An Overview of the LLUNATIC System

Floris Geerts [1], Giansalvatore Mecca [2],
Paolo Papotti [3], and Donatello Santoro [2,4]

[1] University of Antwerp – Antwerp, Belgium
[2] Università della Basilicata – Potenza, Italy
[3] Qatar Computing Research Institute (QCRI) – Doha, Qatar
[4] Università Roma Tre – Roma, Italy

(Discussion Paper)

1 Introduction

Data transformation and data cleaning are two very important research and application problems. Data transformation, or data exchange [7], relies on declarative *schema mappings* to translate and integrate data coming from one or more source schemas into a different target schema. Data cleaning, or data repairing [8], uses declarative *data-quality rules* in order to detect and remove errors and inconsistencies from the data.

It is widely recognised that whenever mappings among different sources are in place, there is a strong need to clean and repair data. Despite this need, database research has so far investigated schema mappings and data repairing essentially in isolation.

We present the LLUNATIC [11, 12] mapping and cleaning system, the first comprehensive proposal to handle schema mappings and data repairing in a uniform way. LLUNATIC is based on the intuition that transforming and cleaning data can be seen as different facets of the same problem, unified by their declarative nature. This declarative approach that allowed us to incorporate unique features into the system and apply it to wide variety of application scenarios.

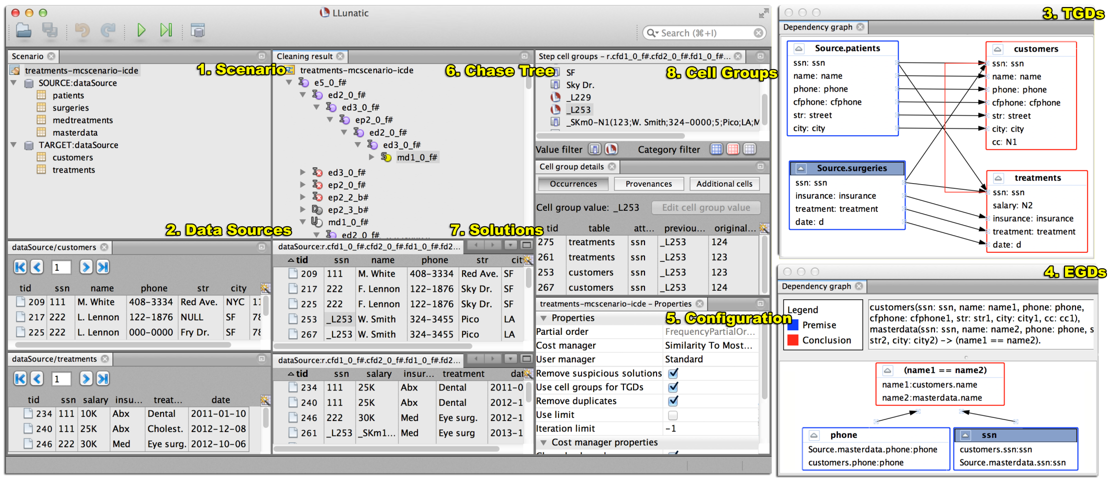
1.1 Variety of Scenarios

LLUNATIC is the first system that supports three kinds of scenarios:

Type 1: schema-mapping scenarios in the spirit of [7], with one or more source databases, and a target database that is related to the sources by a set of schema mappings; integrity constraints can be imposed over the target; given a set of source instances, the goal is to generate a valid instance of the target according to the mappings.

Type 2: data repairing scenarios in the spirit of [8], with one target database, possibly multiple *authoritative* source tables with highly curated data, used to model the so called *master-data* [13], and a set of data-quality rules over the target; these may include functional dependencies, conditional functional dependencies, editing rules, but also inclusion dependencies and conditional inclusion dependencies; here, we are given an instance of the target that is dirty with respect to the constraints, and want to generate a clean instance.

Type 3: mapping and cleaning scenarios, that combine and generalize the two above; here we have multiple sources that may include master-data, one target, a set of mappings that relate the target to the sources, and a rich set of data quality constraints over the target. Examples of such scenarios are in [12].



dataSource/customers

tid	ssn	name	phone	str	city
209	111	M. White	408-3334	Red Ave.	NYC
217	222	L. Lennon	122-1876	NULL	SF
225	222	L. Lennon	000-0000	Fry Dr.	SF

dataSource/treatments

tid	ssn	salary	insu...	treat...	date
234	111	10K	Abx	Dental	2011-01-10
240	111	25K	Abx	Cholest.	2012-12-08
246	222	30K	Med	Eye surg.	2012-10-06

dataSource:r.cfd1_0_f#.cfd2_0_f#.fd1_0_f#.fd2...

tid	ssn	name	phone	str	cit
209	111	M. White	408-3334	Red Ave.	SF
217	222	F. Lennon	122-1876	Sky Dr.	SF
225	222	F. Lennon	122-1876	Sky Dr.	SF
253	_L253	W. Smith	324-3455	Pico	LA
267	_L253	W. Smith	324-3455	Pico	LA

dataSource:r.cfd1_0_f#.cfd2_0_f#.fd1_0_f#.fd2...

tid	ssn	salary	insur...	treatment	dat
234	111	25K	Abx	Dental	2011-0
240	111	25K	Abx	Dental	2012-1
246	222	30K	Med	Eye surg.	2012-1
261	_L253	_SKm1...	Med	Eye surg	2013-1

Cell group value: _L253

tid	table	att...	previou...	original...
275	treatments	ssn	_L253	124
261	treatments	ssn	_L253	123
253	customers	ssn	_L253	123
267	customers	ssn	_L253	124

treatments-mcscenario-icde - Properties

Properties	
Partial order	FrequencyPartialOr...
Cost manager	Similarity To Most...
User manager	Standard
Remove suspicious solutions	✓
Use cell groups for TGDs	✓
Remove duplicates	✓
Use limit	☐
Iteration limit	-1
Cost manager properties	

Source.patients: ssn: ssn, name: name, phone: phone, cfphone: cfphone, str: street, city: city

Source.surgeries: ssn: ssn, insurance: insurance, treatment: treatment, date: d

customers: ssn: ssn, name: name, phone: phone, cfphone: cfphone, str: street, city: city, cc: N1

treatments: ssn: ssn, salary: N2, insurance: insurance, treatment: treatment, date: d

customers(ssn: ssn, name: name1, phone: phone, cfphone: cfphone1, str: str1, city: city1, cc: cc1), masterdata(ssn: ssn, name: name2, phone: phone, s str2, city: city2) -> (name1 == name2).

(name1 == name2): name1:customers.name, name2:masterdata.name

phone: Source.masterdata.phone:phone, customers.phone:phone

ssn: customers.ssn:ssn, Source.masterdata.ssn:ssn

Fig. 1: LLUNATIC in action

The latter kind of scenarios illustrate the main novelty of our approach: on the one side they model the problem of exchanging and repairing data as a single process, and on the other side they conservatively extend and unify type 1 and 2 scenarios, which are the focus of previous approaches. We want to emphasize, however, that our approach brings some interesting extensions also to type 1 and type 2 scenarios. For example, type 1 scenarios generalize traditional data exchange scenarios since we allow for non empty target databases. Type 2 scenarios generalize what it is typically found in data repairing papers, since the vast majority of papers about data repairing have only considered single tables with functional dependencies and their variants, and disregarded foreign keys (i.e., inclusion constraints).

1.2 A Scalable and General Chase Engine

LLUNATIC lays its foundations on two main intuitions. The first one is that the language of embedded dependencies, i.e., *tuple-generating dependencies* and *equality-generating dependencies* [7] can be the basis for a common language for schema mappings and data-quality constraints. The second one is that executing mappings to exchange data and enforcing data quality constraints can be handled nicely under a single, formal semantics. In this, LLUNATIC clearly departs from many previous data-repairing algorithms, that have often relied on rather ad hoc semantics.

Although this extension is far from trivial, it has an interesting consequence: solutions to LLUNATIC scenarios can be operationally generated using a *chase procedure*. In fact, at the core of the system stands a powerful disk-based chase engine, capable of generating parallel-chase trees and multiple solutions. A key feature of the engine is its good scalability in large scenarios. To this end, we developed several optimizations[11, 12] that significantly improve execution times, despite the complexity of the chase.

Notice that, when applied to complex constraints that go beyond simple source-to-target tgds, previous implementations of the chase have had little fortune in scaling to large databases [16]. Furthermore, data-repairing algorithms were developed mainly in main-memory [11], to speed up repair operations that can be expensive on disk; as a consequence, due to memory limitations, these solutions cannot handle large databases.

On the contrary, by modeling the repairing process within its fast chase engine, LLUNATIC can truly repair vast amounts of data. We may therefore say that LLUNATIC is the first general-purpose disk-based chase engine that scales nicely to large scenarios.

As it is well known, the chase is a principled algorithm that plays a key role in many important database-related problems: in addition to data exchange and computing certain answers, these include query minimization, query rewriting using views, and constraint implication. Given its generality, LLUNATIC can be effectively used in all of these settings.

1.3 Organization of the Paper

We believe that LLUNATIC represents an important contribution to both data exchange and data repairing, and using this system users can practically learn key lessons with respect to the technical aspects of the problem. We better substantiate this claim in the next sections. More specifically, in Section 2 we illustrate the main features of the prototype. In doing this, given the focus of this paper, we deliberately choose to omit many of the technical details that are fully documented in the published papers about the system [11, 12]. These papers also describe complex example scenarios that illustrate the wide applicability of LLUNATIC. We rather concentrate on a description of the system from the functional perspective. Based on this, in Section 3, we illustrate a number of key lessons learned using the system, that illustrate what are the main challenges in mapping and cleaning scenarios and how the system solves them.

2 Features of the System

LLUNATIC comes with a GUI, reported in Figure 1, that allows users to easily specify a scenario, explore initial instances and configure the core aspects of the repair process. To start, users specify a scenario, along the lines discussed in Section 1, as follows:

(i) zero or more source databases $\mathcal{S}_1, \ldots, \mathcal{S}_n$; among these, users may indicate some *authoritative sources*, i.e., databases that contain high-quality data, as it happens for master-data, that can be used to improve the mapping and cleaning phase;

(ii) a target database $\mathcal{T}$, that does not need to be empty;

(iii) a (possibly empty) set of source-to-target tgds (s-t tgds) Σ_{st}, that will be used to move data in the source databases to the target;

(iv) a set of data-quality constraints Σ_t over the target, under the form of *cleaning egds* and *extended tgds* [12]; these are extended versions of the classical tgds and egds that generalize most of the constraints used in the literature.

Given a set of source instances $I_0, \ldots, I_n$, and a target instance J_0, for each scenario LLUNATIC computes a set of *minimal solutions*, i.e., target instances $J_1, \ldots J_k$ that satisfy the constraints in Σ_{st} and Σ_t, and "minimally change" the original target instance J_0. To do this, it runs a parallel-chase procedure, that generates a chase tree, as shown in Figure 1. Leaves in the chase tree are solutions. The real power of LLUNATIC stands in the possibility to control the solution-generation process in a very fine way, as discussed in the following paragraphs.

2.1 User-Specified Preference Rules

Data-repairing algorithms in the literature (see *e.g.,* [8] for a survey) try to repair a dirty database by making the smallest number of changes to its cells. Here, with *cell* we denote a tuple-attribute pair $t.A$, i.e., the value of attribute A in the tuple identified by t. There are various minimality conditions for repairs, and repairing algorithms are centered around these notions.

Consider, for example, a simple table Emp with several attributes, the first ones being the *ssn* and *name* of the employee, and a functional dependency d : *ssn* $\rightarrow$ *name* over Emp. Given a dirty instance $J = \{t_1\colon \mathsf{Emp}(123, John, \ldots), t_2\colon \mathsf{Emp}(123, Jack, \ldots)\}$, there are many possible way to repair the database. For example, $J_1 = \{t_1\colon \mathsf{Emp}(123, John, \ldots),\ t_2\colon \mathsf{Emp}(123, John, \ldots)\}$, $J_2 = \{t_1\colon \mathsf{Emp}(123, Jack, \ldots), t_2\colon \mathsf{Emp}(123, Jack, \ldots)\}$ are both possible repairs.

Also $J_3 = \{t_1\colon \mathsf{Emp}(123, Mark, \ldots), t_2\colon \mathsf{Emp}(123, Mark, \ldots)\}$ is a repair, but it is not minimal, since it requires to change two different cells, while a single cell-change is sufficient for J_1 and J_2. There are, however, many other minimal repairs, even for this very simple example. Two of these are as follows: $J_4 = \{t_1\colon \mathsf{Emp}(456, John, \ldots), t_2\colon \mathsf{Emp}(123, Jack, \ldots)\}$, $J_5 = \{t_1\colon \mathsf{Emp}(123, John, \ldots), t_2\colon \mathsf{Emp}(789, Jack, \ldots)\}$. Here we change the data in a *backward way* by falsifying the premise of the dependency. This, in turn, shows why any chase-based solution to these problems cannot be limited to a single chase sequence, and rather needs to generate a *tree* of repairs.

Traditionally, as long as repairs involve a minimal number of changes, they are considered as equally acceptable. However, our example above shows that the minimality criterion is rather weak, and algorithms often choose a repair arbitrarily. LLUNATIC is based on a different philosophy. Informally speaking, its semantics is built on top of a notion of *upgrade*: a repair is a solution as long as it improves the quality of the original database. Improvements cannot be made arbitrarily. On the contrary, a change to the database is considered an upgrade only when it unequivocally improves the data.

To specify when changes are actually upgrades, users can declaratively specify *preference rules*. Consider our example above, and assume that the third attribute of Emp, *conf*, is a measure of the confidence associated with values in the two tuples, and the dirty database is the following: $J = \{t_1\colon \mathsf{Emp}(123, John, 0.4), t_2\colon \mathsf{Emp}(123, Jack, 0.7)\}$. Then, it is natural to state a preference rule saying that a value for the *name*-attribute of Emp should be preferred to another as soon as its confidence is higher. In LLUNATIC this is easily done by a few clicks that identify *conf* as the *ordering attribute* for *name* in Emp. Given this rule, J_1 and J_2 are not equally acceptable as a repair any longer. More specifically, $J_2 = \{t_1\colon \mathsf{Emp}(123, Jack, \ldots), t_2\colon \mathsf{Emp}(123, Jack, \ldots)\}$ is an upgrade of J, while J_1 is not.

By specifying preference rules, LLUNATIC users may associate a *partial order* with the possible repairs of a dirty database. In turn, this yields an elegant notion of a solution: it is a minimal upgrade of the original database that satisfies the constraints. Notice that, as discussed in [11], ordering attributes and partial orders nicely generalize most of the repair-selection strategies that were previously proposed in the literature. Even more important, they unify them under an easy-to-use and effective user-interaction framework. This allows us to nicely model also other forms of preferred values, like, for example master-data. In fact, even assuming that no confidence is available, another possible way to upgrade the database is to rely on a master-data table that states that the name of the employee with *ssn* $= 123$ must be $Jack$.

2.2 User Inputs

In those cases in which no preference rule is available, LLUNATIC does not make arbitrary choices, and rather marks conflicts so that users may resolve them later on.

In order to mark conflicts, LLUNATIC uses two powerful tools, called *lluns* and *cell groups*. Lluns, along with cell groups, are essentially variables with lineage. A llun is a distinguished symbol, L_i, distinct from nulls and constants, that is introduced whenever there is no clear way to upgrade the dirty database by changing a cell to a new constant. Considering again our example above, assume that no preference rule has been stated, or, also, that the two confidence values are equal. Then, the only acceptable way to upgrade the database according to the semantics of LLUNATIC is to introduce a llun as follows: $J_4 = \{t_1\colon \mathsf{Emp}(123, L, 0.4), t_2\colon \mathsf{Emp}(123, L, 0.4)\}$. Here, L represents a value that may be either $John$ or $Jack$, or even a different constant as long as it ensures that it resolves the conflict and upgrades the quality of the database. Unfortunately, given that no preference rule is available, this value is currently unknown, and only an expert user may identify it, possibly at a later time.

Lluns are different from the ordinary variables used in previous approaches due to their relationship to cell-groups. Cell-groups are the building blocks used by LLUNATIC to specify repairs. They not only specify how to change the cells of the database, but also carry full provenance information for the changes, in terms of (a) original values of the cells in the target database; (b) relationships to source and master-data values, if these exist; and (c) user interventions to repair the cells.

Based on this, LLUNATIC offers powerful features to collect user-inputs. First, it is allows to declaratively specify when the chase should be (temporarily) paused to collect inputs from the user by plugging *user managers* into the chase. A user manager is a declarative condition over the chase tree that stops the chase and asks the user for input. Examples are: stop at every new llun; stop after a llun with at least k occurrences; stop at every new chase node that contains lluns etc, just to name a few.

When the chase stops and the user is invoked, s/he may pick up a node in the chase tree, consult its history in terms of changes to the original database, inspect the lluns that have been introduced, and analyze the associated cell groups. Based on this, informed decisions are taken in order to remove lluns and replace them with the appropriate constants.

2.3 Tree-Pruning Strategies

A final, important aspect is related to the complexity of the chase. It is well known that a data repairing process may have to consider an exponential number of repairs, even with a small number of constraints. LLUNATIC was conceived to generate chase trees that may – in principle – correspond to the full space of repairs. However, this exhaustive approach only works on toy examples.

In fact, algorithms in the literature [8] have used many different heuristics in order to speed-up the search for repairs. These range from minimizing the cost of changes to the database, random sampling the space of repairs, to introducing a notion of certain regions to restrict the portions of the database to repair.

On real-life examples, it is crucial to appropriately prune the chase tree in order to guarantee good computation times. LLUNATIC strives to provide the best compromise between an exploration of the space of repairs that is more systematic and thorough that previous algorithms, and a good scalability on large mapping and repairing problems. A key feature, in this respect, are *cost managers*. Cost managers are predicates over the chase tree that a user may specify in order to accept some of the nodes and refuse others. They generalize all of the heuristics listed above, and many more.

Working with cost managers concretely allows users to explore the trade-offs between the quality of repairs, and the cost of their generation. Even more important, cost managers, user-specified preference rules, and user-inputs give a fine-grained control over the solution-generation process, and can be used to learn important lessons, as discussed in the next section.

3 Experiences with the System

In this section we present a number of lessons learned using LLUNATIC over scenarios of different kinds and different sizes. These scenarios come from a number of research projects that our groups have handled in the last few years on (a) integration of Web-available data about events; (b) building integrated rankings of journals and conferences; (c) extracting and integrating Web entities across large collections of sites. In the following paragraphs we illustrate four main experiences, and each of which will demonstrate a number of key aspects.

a. The Benefits of an Integrated Semantics We use the first set of scenarios to discuss an important question, namely: is it really necessary to develop a new semantics for mapping and cleaning together? In the end, these two steps have been studied for years, albeit in isolation, and each of them has a nice semantics and algorithms. Why can't we simply pipeline an existing schema-mapping algorithm and a data-repairing algorithm? We have formally proven that such an approach does not work in general [12]. To show this in a practical way, LLUNATIC can be configured to run with several semantics. One of them is the result of pipelining a standard chase algorithm for tgds, with some of the popular algorithms to repair functional dependencies. By running the system with this semantics on our scenarios, we notice that even in simple cases constraints interact in such a way that alternating the execution of mappings and the repairing of data quality constraints does not terminate. In addition, when the pipelining process terminates, the quality of the solutions computed by this procedure are quite poor, and definitely worse than those generated by LLUNATIC.

b. The Importance of Preference Rules The second set of scenarios is devoted to discussing the importance of user-specified preference rules. LLUNATIC implements various semantics, and may simulate several of the data-repairing algorithms in the literature. Using these alternative semantics, it is clear the impact of the design choices in some of these algorithms on the quality of repairs. On the contrary, whenever users provide clear preference rules – under the form or ordering attributes, or master-data, or even standardization rules – LLUNATIC uses the partial order that these induce on the repairs to systematically upgrade the database, thus improving the quality of the target database in a more principled way. Aside from quality, the adoption of a clear partial order has a strong impact on the scalability of the chase. Intuitively, when the system is able to tell the best way to repair the data for a conflict, it may at the same time discard the many undesirable alternatives, and therefore reduce the size of the chase tree.

c. How to Handle User Inputs There are, however, cases in which no preference rule may dictate how to solve a conflict. We discussed in the previous section how LLUNATIC handles these cases by introducing lluns. Solutions that contain lluns are essentially "partial solutions", in the sense that a user must intervene to pick up the right constant value for each llun. As we discussed, LLUNATIC allows for a wide variety of strategies to collect user inputs and remove lluns. Our goal is to show how these inputs impact the chase. Recall that LLUNATIC computes chase trees. These trees may have considerable sizes, and each node in the tree is a partial repair, and therefore corresponds in principle to a replica of the entire database. We have adopted numerous sophisticated techniques to reduce the cost of computing and storing chase trees, including a super-fast representation system called *delta databases* conceived explicitly for this purpose [11]. Still, computing trees of large sizes may be slow.

In this respect, we want to discuss two main notions: the first one is the fact that lluns and cell groups, together, provide a natural and effective basis to support users in their choices. The second one is that by appropriately providing inputs it is possible to drastically prune the size of the chase tree. Figure 2 shows one of our experiments: we run the chase for the same scenario several times, each with an increasing number of inputs provided by the user. With no user inputs, the chase tree counts over 130 leafs, i.e., possible solutions. With as little as 10 inputs, the tree collapses to a single solution. In general we notice that small quantities of inputs from the user may significantly prune the size of the chase tree, and therefore speed-up the computation of solutions.

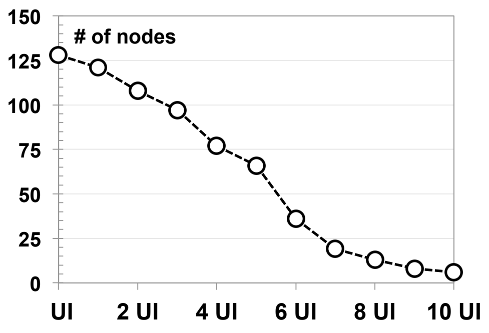


Fig. 2: Impact of user-inputs on the size of the chase

d. When does the Chase Scale? As we mentioned, besides providing a novel strategy to handle mapping and cleaning in an integrated fashion, LLUNATIC is also a general purpose chase engine, capable of handling complex dependencies. It therefore can also be used in other application scenarios. Being a well known and well studied approach to databases, the chase opens a wealth of opportunities for the tool. However, properly implementing the chase has a number of subtleties. We showed in [12] how it performs on scenarios of different size and complexity.

4 Related Work

There has been a host of work on both data exchange and data quality management (see [1] and [8] for recent surveys, respectively). Our mapping & cleaning approach presented in [12] is applicable to general classes of constraints and provides an elegant notion of solution. This is an extension of our earlier work on cleaning scenarios [11] by accommodating for mappings, and work on data exchange [7]. As discussed in [11], cleaning scenarios incorporate many data cleaning approaches including [4, 5, 9, 10, 3]. The same holds for mapping & cleaning scenarios. Furthermore, some of the ingredients of our scenarios are inspired by, but different from, features of other repairing approaches (e.g., repairing based on both premise and conclusion of constraints [5, 3], cells [3, 4], groups of cells [4], partial orders and its incorporation in the chase [2]). As previously observed, these approaches support limited classes of constraints. A flexible data quality system was recently proposed [6] which allows user-defined constraints but does not allow tgds. Early attempts to reason about the interaction of mappings and functional dependencies, and on the scalability of the chase were done in [14, 15].

References

1. M. Arenas, P. Barceló, L. Libkin, and F. Murlak. *Relational and XML Data Exchange*. Morgan & Claypool, 2010.
2. L. Bertossi, S. Kolahi, and L. Lakshmanan. Data Cleaning and Query Answering with Matching Dependencies and Matching Functions. In *ICDT*, pages 268–279, 2011.
3. G. Beskales, I. F. Ilyas, and L. Golab. Sampling the repairs of functional dependency violations under hard constraints. *PVLDB*, 3:197–207, 2010.
4. P. Bohannon, M. Flaster, W. Fan, and R. Rastogi. A cost-based model and effective heuristic for repairing constraints by value modification. In *SIGMOD*, pages 143–154, 2005.
5. G. Cong, W. Fan, F. Geerts, X. Jia, and S. Ma. Improving data quality: Consistency and accuracy. In *VLDB*, pages 315–326, 2007.
6. M. Dallachiesa, A. Ebaid, A. Eldawy, A. K. Elmagarmid, I. Ilyas, M. Ouzzani, and N. Tang. Nadeef: a commodity data cleaning system. In *SIGMOD*, pages 541–552, 2013.
7. R. Fagin, P. Kolaitis, R. Miller, and L. Popa. Data Exchange: Semantics and Query Answering. *TCS*, 336(1):89–124, 2005.
8. W. Fan and F. Geerts. *Foundations of Data Quality Management*. Morgan & Claypool, 2012.
9. W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Towards certain fixes with editing rules and master data. *PVLDB*, 3(1):173–184, 2010.
10. W. Fan, J. Li, S. Ma, N. Tang, and W. Yu. Interaction Between Record Matching and Data Repairing. In *SIGMOD*, pages 469–480, 2011.
11. F. Geerts, G. Mecca, P. Papotti, and D. Santoro. The LLUNATIC Data-Cleaning Framework. *Proceedings of the VLDB Endowment*, 6(9):625–636, 2013.
12. F. Geerts, G. Mecca, P. Papotti, and D. Santoro. Mapping and Cleaning. In *International Conference on Database Engineering (ICDE)*, 232–243, 2014.
13. D. Loshin. *Master Data Management*. Knowl. Integrity, Inc., 2009.
14. B. Marnette, G. Mecca, and P. Papotti. Scalable data exchange with functional dependencies. *Proceedings of the VLDB Endowment*, 3(1):105–116, 2010.
15. B. Marnette, G. Mecca, P. Papotti, S. Raunich, and D. Santoro. ++Spicy: an OpenSource Tool for Second-Generation Schema Mapping and Data Exchange. *Proceedings of the VLDB Endowment*, 4(12):1438–1441, 2011.
16. G. Mecca, P. Papotti, and S. Raunich. Core Schema Mappings: Scalable Core Computations in Data Exchange. *Information Systems*, 37(7):677–711, 2012.