Debugging Data Exchange with Vagabond

Boris Glavic, Jiang Du, Renée J. Miller
University of Toronto
glavic,jdu,miller@cs.toronto.edu

Gustavo Alonso
ETH Zurich
alonso@inf.ethz.ch

Laura M. Haas
IBM Almaden Research Center
laura@almaden.ibm.com

1. INTRODUCTION

In this paper, we present *Vagabond*, a system that uses a novel holistic approach to help users to understand and debug data exchange scenarios. Developing such a scenario is a complex and labor-intensive process where errors are often only revealed in the target instance produced as the result of this process. This makes it very hard to debug such scenarios, especially for non-power users. Vagabond aides a user in debugging by automatically generating possible explanations for target instance errors identified by the user.

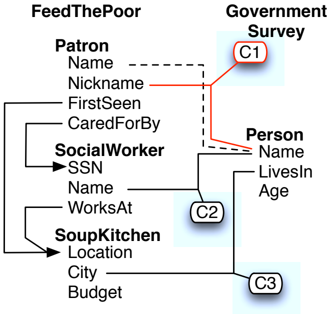
Schema mappings are declarative constraints that model the relationship between a *source* and a *target* schema. Data exchange systems, such as Clio [6], ORCHESTRA [5], and many others, use schema mappings to produce an *instance* of the target schema based on an instance of the source schema. Creating a mapping between two schemata is a semi-automatic, multi-step process. In a first step, *correspondences* between atomic elements of the source and target schema are identified. Based on these correspondences, the constraints of the schemata (e.g., foreign keys constraints), and user input, the system generates the *schema mappings* that are eventually used to create executable *transformations* specified in, e.g., SQL or XSLT.

For large schemata, this multi-step process is error-prone. As mentioned above, often, errors become apparent only in the generated target instance. For example, a user may recognize that some attribute values in the target instance are incorrect. Tracing errors is time-consuming and complex, because of the many possible sources of errors: data, correspondences, schema mappings, or transformations. Previous work focused on aiding the user in debugging by (1) providing additional information, such as provenance, and better query language support for schema mappings (TRAMP [4], MXQL [7], Spider[1]) or (2) through programming language style debugging like breakpoints (Spider [1]). With *TRAMP*, we showed how information about data, its provenance, and mapping scenario information (correspondences, schema mappings, transformations) can be efficiently used to debug a wide range of typical data exchange errors. TRAMP and the other approaches mentioned above have in common that they are more tailored for power users - they require the user to understand what possible sources of errors are and rely on her to guide the debugging process accordingly. In contrast to these approaches, Vagabond automatically generates and ranks *explanations* for errors in a data exchange setting based on user provided input about which parts of a generated target instance are erroneous. The rationale behind this approach is that (1) even inexperienced users are able to recognize instance errors, and (2) for both inexperienced and power users it is much harder to come up with explanations than to verify if a given explanation is correct.

The explanation generation of Vagabond builds on the facilities provided by TRAMP [4] to generate and query data, various kinds of provenance, and mapping information. We consider data, correspondences, mappings, and transformations as potential causes of errors. For instance, a possible explanation for incorrect values in a target relation is that the source data where this information has been copied from is erroneous. Data provenance is used to identify this part of the source data. For each generated explanation we compute which mapping scenario elements and parts of the instance would be affected by the explanation (called the *side-effects*). The user can mark an explanation as correct. This will cause the side-effects of this explanation to be considered as additional errors, thus avoiding the need to mark all target instance errors to debug a data exchange scenario. To present more likely explanations first, we rank them on the number of side-effects they imply. The explanation generation is complemented with visualization of provenance and mapping information. Vagabond provides an easy-to-use GUI for navigating through this information.

In this demo, we show how Vagabond is used to provide explanations for common data exchange errors. The example scenarios we will use in the demonstration allow users to interact freely with the system: (1) Mark target data as incorrect and explore the explanations provided by the system, (2) navigate through data, mapping, and transformation provenance provided by TRAMP and visualized in Vagabond, (3) explore the system internals such as provenance and explanation generation. People attending the demo will learn both about Vagabond as well as about common pitfalls in schema mappings and the practical problems associated with debugging such mappings. On the research side, the demo will demonstrate how to automatically generate error explanations based on data provenance techniques for schema mappings, an area that is attracting increasing attention and exhibits interesting open problems.

Proceedings of the VLDB Endowment, Vol. 4, No. 12
Copyright 2011 VLDB Endowment 2150-8097/11/08... $ 10.00.



Mappings

$$\mathbf{M_1}: Patron(a,b,c,d) \wedge SoupKitchen(c,e,f) \Rightarrow \exists g : Person(b,e,g)$$

$$\mathbf{M_2}: SocialWorker(a,b,c) \wedge SoupKitchen(c,d,e) \Rightarrow \exists f : Person(b,d,f)$$

Transformations

```
T1 :SELECT PA.Nickname AS Name, K1.City AS LivesIn, NULL AS Age
FROM Patron PA, SoupKitchen K
WHERE PA.FirstSeen = K.Location
UNION
SELECT S.Name, K.City AS LivesIn, NULL AS Age
FROM SocialWorker S, SoupKitchen K
WHERE S.WorksAt = K.Location
```

Patron

	Name	NickName	FirstSeen	CaredForBy
pa_1	Peter Fullbright	Pete	GreenPark	777
pa_2	Harriot Welth	Welthy	Manhattan	123

SocialWorker

	SSN	Name	WorksAt
s_1	123	Jessica Good	GreenPark
s_2	666	Laurence Knopfler	GreenPark
s_3	777	Jule Hip	Manhattan

SoupKitchen

	Location	City	Budget
k_1	GreenPark	Toronto	10.000
k_2	Manhattan	New York	5.000

Person

	Name	LivesIn	Age
p_1	Jessica Good	Toronto	NULL
p_2	Laurence Knopfler	Toronto	NULL
p_3	Jule Hip	New York	NULL
p_4	Pete	Toronto	NULL
p_5	Welthy	New York	NULL

Figure 1: Example Data Exchange Scenario.

2. EXAMPLE SCENARIO

One of our demo scenarios models the mapping of data about homeless people provided by several help organizations into a global schema. This schema is used by the government to assess the situation of people without a permanent residence. Fig. 1 shows part of the schema for the "feed the poor" (FP) organization and an excerpt of the government schema. The FP schema models information about *patrons* of soup kitchens (name, nickname, the soup kitchen where they were first sighted, and the social worker that they are assigned to), *social workers* (SSN, name, and the soup kitchen they work for), and the *soup kitchens* of the organization (location, city, and yearly budget). The government schema models *persons* with name, the city they live in, and age. The arrows between the source schema elements represent foreign key constraints.

Schema Mappings: Assume the user has defined the correspondences shown in Fig. 1: The nickname of a patron (C1) and the name of a social worker (C2) both correspond to the name of a person. The city of a soup kitchen corresponds to the LivesIn attribute of a person (C3). Mappings M_1 and M_2 exemplify an initial set of mappings that might be suggested by a mapping system. Mapping M_1 relates the nickname of a patron and the city of the soup kitchen where the patron was first seen to the Name and LivesIn attributes in the target schema. Mapping M_2 relates the names of social workers and the city of their soup kitchen to the target person relation. A typical SQL transformation (T1) that generates the person relation based on these mappings is shown in Fig. 1. This query unions the SQL implementations for M_1 and M_2. Note that the Age attribute is not related to any source element by the mappings. For simplicity, we let T1 set this attribute to `NULL`. (Many data exchange systems would use skolem functions to generate values for this attribute [2]). Executing T1 over the source instance generates the target instance (Person relation) as shown in Fig. 1. For convenience, we show tuple identifiers for all tuples in the source and target instance (e.g., p_1).

Errors in the Scenario: The target instance generated by T1 contains several errors (highlighted attribute values). It is likely that the user will recognize these errors, but understanding their causes is much more involved, even for a "toy" example like this one. Assume the name of social worker "Jule Hip" was recorded incorrectly and should read "Jule Tip". This means, the incorrect name for tuple p_3 in the target (Error E1) is caused by erroneous source data (the highlighted Name attribute value of tuple s_3) and copying of this data to the target. We call this type of error a *source copy error*. In the target "Laurence Knopfler" is recorded to live in "Toronto". For sake of the example, assume that the correct value is "New York" (Error E2). This error is caused by an incorrect foreign key attribute value (WorksAt) which in turn causes s_2 to join with a wrong tuple from the SoupKitchen relation. In our terminology, this is a *source join value error*. The target schema is used to store names. Therefore, correspondence C1 should relate the names instead of the nicknames of patrons to person names (Error E3, a *correspondence error*). Patrons should be considered to live in the city where they are provided with food and not the city where they have first been sighted (Error E4). Thus, mapping M_1 uses an incorrect join path to relate source relations (*source skeleton error*).

3. GENERATING EXPLANATIONS

We now present how Vagabond produces and rankes explanations for errors in a target instance. The input to the explanation generation framework is a set E (called error set) of target attribute values that are suspected to be incorrect. We use a triple (R, t, A), called an error marker, to denote the value of attribute A for tuple t from relation R. In the graphical interface the user can browse the instance data and add an error marker by clicking on a checkbox for the corresponding attribute value. The explanation generation for the current set E is triggered by the user. Under the hood the system (1) systematically explores the search space of possible explanations for each error marker in E and (2) returns a ranked list of all possible sets of explanations Σ that cover E. A set Σ of explanations covers an error set E, if for each error marker e from E there exists an explanation σ from Σ that explains e. Note that one explanation may explain several error markers. For instance, in the example, the explanation that correspondence $C3$ is wrong will explain all incorrect target attribute values that were generated because of this correspondence.

In the example instance these are the attribute values at $(\{(Person, X, LivesIn) \mid X \in (p_2, p_4, p_5)\})$. The covering explanation sets generated by this process are then ranked according to the number of side-effects (explained below) caused by the explanations and presented to the user.

Basic Explanations: For a single error marker we generate a set of basic explanations each considering a different source of error: the source instance data, the correspondences, the mappings, or the transformations that were used to generate the target instance. In addition to the types of errors presented in Sec. 2, we consider the following types: *Superfluous Mapping Error*: A mapping is superfluous and should be removed, *Target Skeleton Error*: A mapping should use a different join path to relate target relations. Transformations are potential sources of errors too, but for reasons of space we do not discuss this type of errors in this paper. Note that we consider explanations of all types for an error marker e, unless we can rule out certain types. E.g., if the value at e is not copied from the source then we can rule out that a source copy error caused e.

Side-effects: We refer to the part of the target instance affected by an explanation σ as the *coverage* of σ. The subset of the coverage that is not in the error set E is called the *side-effect* of the explanation. The *size* of a side-effect is the number of target attribute values it covers. It seems tempting to only consider explanations without side-effects, because in principle an explanation with side-effects invalidates correct target data. However, we must account for the fact that the user may not recognize all errors in the target on first sight or wants to retrieve explanations without having to mark all errors (especially for large instances). Hence, we cannot assume that the set E is complete and, thus, that all side-effects of an explanation are actual side-effects.

Ranking: Our solution to this problem is to not rule out explanations with side-effects, but to rank the explanations on their side-effect size to present more likely explanations first. The system allows a user to indicate the correctness of an explanation σ. This will trigger Vagabond to consider all side-effects of this explanation as additional error markers. We add σ to all generated explanation sets, update their side-effects (remove the side-effects of σ), and then adapted the ranking accordingly. Thus, the user can iteratively debug a mapping scenario without having to manually mark every target instance error.

Source Copy Error: We now discuss the generation of source copy error explanations as an example of how Vagabond works. This type of error occurs if the source instance data where the value at an error marker e has been copied from is incorrect. For instance, assume a user marked a single target attribute value $e = (Person, p_3, Name)$ (Error E1 explained in Sec. 2). This value has been copied from the Name attribute value of the tuple s_3 of relation SocialWorker ($s = (SocialWorker, s_3, Name)$). One explanation for e is that a source copy error σ occurred and the value at s is incorrect. The assumption that s is erroneous could have the implication that other parts of the target instance are incorrect. However, in the example, the coverage of σ contains only e, because the incorrect value "Jule Hip" at s has not been copied to other tuples in the target (and was not used in a join or selection condition).

Vagabond builds on the provenance and query facilities of TRAMP [4, 3] to trace where values have been copied from and to compute side-effects of explanations. TRAMP is an extended DBMS that supports retrieval and querying of different types of provenance information through an SQL language extension. We briefly illustrate the process for the generation of a source copy error for error marker e. Vagabond first determines all source tuples from where the value at e has been copied from by running the following query with TRAMP:

```
SELECT PROVENANCE ON CONTRIBUTION (COPY) Name
FROM Person
WHERE tid = p3
```

This query instructs TRAMP to compute the so-called *Copy-CS* provenance for the projection on Name of the Person tuple with identifier p_3. Copy-CS provenance traces from where in the input (which tuples) the values of tuples in the output of the query have been copied from. The result of this query is $\{(SocialWorker, s_3)\}$, the source tuple from which the value at e originated[1]. Vagabond creates a single explanation σ for this set using mapping information to determine from which attribute values of each returned input tuple the value at e is derived. To compute the side-effect of σ, we need to know which other tuples and/or attribute values are potentially affected by tuples in the explanation. Provenance and mapping information is used to identify these tuples.

4. EXAMPLE SCENARIO EXPLANATIONS

We now discuss which explanations would be produced for Error E2 from the example scenario. Assume that the user realizes that the city assigned to social worker "Laurence Knopfler" in the target is incorrect and creates the error marker $e = (Person, p_2, LivesIn)$. The attribute value at this position in the target instance is *Toronto*. Recall that the correct explanation for this error is that the value at $(SocialWorker, s_2, WorksAt)$ is erroneous. Vagabond will come up with the following explanations for this error:

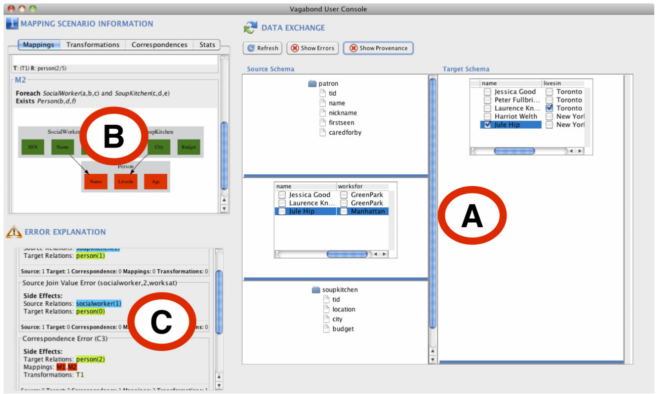
σ_1 (source copy error): This explanation assumes that the attribute values in the source where the value at e has been copied from are incorrect. In the example this is the value "Toronto" at $(SoupKitchen, k_1, City)$. The side-effect of this explanation is that other attribute values where this value is copied to are incorrect too (or tuples may disappear from the target if the value is used in a join or selection condition and the source tuple is in the provenance of these tuples). In our example, the target attribute values at $\{(Person, X, LivesIn) \mid X \in (p_1, p_4)\}$ are side-effects of σ_1.

σ_2 (source join value error): The second explanation considers an incorrect value at $(SocialWorker, s_1, WorksAt)$ to cause tuple s_1 to be joined with a wrong tuple from relation SoupKitchen. This explanation would cause all tuples generated by joining s_1 with another tuple on the WorksAt attribute to be incorrect. No such tuples and, therefore, also no side-effects exist in the example.

σ_3 (correspondence error): Another explanation is that correspondence C3 mapping attribute City from relation SoupKitchen to attribute City of relation Person is wrong. Explanation σ_3 has the side-effect that all City attribute values of Person tuples that are generated by mappings that use C3 are incorrect (all LivesIn attribute values).

σ_4 (superfluous mapping error): Explanation σ_4 states that mapping M_2 is superfluous. The removal of M_2 would

[1] We omit describing how this set would be actually represented by TRAMP (see Glavic [3]).



T: (T1) R: person(2/5)

M2

Foreach *SocialWorker*(a,b,c) and *SoupKitchen*(c,d,e)
Exists *Person*(b,d,f)

name	livesin
□ Jessica Good	□ Toronto
□ Peter Fullbri...	□ Toronto
□ Laurence Kn...	☑ Toronto
□ Harriot Welth	□ New Yorl
☑ Jule Hip	□ New Yorl

name	worksfor
□ Jessica Good	□ GreenPark
□ Laurence Kn...	□ GreenPark
□ Jule Hip	□ Manhattan

Target Relations: person(1)

Source: 1 **Target:** 1 **Correspondence:** 0 **Mappings:** 0 **Transformations:** 0

Source Join Value Error (socialworker,2,worksat)

Side Effects:
Source Relations: socialworker(1)
Target Relations: person(0)

Source: 1 **Target:** 0 **Correspondence:** 0 **M**... ...**ns:** 0

Correspondence Error (C3)

Side Effects:
Target Relations: person(2)
Mappings: M1,M2
Transformations: T1

Figure 2: Vagabond Interface.

cause the side-effect that all tuples generated by this mapping would disappear from the target instance (p_1, p_2, p_3).
$\boldsymbol{\sigma_5}$ **(source skeleton error):** This explanation assumes that mapping M_2 should use a different join path (or no join path at all) to connect tuples from relation SocialWorker with tuples from relation SoupKitchen. This explanation has the potential side-effect that all LivesIn attribute values generated by M_2 are incorrect $(\{(Person, X, LivesIn) \mid X \in (p_1, p_3)\})$ and that tuples that do not have join partners anymore would disappear from the person relation.

As explained before, Vagabond ranks explanation sets before returning them to the user. Currently, the ranking is based primarily on side-effect size and secondarily on the type of explanation (for instance, correspondence errors are considered more disruptive than instance data errors). In the future we plan to add more sophisticated ranking methods. Without presenting the details, the order for the five explanations presented above would be $\sigma_2, \sigma_1, \sigma_5, \sigma_3, \sigma_4$. Note that in this case the correct explanation for the example (σ_2) is also the one to be ranked first. However, in general this may not be the case and we cannot rule out the other explanations without more information being provided by the user. This is the main reason why convenient explanation browsing (as will be explained in the next section) plays an important role for the usability of the system.

5. USER INTERFACE

Fig. 2 shows a screenshot of Vagabond. The main interface consists of a diagram of the target and source schema (A). Selecting a relation switches between the schema and instance of this relation. In Fig. 2, the instances of relations SocialWorker and Person are shown. The left side of the screen shows a panel for correspondences, mappings, and transformations (B) and below a panel for explanations (C).
Browsing Explanations The explanation panel (C) is used to present explanations for an error set as a list to the user. The user can inspect these explanations to understand their implications and check them for correctness. Clicking on an explanation highlights its side-effects in the target instance and the mapping scenario elements affected by it and shows statistics about this error (target side-effect size, number of affected mappings, ...). For instance, for explanation σ_3 from the example, Vagabond will highlight the correspondence C3, the mappings M_1 and M_2 (because they use this correspondence), the transformation T_1 that implements these mappings, and the LivesIn attribute values of all target tuples (because these form the side-effect of this explanation). If the user realizes that an explanation is correct, she can indicate that to the system and Vagabond will adapt the explanations accordingly (as described in Sec. 3).
Statistics and Anomaly Detection To simplify the debugging process, Vagabond checks for unusual patterns in the mapping scenario and provenance information. The system reports irregularities and various statistics to the user without the need for marking errors in the target instance. One example is mappings that do not create any tuples in the target instance. Such mappings are highly likely to be incorrect. Another example is usage patterns for source relations. For each relation we identify tuples that have been accessed unusually frequent (or infrequent) by the mappings.
Provenance Browsing: In addition to the explanation generation, Vagabond acts as a graphical front-end to the provenance and query facilities of TRAMP. Thus, enabling a user to debug scenarios manually and to determine the implications of explanations. For instance, if a user selects a target instance tuple t, Vagabond highlights the source tuples from which tuple t is derived from, i.e., the data provenance of t. Selecting a mapping M highlights the target and source relations used by M and the tuples produced by M.

6. DEMO ANATOMY AND CONCLUSIONS

In the demonstration, we will showcase Vagabonds capabilities by means of an extended set of mapping scenarios each highlighting typical mapping errors. The audience will also have access to the system internals to see how, for instance, provenance queries are triggered by the explanation generation and user interaction. With Vagabond we have developed a practical system for debugging schema mappings that makes automatically generated explanations available through a convenient GUI. Error explanations enable non-power users to debug data exchange settings and simplify the debugging process for more experienced users.

7. REFERENCES

[1] L. Chiticariu and W.-C. Tan. Debugging Schema Mappings with Routes. In *VLDB*, pages 79–90, 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] B. Glavic. *Perm: Efficient Provenance Support for Relational Databases*. PhD thesis, University of Zurich, 2010.

[4] B. Glavic, G. Alonso, R. J. Miller, and L. M. Haas. TRAMP: Understanding the Behavior of Schema Mappings through Provenance. *PVLDB*, 3(1):1314–1325, 2010.

[5] Z. G. Ives, T. J. Green, G. Karvounarakis, N. E. Taylor, V. Tannen, P. P. Talukdar, M. Jacob, and F. Pereira. The ORCHESTRA Collaborative Data Sharing System. *SIGMOD Record*, 37(2):26–32, 2008.

[6] R. J. Miller, L. M. Haas, M. A. Hernández, R. Fagin, L. Popa, and Y. Velegrakis. Clio: Schema Mapping Creation and Data Exange. *Conceptual Modeling: Foundations and Applications*, page 236, 2009.

[7] Y. Velegrakis, R. J. Miller, and J. Mylopoulos. Representing and Querying Data Transformations. In *ICDE*, pages 81–92, 2005.