Cruz
wctan@cs.ucsc.edu

ABSTRACT

A schema mapping is a high-level declarative specification of how data structured under one schema, called the source schema, is to be transformed into data structured under a possibly different schema, called the target schema. We demonstrate SPIDER, a prototype tool for debugging schema mappings, where the language for specifying schema mappings is based on a widely adopted formalism. We have built SPIDER on top of a data exchange system, Clio, from IBM Almaden Research Center. At the heart of SPIDER is a data-driven facility for understanding a schema mapping through the display of *routes*. A route essentially describes the relationship between source and target data with the schema mapping. In this demonstration, we showcase our *route engine*, where we can display one or all routes starting from either source or target data, as well as the intermediary data and schema elements involved. In addition, we demonstrate "standard" debugging features for schema mappings that we have also built, such as computing and exploring routes step-by-step, stopping or pausing the computation with breakpoints, performing "guided" computation of routes by taking human input into account, as well as tracking the state of the target instance during the process of computing routes.

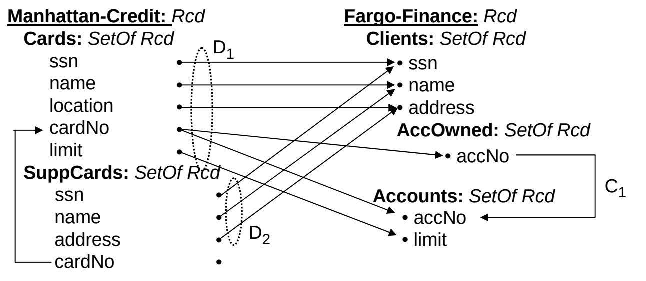
1. INTRODUCTION

The *data exchange problem* is to transform data structured according to a *source* schema into data that conforms to a *target* schema [3, 4]. Usually, the target schema is created independently of the source schema and may have constraints of its own. The behavior of a data exchange system is largely governed by the specification of *schema mappings*. A schema mapping is a high-level declarative specification of how data structured under a source schema $\mathbf{S}$ is to be transformed into data structured under a target schema $\mathbf{T}$. In research on relational-to-relational data exchange as well as data integration [4, 5], the widely adopted language for specifying schema mappings is based on the formalism of *tuple generating dependencies (tgds)* and *equality generating dependencies (egds)*. Schema mappings are desirable because they are relatively easier to manipulate and understand in contrast with the complex queries such as SQL, XSLT or XQuery, that are often used to implement and execute the schema mappings in order to perform the exchange. In fact, the data exchange system Clio [3] uses a language based on tgds and egds for specifying schema mappings and it compiles a schema mapping into an internal representation from which executables such as SQL, XSLT or XQuery queries are generated in order to perform the exchange. Clio takes as input a source instance and applies the generated query to obtain a target instance that satisfies the schema mapping. Valuable programming effort needed to implement the desired exchange could be saved if the schema mapping is accurately specified to reflect the user's intention. We view the language of schema mappings as a (higher-level) programming construct for specifying the exchange and it is for the same motivation as building a debugger for a programming language that we build a debugger for schema mappings. Our tool can be seen as an effort towards developmental support for programming with the language of schema mappings.

The need for such a debugging tool can be seen from several other factors. First, schema mappings in data exchange systems (as well as data integration systems) are often generated by schema matching tools [7]. While the generated mappings are often close to a user's intention, it is very likely that they need to be further refined before they accurately reflect the user's intention. Second, the schema mappings, whether they are known or generated through schema matching tools, can often be large and therefore difficult to debug or understand. Finally, a debugging tool provides a facility for users to understand their specification through "trial-and-error". Therefore, a facility that would allow a user to understand the schema mappings by browsing through and probing the data at hand will be extremely useful for enhancing the user's understanding of the specification of the system.

We demonstrate SPIDER, a debugging tool for schema mappings that is currently built on top of Clio [3]. A debug button from Clio activates SPIDER, which will inherit the schema mapping, as well as the source and target instances from Clio. A primary feature of SPIDER is a facility for understanding and debugging schema mappings by visualizing *routes* for some data in the source or target. A route essentially illustrates the relationship between some source and target data with the schema mapping. A route is informative in that it shows the source data, the source and target schema elements, as well as the intermediate data in the target instance that led to the target data. It has a logical semantics that is independent of the underlying procedures used to implement the exchange. So SPIDER is able to debug a schema mapping without referring to Clio's underlying execution engine (whether it is based on SQL, XSLT or XQuery).

[*]Supported in part by NSF CAREER Award IIS-0347065 and NSF grant IIS-0430994.



Source-to-target dependencies:

D_1: foreach x_1 in Manhattan-Credit.Cards
exists x_2 in Fargo-Finance.Clients, x_3 in x_2.AccOwned,
x_4 in Fargo-Finance.Accounts
where x_3.accNo=x_4.accNo
with x_1.ssn= x_2.ssn and x_1.name= x_2.name and x_1.location= x_2.address
and x_1.cardNo= x_3.accNo and x_1.limit= x_4.limit

D_2: foreach x_1 in SuppCards
exists x_2 in Fargo-Finance.Clients
with x_1.ssn= x_2.ssn and x_1.name= x_2.name and x_1.address= x_2.address

Target dependency:

C_1: foreach x_1 in Fargo-Finance.Clients, x_2 in x_1.AccOwned
exists x_3 in Fargo-Finance.Accounts with x_2.accNo=x_3.accNo

Figure 1: An example relational-to-XML schema mapping

Summary of features We demonstrate four main functionalities that we have implemented in our system. (a) A user can select some target data and SPIDER can display routes taken by some source data and intermediate data to arrive at the selected target data. We also allow a user to select source data, instead of target data, and SPIDER will display routes for the selected source data. This feature is important for understanding how the selected target data was created in the result, or the consequences of the selected source data under the specified exchange. (b) SPIDER has the ability of displaying one route fast and alternative routes as needed. It can also display *all* routes for selected target or source data. The latter feature is important for understanding data exchanged from distinct but highly overlapping data sources, or data exchanged to multiple targets. In displaying all routes, a concise representation of all routes is presented to the user by factoring common steps in the routes. Furthermore, this representation is complete in the sense that every minimal route can essentially be found from the representation. (c) SPIDER is also equipped with “standard” debugging features such as breakpoints, step-by-step computation of routes and a “watch” window for visualizing how the target instance is created, as well as the bindings for variables used in a dependency at each step. In the process of computing routes step-by-step, the user has the option of applying a dependency that she thinks is most relevant at each step and she may also backtrack to explore alternative dependencies as desired. (d) The user may select a (source or target) schema element and request SPIDER to display the schema-level “routes” of the selected element according to the schema mapping. This feature enables one to understand the relationships between source and target schema elements, independent of the instances.

Our main technical contribution is an implementation of a debugger for schema mappings, with the above described features. To the best of our knowledge, SPIDER is the first debugging tool for schema mappings that is equipped with functionalities similar to that of a standard debugger for a programming language. Even though we use Clio as our underlying data exchange system, our implementation is independent of Clio. Consequently, we believe that the techniques we have developed can be applied to future data exchange systems, as well as data integration systems that make use of schema mappings based on similar logical formalisms. Furthermore, SPIDER works for any combination of relational and XML source and target schemas. In addition, we have shown in [1] that the route algorithms of SPIDER have the following properties. (1) Whether computing all routes or one route, the algorithms execute in polynomial time in the size of the input. (2) Our algorithm for computing one route is complete: It will find a route if there is one. (3) Our algorithm for computing all routes returns a structure that concisely embeds all routes, where *all* means every minimal route can essentially be found in this structure.

Commercial systems such as Altova’s Mapforce and Stylus Studio ship with integrated facilities for data exchange, but debug only at the “lower-level”; their debuggers are for the XSLT or XQuery language that is used to specify the exchange. We refer the interested reader to [1] for a discussion on related work.

2. DEMONSTRATION OVERVIEW

In this section we use a simple schema mapping to illustrate the main features of SPIDER, as well as describe some of the techniques implemented in the system. We emphasize that real life schema mappings are, in general, much more complex (thus harder to debug without the use of some tool) than our illustrative example, which we keep simple, for ease of exposition. In our actual demonstration, however, we will use a more complex schema mapping, to better justify the need for a debugging tool for schema mappings.

2.1 Our schema mapping example

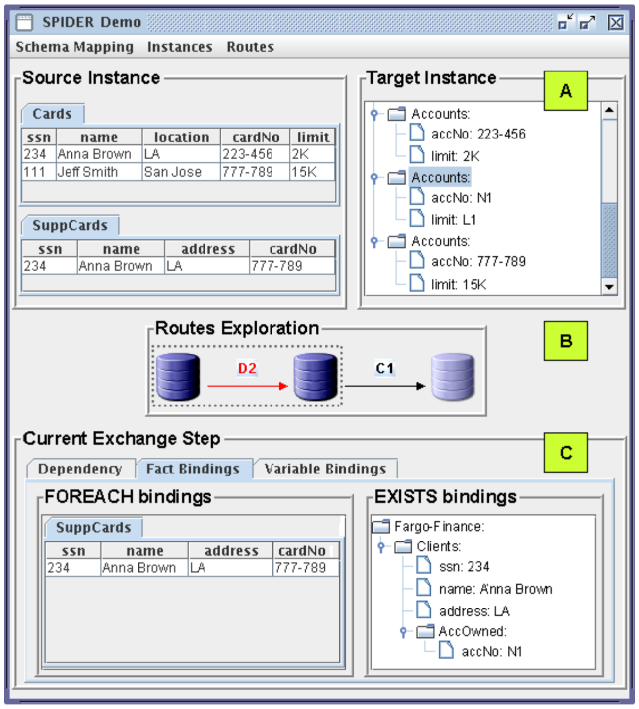
We use the nested relational model and mapping language of [6] for describing our schema mapping (see Figure 1). The top of Figure 1 shows a relational source schema Manhattan-Credit and a nested target schema Fargo-Finance. The bottom of Figure 1 shows *source-to-target dependencies (s-t dependencies)* D_1 and D_2 and the *target dependency* C_1. (The dependencies are also illustrated as arrows). Clio generates the arrows in Figure 1 semi-automatically (based on user input), interprets the arrows into dependencies and compiles the schema mapping into executable SQL, XSLT or XQuery queries to execute the exchange.

In this example, the objective is to migrate every card holder and supplementary card holder of Manhattan Credit as a client of Fargo Finance. The specification of the exchange is described by the dependencies in Figure 1. According to D_1, for each *Cards* record of Manhattan Credit (refer to foreach clause of D_1), there must exist *Clients* and *Accounts* records in Fargo Finance (exists clause of D_1) with corresponding social security number, name, address, account number and limit values (with clause of D_1). Similarly, the s-t dependency D_2 specifies that for each *SuppCards* record in the source, there must be a *Clients* record in the target with the same social security number, name and address. In addition to D_1 and D_2, the schema mapping includes a target dependency C_1 that the exchanged target instance must satisfy. This constraint specifies that for every *Clients* record, there must be an *Accounts* record with the same account number. We clarify that Clio’s implementation currently does not support egds (e.g., key constraints) in the target, although the general framework [2, 8] and our debugger allow for such specification.

2.2 Some debugging features of SPIDER

In the following, we illustrate some features of SPIDER that Alice, a banking specialist, may use to identify problems with the schema mapping in Figure 1.

We assume Alice uses Clio to generate a target instance that satisfies the schema mapping in Figure 1 from a given source in-



Source Instance

Cards

ssn	name	location	cardNo	limit
234	Anna Brown	LA	223-456	2K
111	Jeff Smith	San Jose	777-789	15K

SuppCards

ssn	name	address	cardNo
234	Anna Brown	LA	777-789

Target Instance

```
Accounts:
    accNo: 223-456
    limit: 2K
Accounts:
    accNo: N1
    limit: L1
Accounts:
    accNo: 777-789
    limit: 15K
```

Current Exchange Step

FOREACH bindings

SuppCards

ssn	name	address	cardNo
234	Anna Brown	LA	777-789

EXISTS bindings

```
Fargo-Finance:
    Clients:
        ssn: 234
        name: Anna Brown
        address: LA
        AccOwned:
            accNo: N1
```

Figure 2: Screenshot from SPIDER. (A) Source and target instances; (B) high-level view of a route; (C) detailed view of the exchange step with D_2.

stance. The debugging process is initiated from Clio through a "debug" button which triggers the display of SPIDER's visual interface (Figure 2). SPIDER automatically imports the source and target instances from Clio. Alice can also use an alternative source instance for debugging purposes and a corresponding target instance is automatically computed using Clio's engine and loaded in SPIDER.

Suppose Alice specifies a source instance consisting of the three tuples (or facts) shown below.

t_1:*Cards*(234, Anna Brown, LA, 223-456, 2K)
t_2:*Cards*(111, Jeff Smith, San Jose, 777-789, 15K)
t_3:*SuppCards*(234, Anna Brown, LA, 777-789)

A target instance (generated by Clio) that satisfies the schema mapping in Figure 1 with the source instance is shown below (facts t_4 to t_8).

t_4:*Clients*(234, Anna Brown, LA, {*AccOwned*(223-456),
AccOwned(N_1)})
t_5:*Clients*(111, Jeff Smith, San Jose, {*AccOwned*(777-789)})
t_6:*Accounts*(223-456, 2K)
t_7:*Accounts*(N_1, L_1)
t_8:*Accounts*(777-789, 15K)

The *AccOwned* facts are nested inside the *Clients* facts t_4 and t_5. The values N_1 and L_1 are "unknown" or *null values* created during the exchange. The schema mapping allows for incomplete specification in general, hence it might not contain information concerning the creation of certain required elements in the target. As an example, the accNo value of *AccOwned* facts is left unspecified by the s-t dependency D_2. As a consequence, such values are automatically generated [6].

2.2.1 *Computing routes*

SPIDER can compute (multiple) routes for source or target facts and we explain how Alice makes use of such routes in order to understand and debug the schema mapping in Figure 1. While browsing through the target instance, Alice discovers something unusual about the fact t_7: it has unknown account number and limit values. In order to understand how t_7 was obtained, she probes the element of *Accounts* (highlighted in Figure 2.A) with the two unspecified values. SPIDER returns a route which, essentially, illustrates the relationship between some source data and some target data which includes the fact that was probed.

A route can be visualized at two levels of detail. In the *high-level* view, a route is schematically displayed as a sequence of steps. Figure 2.B shows the high-level view of the route computed for the fact t_7, which contains two steps with D_2 and respectively, C_1. From this view, Alice may probe any step in the route and SPIDER shows a *detailed view* containing specific information about that step. Figure 2.C illustrates the detailed view of the first exchange step with D_2 which shows the *fact bindings* for the foreach and exists clauses of D_2. Intuitively, the source fact t_3 is a witness for the target fact t'_4: *Clients*(234, Anna Brown, LA, {*AccOwned*(N_1)}). (An alternative view, not shown in Figure 2.C, illustrates the *bindings for the variables* in the foreach and exists clauses of D_2, similar to what is shown in Figure 3.B). The detailed view for the second step in the route (not shown in Figure 2) illustrates that t'_4 is a witness for the probed target fact t_7 with the target dependency C_1. By analyzing the steps of the route, Alice realizes that the dependency D_2 misses the association with the source *Cards* relation (specified by the foreign key on the cardNo attribute), and can modify D_2 accordingly.

The algorithm for computing one route for a set of target facts is described in detail in [1]. Intuitively, each selected target fact t is first matched against the exists clause of the s-t and target dependencies. If there exists a s-t dependency that can witness t with some source facts, the corresponding step is appended to the current route which is initially empty. Otherwise, if t can be witnessed with some target dependency and some other target facts T, the computation proceeds recursively to find a route for T, which (if found) is appended to the current route. The process repeats until there are no "unwitnessed" selected target facts and in case some selected target fact cannot be witnessed with any dependency in the schema mapping, the algorithm stops with output "no route". Our algorithm is sound and complete and we refer the reader to [1] for more details.

Similar to computing routes for target facts, SPIDER can also compute routes for selected source facts. This feature is useful for identifying consequences of the selected source facts in the target. The algorithm for finding one route for a set of source facts is similar in spirit to our algorithm for finding one route for a set of target facts, except that now we first match every selected source fact t against the foreach clause of a s-t tdependency that together with t (and possibly other source facts) can witness some target facts T. We then may repeat the process by trying to find a target dependency that together with some of the facts in T witness some other target facts, and so on, until no more target facts can be witnessed.

In general, observe that there may be multiple routes for selected source or target data. For example, there are two routes for the target fact t: *Clients*(234, Anna Brown, LA). One route shows that t can be witnessed with the source fact t_1 and the s-t dependency D_1, while the second route witnesses t with t_3 and D_2. As a consequence, SPIDER is not limited to the computation of a single route and *alternative* routes (or *all* routes) can be efficiently computed on demand for a set of source or target facts. We have modified our algorithm for computing one route to compute an alternative route by allowing it to continue searching for another route, as opposed to abandoning the computation as soon as one route is computed.

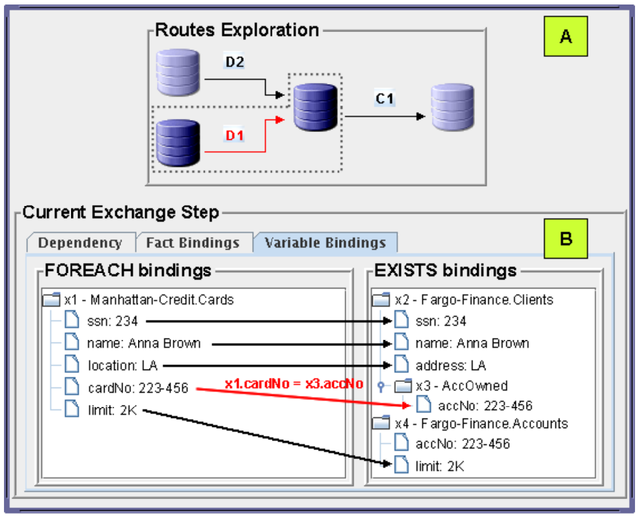


Figure 3: (A) Guided exploration of an alternative route; (B) "watch" window for the variables of D_1.

If an alternative route is sought for some selected fact t, we simply "undo" the last choice of dependency and facts made by the algorithm when computing one route for t and proceed with the next choice, if any. Similarly, all routes can be obtained by exhaustively exploring all possible choices of dependencies and facts in each step, instead of abandoning the computation when the first route is output. When the user requests the computation of multiple routes, we display these routes in a tree-like manner, with the common parts factored out. Our algorithm for computing all routes is complete in the sense that it produces a structure which, essentially, embeds every *minimal* route [1].

2.2.2 Guided routes exploration

At any step in a route, the system is able to suggest several dependencies that can be applied, if any. The user may choose one of these candidates and decide upon the direction of the route, thus "guiding" the exploration of the route as desired. At a later moment, the user can reconsider her decision and explore some alternative segment of the route.

Suppose that in Figure 2.C Alice selects the fact *Clients*(234, Anna Brown, LA) denoted as t. As illustrated in Figure 3.A, SPIDER shows that there is an alternative route for this fact with the dependency D_1, in addition to D_2. Figure 3.B presents the situation where Alice examines the effects of applying the dependency D_1. (This is an alternative view to the one presented in Figure 2.C.) In this view, Alice examines the *bindings for variables* x_1,...,x_4 in the foreach and exists clauses of D_1, instead of fact bindings. (Note that the exchange step with D_1 witnesses the part of t'_4 denoted as t.) The arrows indicate the way values in the source are used to generate values in the target, according to D_1.

2.2.3 "Standard" debugging features

Similar to debuggers for programming languages, SPIDER is equipped with standard features such as step-by-step route computation, breakpoints and "watch" windows. Routes can be computed in one run, or step-by-step. In the latter mode, the user is allowed to set breakpoints on specific dependencies that when reached, trigger a stop in the computation of the route. The user can examine the subsequent steps one at a time, or she can decide to demand the continuation of the computation until reaching another breakpoint or the end of the route. At each step, the user is also allowed to choose a specific tgd to be applied, thus guiding the exploration of the route as explained above. When examining a step in a route, the user can "watch" the bindings for the facts or variables in the dependency as described earlier (Figures 2.C and 3.B). If desired, SPIDER can also automatically highlight selected bindings on the source or target instances. In addition, SPIDER is equipped with a "watch" window where one can visualize the creation of the target instance while exploring routes.

2.2.4 Schema-level exploration of schema mappings

In addition to its data-driven debugging capabilities, SPIDER also facilitates the understanding of schema mappings directly at the level of the source and target schemas through the display of *schema-level* routes. Given a schema element T_e, SPIDER can schematically show the source or target schema elements that are directly or indirectly responsible for creating values for T_e, as well as other target schema elements whose values are copied from T_e. This feature allows a user to focus on parts of the schema mapping that are relevant for selected source or target schema elements.

3. SYSTEM ARCHITECTURE

SPIDER is currently implemented in Java 1.5, on top of Clio. The architecture of the system consists of four modules. The *schema mapping loader* and the *data loader* modules are responsible for loading schema mappings and respectively, relational and XML instances. Functionalities for computing and exploring routes, as well as support for the standard debugging features are implemented in the *routes engine*. Finally, the *schema-level tracer* module permits the exploration of schema-level routes for source and target schema elements.

4. REFERENCES

[1] L. Chiticariu and W. Tan. Debugging schema mappings with routes. In *Proceedings of the International Conference on Very Large Data Bases (VLDB) (To appear)*, 2006.

[2] R. Fagin, P. G. Kolaitis, R. J. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *Theoretical Computer Science*, 336(1):89–124, 2005.

[3] L. M. Haas, M. A. Hernandez, H. Ho, L. Popa, and M. Roth. Clio grows up: from research prototype to industrial tool. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 805–810, 2005.

[4] P. G. Kolaitis. Schema mappings, data exchange, and metadata management. In *ACM Symposium on Principles of Database Systems (PODS)*, pages 61–75, 2005.

[5] M. Lenzerini. Data Integration: A Theoretical Perspective. In *Proceedings of the ACM Symposium on Principles of Database Systems (PODS)*, pages 233–246, 2002.

[6] L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernández, and R. Fagin. Translating Web Data. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*, pages 598–609, 2002.

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

[8] C. Yu and L. Popa. Constraint-based xml query rewriting for data integration. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 371–382, 2004.