Change Management in Large Information Infrastructures – Representing and Analyzing Arbitrary Metadata

Boris Stumm
stumm@informatik.uni-kl.de

Stefan Dessloch
dessloch@informatik.uni-kl.de

Abstract: With information infrastructures getting more and more complex, it becomes necessary to give automated support for managing the evolution of the infrastructure. If changes are detected in a single system, the potential impact on other systems has to be calculated and appropriate countermeasures have to be initiated to prevent failures and data corruption that span several systems. This is the goal of Caro, our approach to change impact analysis in large information infrastructures. In this paper we present how we model the metadata of information systems to make a global analysis possible, regardless of the data models used, and how the analysis process works.

1 Introduction

In today's businesses, information infrastructures are getting more and more complex. There are many heterogeneous systems with a manifold of mutual dependencies leading to unmanageability of the overall infrastructure. New dependencies between existing systems evolve and new systems are added. Generally, there is no central management of all systems.

Small, local changes to system metadata can have a major impact on a company-wide scale due to the dependencies between systems. This can lead to failures of large parts of the information infrastructure. Probably all companies have some sort of change management to prevent this from happening. However, the more complex the information infrastructure gets, the more difficult it gets to ensure the correct operation of all systems. It is necessary to automate as much of the change management as possible. This will avoid human failures, and in general will be faster than manually keeping track of all dependencies between systems. For our generic approach, we use a broad interpretation of the terms *system*, *change*, and *metadata*. A system is an abstract concept which provides services that can be used by other systems, and in turn may depend on other systems, too. Concrete systems could be DBMS, libraries, websites, or end user applications. The metadata of a system consists of every information that could, if it is changed, influence the correct operation of other systems. A classic example for metadata are database schemas, but also APIs, file locations, or system configurations. Even non-functional aspects like quality and performance assertions are included in our definition of metadata. Our goal is to be able to process all forms of metadata, to make automated CIA as precise as possible.

Problem Statement The core of change management is the analysis of the potential impact of a local change on other systems. This is called *change impact analysis* (CIA). Ideally, the analysis is performed *before* applying a change (*preventive CIA*) to prevent failures. However, this is not always possible, if the changing system does not know about or does not care about a dependent system because it belongs to a different organizational domain. A good example here is a web data integration system relying on screen scraping of external websites. It is then necessary to perform *reactive CIA*. The faster this happens, the more likely it is to avoid or reduce the impact of system failures from happening.

There exist several problem areas that have to be coped with to perform automated, generic CIA at a company-wide scope:

- The heterogeneity of the metadata makes a direct analysis practically impossible, since an analyzer would have to understand all existing metadata models or formats.
- For analysis, metadata has to be provided in an explicit way. In some cases, this is not a problem; for example, SQL databases provide a standardized way to read schema information and file system structures can be easily scanned. In many cases this is more difficult, because the systems provide no appropriate way to read the metadata, or metadata is only implicitly defined in compiled code. Some metadata, like non-functional aspects, may not be extractable automatically at all. Thus, in CIA it has to be assumed that the provided metadata is not complete.
- System autonomy and organizational constraints lead, on one hand, to failure or non-applicability of preventive CIA, and on the other hand, make reactive CIA more difficult because changes may go undetected for a long time.

Contribution In this paper, we present a meta-model used to store arbitrary, heterogeneous metadata, and an analysis component for a fine-grained and customizable impact analysis. In practice, it is impossible to resolve the mentioned problems of metadata incompleteness and system autonomy. Therefore, our principle is to work with what we have, and not to require the input metadata to be of a certain quality. Of course, if the provided metadata is incomplete, the analysis results will not be as exact and reliable as otherwise. We want to avoid overlooking a potential problem in all cases. Therefore, with incomplete metadata, the probability of more false alarms increases. In the worst case, when it is only known that system A accesses system B "in some way", every change of B, regardless of its real impact, will result system A to be marked as impacted by a change, and may trigger counteractions. With more complete metadata, analysis results will improve, reducing the probability of false positives. This "best effort" approach makes Caro applicable to arbitrary environments. The trade-off between the effort of providing complete metadata and the quality of the analysis results can be determined separately for each case.

Related Work In the context of information integration, much research has been done. Some approaches are complementary to ours, and others are similar to Caro in some aspects. The most important distinguishing features of Caro are its genericity, robustness and scope. It makes no assumptions about the environment it operates in, and can be used for any scenario where change impact analysis is necessary.

Dorda et al. [DSS04] present an approach which is similar to Caro with respect to the problems addressed. However, the solution they propose is different in two fundamental points: They require a central documentation (or metadata) repository and a strict process policy. This constrains their approach to scenarios where it is feasible to have a central repository and to enforce adherence to defined processes. While they want to avoid integration clusters, we think that such a clustering (and thus decentralization) in large enterprise information systems cannot be avoided.

Deruelle et al. [DBGN99] present another approach to change impact analysis. They use a multigraph and change propagation rules for analysis, which is very similar to Caro. Their approach has several limitations: The focus lies on preventive CIA, thus they lack a framework to support reactive CIA. Apparently, they do not consider the problem of incomplete metadata. Also, their meta-model and rules are rather specialized, which makes the extension to support other data models and change types more difficult than with Caro.

Various other approaches to CIA in information systems exist that are limited with respect to the supported data models [Ke02] or scope and support of exact analysis [MAL+05]. The concepts of change impact analysis in software systems [BA96, Aji95, RT01] are similar to the ones we use. However, the models and analysis procedures focus on the elements that are found in software: methods, signatures, classes, attributes and so on. In addition, CIA for software systems is usually done preventively. Aspects of heterogeneity, metadata incompleteness and distribution are not as relevant as they are in information systems.

Research done in the field of schema evolution [Rod92, AFK+04, ZR98], schema matching [RB01, MRB04, MP99] or model management [Ber03] are complementary to our approach. Especially the latter approaches are used to plan and realize integration, generally between only two or a small group of systems, as well as adapt systems to changing requirements. Caro is not designed for use in the initial stages of an integration project. It will take the results of such a project, namely the dependencies between the systems that were created based on schema matches or mapping definitions, and monitor them for changes. When a change occurs, Caro will analyze the impact of it. Depending on the results, different actions may be taken. The most simple thing to do is to notify a responsible person, or to shut down the system to prevent further damage. Also, Caro may interface other information integration tools and provide analysis results to them, e.g. for automatic repair. Caro focuses on the monitoring of systems participating in the overall information infrastructure and the detection of the global impact of changes. As such, it "fills the gap" to an overall management of a heterogeneous integrated environment .

The Object Management Group (OMG) defined with the Common Warehouse Metamodel (CWM) [CWM03] a way to relate elements of different data models to each other with respect to structure. While our type system is similar to CWM in some places, the focus in Caro is to express change impact semantics, not schema structure. It is easy to to use CWM or other existing models in Caro, as we show in section 3.

The algorithms we use for CIA are heavily based on graph matching [Bun00]. We use a modified version of the CSI-Algorithm [KH04], which is in turn an improved version of Ullman's algorithm [Ull76].

Paper Structure This paper is structured as follows. First, in section 2, we give a short overview of Caro, and position this contribution in the larger context of the Caro architecture as well as a high-level overview about our meta model and the analysis process. The formal foundations for model and analysis follow in section 3. Our current research prototype is presented in section 4. Besides the implementation itself, we discuss some performance issues there, too. Finally, we conclude in section 5 and give an outlook on further work to be done in this field of research.

2 Caro Overview

Caro is an approach to change management in large information system infrastructures. Basically, Caro monitors all participating information systems. If changes occur, they are analyzed for their impact on the rest of the information infrastructure. In figure 1, the three-layered architecture of Caro is depicted. The lowest layer consists of the information systems themselves. The middle layer consists of the metadata agents (MDAs). For each system, there is a metadata agent responsible for it[1]. An MDA monitors an information system and notifies the server in the upper layer of changes. It is also responsible for transforming the systems metadata into the common format which is introduced in the next section. Finally, if during change impact analysis potential problems are detected, the MDAs are responsible for initiating countermeasures ranging from notifying the administrator to shutting down the system. The upper layer of our architecture consists of the Caro servers. There, the inter-system dependencies are stored and the analysis process is coordinated.

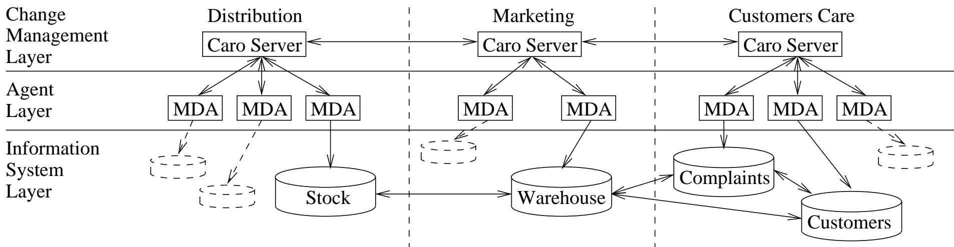

Figure 1: Distributed Caro architecture

We assume that it is not feasible to have a single Caro server responsible for a whole information infrastructure. Due to organizational circumstances, it may be necessary to run more than one Caro server, as shown in figure 1. Between different departments each running a Caro server, there exists a single connection point to other departments, which simplifies setup and communication greatly in comparison to a centralized approach.

Caro servers can be used for preventive as well as reactive CIA:

[1] A single MDA implementation may be responsible for more than one IS, but logically, there is one MDA per information system.

– DBMS (system DB)

```
create table employees (
    id integer primary key,
    name varchar(50),
    salary integer
);
```

```
create table projects (
    pid integer primary key,
    p_name varchar(50),
    deadline date,
    manager integer references employees(id)
);
```

– reporting application (system RA)

```
    select e.name, count(e.name)
    from employees e, projects p
    where e.id = p.manager
    group by e.name;
```

```
    select *
    from projects
    where deadline < current_date + 30;
```

Figure 2: Example SQL schema

- For preventive CIA, users can propose a hypothetical change request to Caro, and get the CIA results without actually modifying the systems. This way, the user can make sure that either his proposal does not affect other systems, or that the other systems administrators know in advance of the change. A common use of this is when modifying database schemas by adding, deleting or altering views and tables.
- In the case of reactive CIA, an MDA notifies its server of a change in the information system it monitors, and impact analysis is initiated. If a potential impact on other systems was found, the corresponding MDAs will be notified and can then take adequate actions. Caro reacts to a change, and while it is not always possible to undo or mask the change, every system that is potentially affected will at least be notified. This way, change impact is detected quickly after the change, making it possible to prevent possible damage like data corruption or system failures. For example, in a screen scraping scenario, if the website from which information is extracted is changed, several actions are possible, depending on the abilities of the impacted systems and the availability of corresponding tools. The most simple action is to notify an administrator and/or to shut down dependent systems. There may also be means to notify users that current results may not be accurate due to a change, or even modify the extraction procedure to match the new website structure. Caro acts as agent that mediates between systems, users, administrators and tools.

In this paper, we focus on the model we use to represent the heterogeneous metadata of the information systems, and how the metadata is analyzed for changes. The metadata extraction and transformation as well as the monitoring tasks that the MDA has to handle are not in the scope of this paper. However, in section 4 we discuss some of our experiences with extracting and transforming SQL metadata in our prototype.

To illustrate our meta-model and the analysis process, we will use the example depicted in figure 2. There is a DBMS named DB with a schema consisting of two tables, and a reporting application RA that uses two different select statements to access the data in the DBMS.

```
create table employees (                create table projects (
    id integer primary key,                 pid integer primary key,
    name varchar(50),                       p_name varchar(50),
    salary integer                          deadline date,
);                                          manager integer references employees(id)
                                        );
```

Figure 3: Usage of the reporting application

Systems and Dependencies Our approach targets large information infrastructures with many systems that are related to each other by their dependencies. Each system is described by its metadata, which also contains the description of the dependencies a system may have. We define the terms *usage* and *provision* to specify the parts of a system's metadata which express dependency information.

A provision is the part of the metadata that captures services provided to some other system. For a DBMS, this is the part of the schema that is accessible from other systems. For web services, the accompanying WSDL file can be seen as provision. Since access rights may not be the same for everyone, a system can have more than one provision. The counterpart of a provision is the usage. In a usage, a system specifies the services provided by another system that it depends on. Usages are always subsets of the corresponding provision. We avoided the well known term "export schema" and "import schema" for provisions and usages, since they may contain more than only schema information.

In our example, the provision of DB consists of the complete schema in figure 2. The usage of RA (in respect to DB) is depicted in figure 3. Only the elements highlighted in bold belong to the usage. The problem of incomplete metadata becomes visible here. The two select statements do not specify RA's metadata completely. Probably RA makes assumptions about the data types of the columns and about primary keys. The only information we can deduct from the select statements is that the `deadline` column is expected to be of type `date` because it is compared with a date in the where clause. Also, we have to assume that RA needs all columns of the `projects` table. That means that if any column is added to or deleted from the `projects` table in DB, Caro must assume that it has an impact on RA. If, however, the `salary` column of the `employee` table is deleted, or an Clim is added, Caro should recognize that this has no impact on RA.

Of course, it is not realistic to have such a fine-grained documentation of all schema dependencies in an large and complex environment. But in general, it is at last possible to find out about dependencies at the system level, which is "better than nothing". The likelihood of false alarms will rise, but changes will not go undetected. In some areas, however, it is feasible to get fine-grained dependency information, e.g., by extracting it from BPEL [BPE06] documents.

The Caro Meta-Model The model we use to represent metadata is a simple digraph with typed nodes. Each node represents a metadata element or a relationship between two metadata elements, similar to the E/R model [Che76]. Nodes are *atomic information units*. So, each change that occurs can be described by a set of added and deleted nodes and edges. Figure 4 shows how metadata of DB and RA as well as their provision and usage

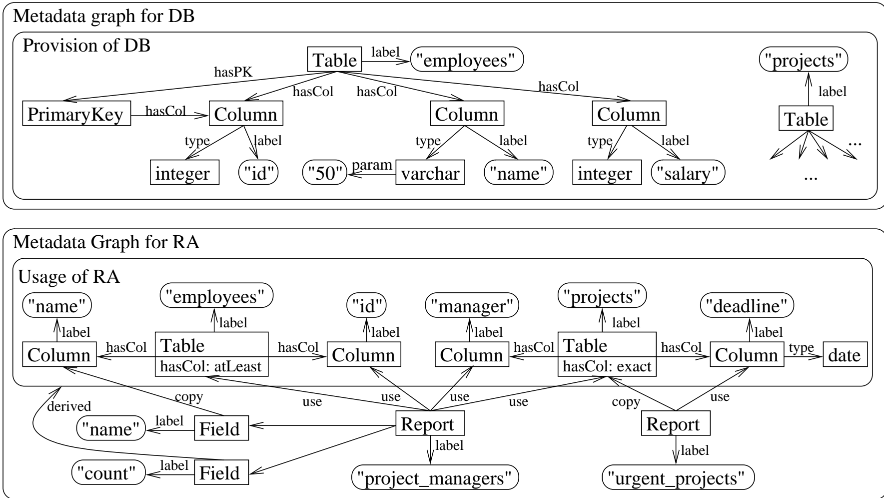

Figure 4: Metadata graphs for the example in figure 2

(which are just subgraphs) look like in our graph model. Te reporting application is a proprietary application with a custom meta model, indicated by the Report and Field node types. Note that the notation used is informal and serves only for illustration purposes. Nodes of type `Literal` are shown in boxes with rounded corners and in double quotes, all other nodes are labeled with their type. For the sake of simplicity, we omitted the relationship nodes and assigned their type to the edges of the graph. Element nodes belonging to a usage may have a *completeness specification* for incoming and outgoing relationships. Values can be `atLeast` or `exact`, both shown in the figure. The `employee` table node specifies that it needs *at least* the `hasCol` relationships that are specified in the usage, and thus, analysis wont care if other columns are added or deleted. The `projects` table, however, specifies that it needs *exactly* the `hasCol` relationships that are specified. Thus, if any column is added or deleted here, the analysis will recognize a potential problem. The usage is connected to the rest of the graph via various *derivation* relationships like `derived`, `use` or `copy` that express the different semantics of the dependencies. These types are defined in the change impact description model defined below.

It is straightforward to represent arbitrary metadata in our model, since any type system may be used. For a generic change impact analysis, these metadata representations will be useless, since they are not meta model independent. For meta model independence, Caro defines the *change impact system description model* (CISDM) which expresses the semantics of metadata nodes that are relevant for CIA. Figure 5 shows some of the types the CISDM defines in a UML-like [UML03] notation. Relationship nodes always connect two element nodes (which can be of type `Element` or `Literal`. For each property that is relevant to CIA, subclasses of `Element` and `Relationship` are defined. The associations between the subclasses restrict which element types a relationship type may connect. Concrete meta models are connected to the CISDM via inheritance as it is shown in figure 6 for a small subset of the SQL and XML meta models. In contrast to other

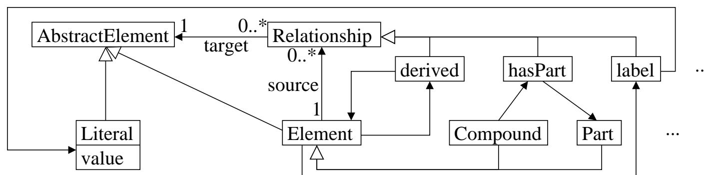

Figure 5: Some elements from the change impact system description model

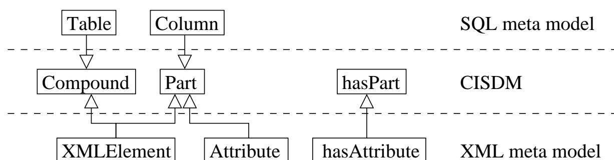

Figure 6: Concrete data models inheriting from the CISDM

existing meta models, like the CWM [CWM03], the CISDM does not focus on an abstraction layer for metadata exchange, modeling similar concepts in different meta models as subclasses of a common superclass. The CISDM only captures the semantics needed for CIA, which simplifies the addition of new meta models. The analysis process only works on the CISDM types, and does not mind the underlying data model.

Performing Impact Analysis Each node in a metadata graph has a set valued status attribute, the *change status*, which will be set during CIA. Initially, the status attribute of all nodes is empty. If a change occurs, either proposed by a user or determined by an MDA, each node that was added to the graph gets the status `added`, and each deleted node is not actually deleted but gets the status `deleted`. After that, CIA is conducted as a two-step process that may cascade over several metadata graphs, if there is a chain of dependent systems. First, *intra-model analysis* performed, followed by *inter-model analysis*:

- In the intra-model analysis phase, the status attribute of all nodes affected by the change is set accordingly. For example, if a part is deleted from a compound, the compound will get the status `partDeleted`. In our example, that happens with the table where a column was deleted.
- The inter-model analysis just copies all change status values of a provision to the corresponding nodes in the related usage.
- The process repeats for the metadata graph of the dependent systems, and stops when at some point all potentially impacted nodes are marked accordingly.

We illustrate this using our example scenario. Consider the changes at top of figure 7. We included only the actually changed parts of the provision. Nodes marked with $\ominus$ have been deleted, and nodes marked with $\oplus$ have been added (1). In the intra-model analysis for DB, the `Table` elements will get the status `partDeleted`/`partAdded` (2). The subsequent inter-model analysis copies all change status values to the usage of RA (3). Without preceding intra-model analysis, no status values would have been copied,

alter table employees drop column salary; alter table projects add column description varchar(2048);

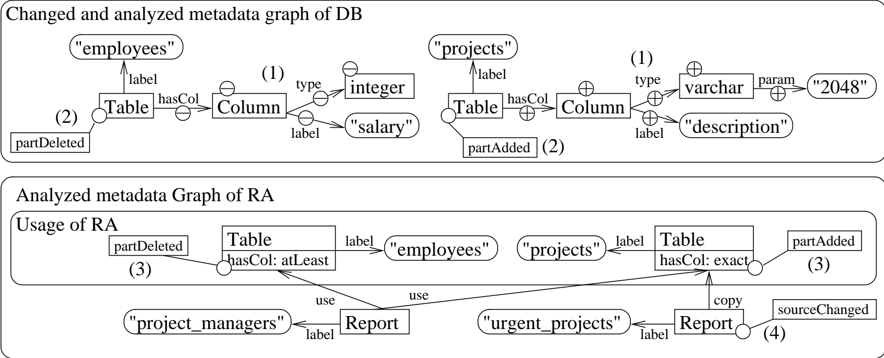

Figure 7: Metadata graphs after analysis

since the usage does not mention the `salary` column, and cannot know yet about the `description` column. As a last step, intra-model analysis is performed in the metadata graph of RA, adding the status `sourceChanged` to the report `urgent_projects`.

Technically, intra-model analysis is done by applying a set of change propagation rules to the metadata graph until no more rules can fire. As example, a rule might look for all added parts, and mark the corresponding compound as with the `partAdded` status, as it happened in our example. In the example, only one report was marked as problematic, which is correct, but it is not obvious why the other report was *not* marked. The rule responsible for this can be stated in an informal way as “if a relationship was added or removed from an element, and the completion specification for this relationship is `exact`, add the status `sourceChanged` to all elements connected via a `derived` or `copy` relationship”. In chapter 3 we will show a formal way to specify such rules. Inter-model analysis works by first matching corresponding usages and provisions, and then copying the change status values to the usages.

3 Formal Model

After having given an overview of our model and analysis process, how metadata is represented and how the generic CIA process works in the previous section we are now ready to present the formal model of the Caro metadata graph and the analysis process. In all formulas, lower case letters stand for single elements of a set, upper case letters represent sets, and letters in gothic stand for sets of sets. If an element is a tuple $x = (a, b, c, \ldots)$, subelements are referred to as a_x, b_x, c_x etc. Figure 8 summarizes the naming convention for variables used throughout the paper and may serve as a reference.

v, V	nodes	k, K	node matches	g, G	metadata graphs
$e, (a, b), E$	edges	c	conclusions	s, S	change status
t, T	types	$m, (l, u), M, \mathfrak{M}, \mathfrak{S}$	graph matches	a	attribute function
r, R	rules	p, P	premises	Pr, Us	provisions, usages

Figure 8: Variable naming scheme

3.1 Meta-Model

Let us first describe how the meta-model concepts informally introduced in section 2 can be captured in a formal metadata model. A digraph d is an ordered pair $d = (V_d, E_d)$. V_d is the set of nodes, and $E_d \subseteq V_d \times V_d$ with $E_d = \{(v_a, v_b) | v_a, v_b \in V_d, v_a \neq v_b\}$ a set of ordered pairs defining the edges. A subgraph $s \subseteq d$ is defined as $s = (V_s, E_s)$ with $V_s \subseteq V_d, E_s \subseteq V_s \times V_s, E_s \subseteq E_d$. A metadata graph g is a graph with additional properties $g = (V_g, E_g, a_g, Pr_g, Us_g)$ such that $d = (V_g, E_g)$ is a digraph; G is the set of all metadata graphs. Pr and Us are the sets of provision and usage subgraphs: $Pr \subseteq \wp(d)$, $Us \subseteq \wp(d)$, with $\wp(d)$ being the set of all subgraphs of d. The attribute function a will be defined later.

In the formal model, a node type as introduced in figure 5 is just a named set of nodes. We call the set of all nodes $Node$, and thus, all node types T are a subset of $Node$. The basic node types are defined as follows:

$$AbstractElement = Element \cup Literal; AbstractElement \subset Node$$
$$Relationship \subset Node$$
$$Relationship \cap AbstractElement = \emptyset; Literal \cap Element = \emptyset$$

Attributes of a node are defined by the attribute function $a : AN \times V \to AV_{AN}$. AN is the set of attribute names $AN = \{\text{type}, \text{status}, \text{value}, \text{completeness}_{\text{in/out}}\}$, and AV_{AN} is the set of possible values for the corresponding attribute. The function a is defined as

$$a(\text{type}, v) \mapsto \quad T; T \subset Node$$
$$a(\text{status}, v) \mapsto \begin{cases} \text{if } v \notin Literal : S_v; S_v \subseteq S \\ \text{else: undefined} \end{cases}$$
$$a(\text{value}, v) \mapsto \begin{cases} \text{if } v \in Literal : z; z \in String \\ \text{else: undefined} \end{cases}$$
$$a(\text{completeness}_{\text{in/out}}, v) \mapsto \begin{cases} \text{if } v \in Element : Cs_v; Cs_v \subseteq Cs \\ \text{else: undefined.} \end{cases}$$

For $a(x, v)$ we also write $x(v)$. Each node is assigned a *most specific type*. Only literal nodes have a value (the literal they represent), which is always a

string. The status is a subset of all possible status values S, with $S = \{\text{added}, \text{deleted}, \text{partAdded}, \text{partDeleted}, \text{partChanged}, \text{sourceChanged}, \ldots\}$. Literals have no status attribute, because they represent only values.

The completeness attribute is only specified for element nodes; relationship nodes are always complete according to equation (2). The completeness specification Cs is defined as $Cs = \{(T, cs | T \subseteq Relationship, cs \in \{atLeast, exact\}\}$. To avoid conflicts, subtypes must have the same or a stronger completeness value, and each type may only have one completeness value: Let Cs_v be a completeness specification, then $\forall x, y \in Cs$:

$$T_x \subset T_y \implies cs_x = cs_y \vee cs_x = \text{exact}$$
$$x \neq y \implies T_x \neq T_y$$

Generally, all nodes have an identity, and nodes are not shared between metadata graphs. Literals are an exception to this, because literal values are unique: $\forall n_1, n_2 \in Literal : value(n_1) = value(n_2) \Leftrightarrow n_1 = n_2$. Therefore, nodes in the intersection of the sets of nodes of two graphs are always literal nodes:

$$\forall x, y \in G : x \neq y \implies V_x \cap V_y \subseteq Literal \tag{1}$$

Graph edges can only connect elements with relationships and vice versa, enforcing a bipartite element/relationship graph. $\forall (a, b) \in E$:

$$a \in Element \implies b \in Relationship,$$
$$a \in Relationship \implies b \in AbstractElement$$
$$a \notin Literal.$$

Subtypes can be further restricted, as we discussed in section 2. Moreover, all relationship nodes must have exactly one incoming and one outgoing edge:

$$\begin{aligned}
&\forall v \in Relationship \exists a, b \in V : (a, v) \in E \wedge (v, b) \in E && \text{(at least one edge)} \\
&\forall a, b \in AbstractElement, v \in Relationship : && (2) \\
&\qquad (a, v), (b, v) \in E \implies a = b && \text{(at most one edge)} \\
&\qquad (v, a), (v, b) \in E \implies a = b &&
\end{aligned}$$

3.2 Analysis

As already outlined in the previous section, analysis is performed using intra-model and inter-model analysis steps. These are formalized in the this subsection. In both steps, matching two graphs is an important operation. In intra-model analysis it is used to find out where to apply rules, and in inter-model analysis it is used to match a usage against a provision graph.

We define a *node match relation* $K \subseteq Node \times Node$:

$$K = \{(u, l) | u, l \in Node \wedge (u, l \notin Literal \wedge u \neq l) \vee (u, l \in Literal \wedge u = l)\}.$$

We chose u and l are mnemonics for *upper* and *lower*, if we imagine an "upper" graph being matched against a "lower" graph. Literals match only themselves, whereas other nodes match only different nodes. This is due to definition (1). Depending on the context, a refined $K_x \subset K$ will be defined. Note that K_x is *not* necessarily symmetric!

Let $g_u, g_l \in G$ be two graphs. We define a *graph match relation* $\mathfrak{M}_{g_u,g_l} \subseteq \wp(K)$ as

$$\begin{aligned}\mathfrak{M}_{g_u,g_l} = \{M|M \in \wp(K) \wedge \forall m_1, m_2 \in M :& \\ (u_{m_1}, u_{m_2}) \in E_{g_u} \wedge (l_{m_1}, l_{m_2}) \in E_{g_l} & \quad \text{(match edges)} \\ u_{m_1} = u_{m_2} \Leftrightarrow l_{m_1} = l_{m_2}\}. & \quad \text{(bijectivity)}\end{aligned}$$

For analysis, only the set of matches of greatest common subgraphs $\mathfrak{S}_{g_u,g_l} \subseteq \mathfrak{M}_{g_u,g_l}$ with $\mathfrak{S}_{g_u,g_l} = \{M|M \in \mathfrak{M}_{g_u,g_l} \wedge \nexists M' \in \mathfrak{M}_{g_u,g_l} : |M'| > |M|\}$ is relevant.

If $|\mathfrak{S}_{g_u,g_l}| = 1$, the match between g_u and g_l is called unique, if $|\mathfrak{S}_{g_u,g_l}| > 1$ it is ambiguous. In the following, we refer to $M_{u,l} \in \mathfrak{S}_{g_u,g_l}$ as *match* between g_u and g_l.

A matched subgraph $s_{l_M} = (V_s, E_s)$ of g_l (and accordingly, s_{u_M}) itself is defined as

$$V_s = \{v|v \in V_{g_l} \wedge \exists v_u \in g_u : (v, v_u) \in M\} \text{ and } E_s = \{e|e_a, e_b \in V_s \wedge e \in E_g\}.$$

Generally, finding the greatest common subgraph of two graphs is a hard problem. The metadata graphs that are analyzed by Caro, however, have a very regular structure. That is true for the example graph in figure 4 and we argue that in all practical cases, the structure will be similarly regular. This speeds up the matching process significantly. Our benchmarks (see section 4) show that the approach is fast enough to be used in real world scenarios.

Intra-Model Analysis The intra-model analysis works by applying a set of propagation rules onto the graph g_c. These rules add change status values, which may enable other rules to fire. This continues until no more rules can fire. Rules have a premise, basically a digraph that is matched against the metadata graph, and a conclusion specifying a node and the status to add to this node. First, we define the node match relation for intra-model analysis:

$$\begin{aligned}K_r = \{k|k \in K \wedge \text{type}(k_u) \supseteq \text{type}(k_l) \wedge \\ \text{status}(k_u) \subseteq \text{status}(k_l) \wedge \\ \text{completeness}_{\text{in/out}}(k_u) \subseteq \text{completeness}_{\text{in/out}}(k_l)\}.\end{aligned}$$

A rule node k_u matches a graph node k_l if its type is a supertype of the graph nodes type, its status is a subset of the graph nodes status, and the completeness specification also is compatible. This node match relation is not symmetric.

A rule r is a pair $r = (p, c)$. The premise p is a triple $p = (V_p, E_p, a_p)$, V_p, E_p, a_p defined like for metadata graphs. The conclusion c is a pair $c = (v, s)$ with $v \in V_p, s \in S$. R is the set of all rules. As an example, figure 9 shows the rule that marks compounds as changed if parts were added, as it happened in step (2) in figure 7. Rules are applied to the graph

$$\begin{aligned}
&r = (p, c); && a_p(\text{type}, v_1) \mapsto \textit{Compound} \\
&p = (V_p, E_p, a_p); && a_p(\text{type}, v_2) \mapsto \textit{hasPart} \\
&c = (v_1, \text{partAdded}); && a_p(\text{type}, v_3) \mapsto \textit{Part} \\
&V_p = \{v_1, v_2, v_3\}; && a_p(\text{status}, v) \mapsto \begin{cases} \text{if } v = v_3 : \{\text{added}\} \\ \text{else: } \emptyset \end{cases} \\
&E_p = \{(v_1, v_2), (v_2, v_3)\}; && a_p(\text{completeness}_{\text{in/out}}, v) \mapsto \emptyset
\end{aligned}$$

Figure 9: Example rule

with the $apply : G \to G$ function: $apply(g) \mapsto g' = (V_g, E_g, a'_g, Pr_g, Us_g)$ such that

$$a'_g(attr, v) \mapsto \begin{cases} \text{if } (attr = \text{status} \wedge \exists r \in R, \exists M \in \mathfrak{S}_{g_r,g} : & \text{(there is a matching rule)} \\ \qquad |M| = |V_{p_r}| \wedge & \text{(rule matches completely)} \\ \qquad \exists m \in M : m_u = v_{c_r} \wedge m_l = v) : & \text{(rule changes this node)} \\ \quad a_g(\text{status}, v) \cup s_{c_r} \\ \text{else } a_g(attr, v). \end{cases}$$

We define the analysis result relation $A \subseteq G \times G$, with

$$A = \{(g, g') | g, g' \in G \wedge apply(g') = g' \wedge \exists n \in \mathbb{N} : \underbrace{apply \circ apply \circ \ldots \circ apply}_{\text{n times}}(g) = g'\}.$$

Inter-Model Analysis In the inter-model analysis, a provision and a corresponding usage are matched, and the change status values are copied from provision to usage. Let g_u, g_l be two metadata graphs where system u depends on system l. Then inter-model analysis transforms g_u to $g'_u = (V_{g_u}, E_{g_u}, a'_{g_u}, Pr_{g_u}, Us_{g_u})$.

Let $pr_l \in Pr_l$ be a provision subgraph, $us_u \in Us_u$ the corresponding usage subgraph, and $M \in \mathfrak{S}_{us_u,pr_l}$ a graph match. The used node match relation is $K_t = \{k | k \in K \wedge type(k_u) = type(k_l) \wedge \text{added} \notin status(k_l)\}$. That is, the usage is matched against the provision without the added nodes to avoid ambiguous matches.

The new attribute function a'_{g_u} is defined as

$$a'_{g_u}(attr, v) \mapsto \begin{cases} \text{if } attr = \text{status} \wedge \exists m \in M, v_l \in V_{g_u} : v = m_u \wedge v_l = m_l : \\ \qquad a_{g_l}(attr, v_l) \\ \text{else } a_{g_u}(attr, v) \end{cases}$$

Computational issues With subgraph matching, there exists the possibility of ambiguous subgraphs, that is, there is no unique greatest common subgraph according to the chosen K. In intra-model analysis, if $|\mathfrak{S}_{g_r,g}| > 1$ the rule r matches more than one time,

which is expected, since there may be more than one place that was changed. In inter-model analysis, however, $|\mathfrak{S}_{g_u,g_l}| > 1$ means that there is an ambiguity in matching the graphs. In theory, this leads to non-deterministic behaviour of the analysis, which we want to avoid. In practice, the issue is not so severe, since the usage is a subgraph of the provision, and will be matched completely. An ambiguous match indicates an error in the usage or provision subgraph.

Another important issue is the computability and termination of our algorithms. We require $|Node| < \infty, |R| < \infty, |S| < \infty$. All other sets of elements used in our definitions are built using set operations and thus are finite, too. That makes it possible to evaluate all expressions by enumerating all possibilities.

Only for the analysis result relation A we need to show that it is computable, i.e., the process terminates. This is intuitively clear, since the *apply* function only adds monotonically change status values. To prove it, we have to show that

$$\forall g \in G\, \exists n \in \mathbb{N}\, \exists g' \in G : apply(g') = g' \wedge \underbrace{apply \circ apply \circ \cdots \circ apply}_{\text{n times}}(g) = g'. \quad (3)$$

We define $scount : G \to \mathbb{N}$ as $scount(g) \mapsto \sum_{v \in V_g} |a_g(status, v)|$.

Since $\forall v \in V_g : a_{apply(g)}(status, v) \supseteq a(status, v)$, also $scount(apply(g)) \geq scount(g)$, that is, *apply* is monotonic. Further, $\forall v : a(status, v) \subseteq S$. That means $max(|a(status, v)| = |S|)$. Therefore, $\forall g \in G : scount(g) \leq |V_g| \cdot |S|$, so there is an upper bound for $scount(g)$, which proves (3).

3.3 Extensibility

While we try to handle as many cases as possible in the CISDM and the corresponding analysis rules, there will naturally be cases where our rules and types do not suffice or even give wrong results with certain metadata. The analyzer makes no difference between CISDM types or data models subclassing it. And while the provided rules only work on CISDM types, user defined rules can use any node type in their premises. Users can extend/customize Caro by providing their own types, nodes, change status values and rules. This way they can provide missing functionality, and even “override” the system provided analysis rules by just ignoring the change status values they produce. Of course, this is not advisable in a distributed environment, since it has to be made sure that change status values propagated to other systems are understood by their metadata agents. However, in some cases customizing may be necessary, since we cannot anticipate all possible scenarios.

4 Implementation and Performance

The current prototype implementation of Caro is written in Java. The analysis component is capable to do intra- and inter-model analysis. The prototype is already in use in a small setup: The computer science department of the University of Kaiserlautern uses a custom-made exam registration system, PAS, for students. PAS is a web application using the PostgreSQL [PGS05] DBMS to store data. To ensure that the schema in the production system matches the current implementation, we deployed Caro. In terms of our model, the DBMS is one system, and the web application another, dependent system. Caro extracts metadata from the information schema for the DBMS, and parses the DDL and DML statements in the web applications source code. This will support developers by pointing out changes that were made to the schemas since the last update.

In this section, we present the important components of the implementation and give performance results. As base data for all performance tests, we used an SQL schema consisting of 350 tables with a total of 4923 columns, which amounts to 46609 nodes in the metadata graph. The schema is borrowed from HISPOS [HIS], an administration application for universities. For testing, we used a machine with four dual core 1.5 GHz processors, and assigned 1.75 GB ram to the (single threaded) benchmark application. Finally we make some remarks about deployment effort of Caro based on our current experiences.

As we have seen in section 2, metadata agents are responsible for extracting the metadata from the information systems, converting it to the graph model, and sending it to the change manager. Currently we are able to extract metadata directly from the information schema of an SQL DBMS, or from DDL statements. We also are able to read DML statements. We support tables, views, primary and foreign keys and, to a limited degree, constraints. For data transfer and export, we use GXL [WKR01], an XML format for storing graphs.

```
@prefix rdf:    <http://www.w3.org/1999/02/22-rdf-syntax-ns#>.
@prefix rdfs:   <http://www.w3.org/2000/01/rdf-schema#>.
@prefix owl:    <http://www.w3.org/2002/07/owl#>.
@prefix :       <http://caro.bstumm.de/rdf/Caro/CISDM#>.

:Node a owl:Class.
:Element rdfs:subClassOf :Node,           [a owl:Restriction; owl:onProperty :outrange;
                                             owl:allValuesFrom :Relationship].
:Literal rdfs:subClassOf :Node,           [a owl:Restriction; owl:onProperty :outrange;
                                             owl:allValuesFrom owl:Nothing].
:AbstractElement rdfs:subClassOf :Node; owl:unionOf (:Element :Literal).
:Relationship rdfs:subClassOf :Node,      [a owl:Restriction; owl:onProperty :outrange;
                                             owl:allValuesFrom :AbstractElement].
:hasPart rdfs:subClassOf :Relationship,   [a owl:Restriction; owl:onProperty ciam:inrange;
                                             owl:allValuesFrom :Compound],
                                          [a owl:Restriction; owl:onProperty ciam:outrange;
                                             owl:allValuesFrom :Part].
```

Figure 10: Subset of the CISDM definitions

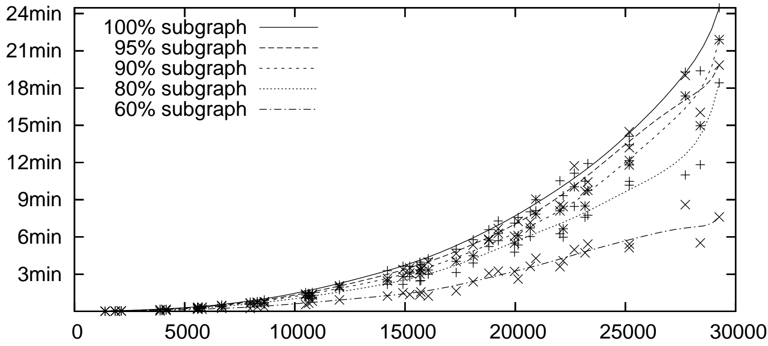

Figure 11: Graph matching benchmark results

Model Implementation The model implementation consists mainly of a simple, hash map based graph class and the type system. Types as well as edge restrictions are defined in OWL [OWL04] for convenience reasons: One can easily define subtype relationships and restrictions for relationship types, and then let the OWL reasoner of the Jena Framework [HP05] infer all implicit knowledge, such as the transitive hull of the subtype relationship. Figure 10 shows a small subset of the type definitions. We use the N3 syntax [N3] here.

Inter-Model Analysis Implementation We use a modified version of Krissinel's and Henrick's CSI algorithm described in [KH04], which is in turn an improved version of Ullman's algorithm [Ull76]. To measure performance, we randomly selected a subset of the HISPOS tables as provision. From this provision, we randomly deleted a certain percentage of tables and columns to construct a usage subgraph. Figure 11 shows the results of our measurements. On the x-axis, the number of nodes in the provision is plotted, the y-axis shows how many seconds it took to match the subgraphs. The different lines denote averages of test runs with different sized usage subgraphs. The benchmark results indicate a roughly quadratic runtime, which is the best case for the algorithm we chose. To improve performance, we can make some assumptions about the metadata graphs that are to be matched. First, there exists a set of already matched nodes to start with, namely the literals. As we required in our definitions, literal nodes are unique, and if two literal nodes in different graphs have the same value, it is always a match. Second, even if metadata graphs are not completely connected, in each part there will exist at least one literal. This is reasonable for metadata, since in general, most metadata elements, such as tables, columns, elements, etc., are named. Finally, the metadata graph is "almost a tree", meaning that the number of relationship nodes is similar to the number of element nodes, and that there is a spanning tree representing a main hierarchy.

These assumptions are reasonable for all practically relevant metadata graphs. By exploiting them (which our current implementation does not do very well), much larger metadata graphs could be handled as it is currently the case. This is a part of our ongoing work.

```
Set<Node> changedNodes = getAllChangedNodes();
while (!changedNodes.isEmpty()) {
    Node v = changedNodes.removeANode();
    for (Rule r: getRules()) {
        Set<Match> M = getMatches(r, v);
        for (Match m: M) {
            Set<Node> changed = apply(r, m);
            changedNodes.add(changed);
}   }   }
```

Figure 12: Rule matching algorithm

Intra-Model Analysis Implementation We argue that most practical rules will be similar to the example in figure 9 in complexity, having one node where a change status test is made and another node where a change status is added. Our current implementation is based on this fact. The basic matching algorithm is sketched in figure 12.

First, we get the set of changed nodes as obtained through change determination or inter-model analysis. We then iterate over this node set. We take out a node v and check if there is a rule with matches containing v. If so, we apply the conclusion to each match, and put all nodes that were changed by the conclusion back into the set of changed nodes. Each node can be modified at most once by each rule, which means that each node can be put back into the set at most $|R|$ times. The while loop will thus be executed at most $|V| \times |R|$ times. The outer for loop will be executed $|R|$ times. Assuming rule premises consisting of three nodes like in figure 9, there will be at most in the order of $|V|$ matches. The matching process itself has the same complexity. It makes at most two hops through the graph, and since relationship nodes always are connected to two other nodes, the search space is bound by $|V|$. This leads to an overall complexity of our implementation of at most $O(n^2 r^2)$ in the average case, with only simple rules. More complex rules with more than a few nodes will be more expensive to process in terms of computing time. To measure rule matching performance, we used the HISPOS schema again. We marked one node as added, and used a set of "flooding" rules: each rule matched two nodes of any type, connected by an edge, where one node had a certain change status X, and then added the status Y to the other node. The next rule would then look for nodes with status Y and add status Z, and so on, so that finally, all nodes in the graph will be marked with all possible change status values. Figure 13 shows the results of the benchmarking. It shows that the time needed increases proportional to the graph size and the number of rules that are matched.

The complete HISPOS schema with 350 tables and a ruleset of 159 rules took around four minutes to be analyzed. Since this is a worst case scenario, and most times, changes will cover only a small part of a metadata graph, intra-model analysis can be done in an interactive fashion, which is important for preventive CIA.

Deployment Effort While the Caro model and analysis component works independent of any data model and is able to analyze arbitrary metadata graphs, the effort to extract and transform metadata into the graph model must be considered, too. Although discussing this issue is not the focus of this paper, we want to share some experiences that we got during the implementation of the prototype. Our SQL metadata importer prototype took us about

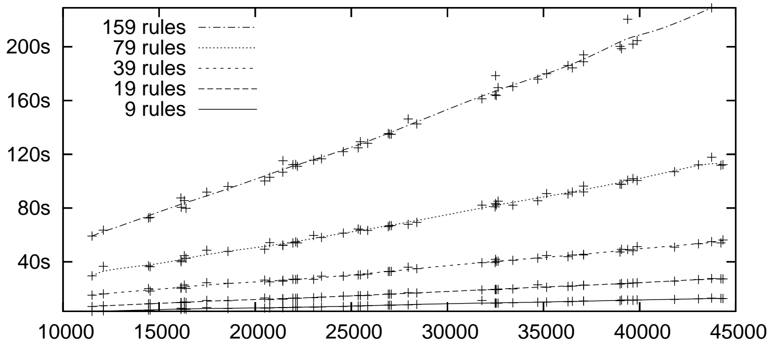

Figure 13: Rule benchmark results

four man days to write, and is currently capable of extracting tables, views, columns, types, primary and foreign keys, check constraints as well as view and constraint dependencies out of an SQL information schema as defined in the SQL:2003 standard [ANS03]. This can serve as a very approximate measure of the effort it takes to extract and transform other metadata, like XML schemas, WSDL files, etc. Import components only have to be written once per data model/system type. Ideally, vendors would directly supply MDAs for their products, which is of course not possible for legacy or home-brew systems. Here, a decision has to be made between extracting and transforming the metadata at a fine-grained level, and therefore getting good analysis results, or to be satisfied with a coarse grained metadata description and a loss of analysis precision. The trade-off is between putting much initial effort into the MDAs, or to later put more effort in detecting the exact problem causes.

5 Conclusion and Outlook

The research area of autonomic computing [KC03] has become increasingly important over the last years. Most times, the scope is local to a single system, or a group of tightly coupled systems, for load balancing or disaster prevention. While the approach we presented in this paper does not directly enable self-* in a system, it contributes to autonomic computing at a larger scale. It provides a mechanism for communication between systems, so each system is aware of changes in its surroundings, which in turn enables the systems to act more autonomically.

Caro performs change impact analysis by storing metadata and analyzing the impact of metadata changes. This happens in a generic way, such that arbitrary data models are supported, ranging from SQL or XML schemas over configuration files, directory structures, APIs to non-functional information like quality or performance assertions. No assumptions are made about any processes for preventive change impact analysis, and at least reactive CIA can always be performed. Our approach also works in a reasonable way if

the provided metadata is incomplete, accepting a higher rate of false positives as trade-off. This is an important fact, because in most real-world scenarios it is not feasible to provide complete metadata descriptions. Thus, our approach is not only generic, but also adapts to the environment where it is deployed. The better the provided input data, the better the analysis results will be, but even with bad input, Caro is able to conduct CIA on a best-effort basis.

We gave a formal description of the our model and analysis process, showing that model and analysis process are well defined. Analysis has shown to be deterministic and computable, which is very important for change impact analysis. The performance benchmarks we conducted show that our approach performs well for reasonable sizes of metadata graphs, and that intra-model analysis is fast enough to be used in an interactive scenario for preventive CIA. Finally, Caro is extensible at all points: Node types, rules and specific change statuses can be added to the model as needed, if the built-in constructs are not sufficient in a certain case.

There are, however, some areas which need to be researched further. We need to look for a way to gather and add change provenance information to the graph to further improve the automated support given by Caro. Also, performance of inter-model analysis could be increased by taking into account more the regular graph structure of metadata graphs. Another important task is to provide interfaces to other approaches which can be used in a complementary way: The results Caro provides can be used in schema evolution tools to adapt systems to the new situation. On the other hand, information gathering tools could be used to help Caro getting the metadata from the systems and watching for changes (e.g., [SSKS95, MAL+05]). This way, we think that our approach can act as a framework binding together many research efforts that often only work in a very constrained environment.

We stated in section 4 that the analysis works fast enough to be used in an interactive fashion. Of course, nobody is able to work interactively with a 45000 node graph. We currently work on an abstracting graph editor which is able to provide a natural (e.g., SQL DDL statements) way of editing a metadata graph. For this, we use style sheets for specifying how the graph should be presented to the user.

References

[AFK+04] P. Andritsos, A. Fuxman, A. Kementsietsidis, R. J. Miller, and Y. Velegrakis. Kanata: Adaptation and Evolution in Data Sharing Systems. *SIGMOD Record*, 33(4), Dec 2004.

[Aji95] S. Ajila. Software Maintenance: An Approach to Impact Analysis of Objects Change. *Software – Practice and Experience*, 25(10):1155–1181, October 1995.

[ANS03] ANSI/ISO. *Information technology – Database languages – SQL – Part 2: Foundation (SQL/Foundation)*, 2003.

[BA96] Shawn A. Bohner and Robert S. Arnold, editors. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996.

[Ber03] Philip A. Bernstein. Applying Model Management to Classical Meta Data Problems. In *Proc. of the 1st Conference on Innovative Data Systems Research (CIDR)*, 2003.

[BPE06] OASIS Open. *Web Services Business Process Execution Language Version 2.0 (public review draft)*, 2006.

[Bun00] Horst Bunke. Graph matching: Theoretical Foundations, Algorithms, and Applications. In *Proc. Vision Interface*, pages 82–88, Motreal, 2000.

[Che76] Peter Pin-Shan Chen. The entity-relationship model – toward a unified view of data. *ACM Trans. Database Syst.*, 1(1):9–36, 1976.

[CWM03] Object Management Group (OMG). *Common Warehouse Metamodel (CWM) Specification Version 1.1*, 2003.

[DBGN99] L. Deruelle, M. Bouneffa, G. Goncalves, and J.-C. Nicolas. Local and Federated Database Schemas Evolution: An Impact Propagation Model. In *Proc. of the 10th International Conference on Database and Expert Systems Applications (DEXA)*, 1999.

[DSS04] Clemens Dorda, Hans-Peter Steiert, and Jürgen Sellentin. Modellbasierter Ansatz zur Anwendungsintegration. *it – Information Technology*, 46(4):200–210, 2004.

[HIS] HISPOS-GX. http://www.his.de/Abt1/HISPOS.

[HP05] Hewlett-Packard. Jena – A Semantic Web Framework for Java, 2005.

[KC03] Jeffrey O. Kephart and David M. Chess. The Vision of Autonomic Computing. *Computer*, 36(1), 2003.

[Ke02] A. Keller and Ch. eNSEL. An Approach for Managing Service Dependencies with XML and the Resource Description Framework. Technical report, IBM, 2002.

[KH04] Evgeny B. Krissinel and Kim Hendrick. Common Subgraph Isomorphism Detection by Backtracking Search. *Software – Practice and Experience*, 34, 2004.

[MAL+05] R. McCann, B. AlShebli, Q. Le, H. Nguyen, L. Vu, and A. Doan. Mapping Maintenance for Data Integration Systems. In *Proceedings of the 31st VLDB Conference*, 2005.

[MP99] Peter McBrien and Alexandra Poulovassilis. Automatic Migration and Wrapping of Database Applications - A Schema Transformation Approach. In *ER*, 1999.

[MRB04] Sergey Melnik, Erhard Rahm, and Philip A. Bernstein. Developing Metadata-Intensive Applications with Rondo. *Journal of Web Semantics*, 1(1), 2004.

[N3] Notation 3. http://www.w3.org/DesignIssues/Notation3.html.

[OWL04] World Wide Web Consortium. *OWL Web Ontology Language Guide*, 2004. Available online: http://www.w3.org/TR/2004/REC-owl-guide-20040210/.

[PGS05] PostgreSQL, 2005. http://www.postgresql.org.

[RB01] Erhard Rahm and Philip A. Bernstein. A survey of approaches to automatic schema matching. *VLDB Journal*, 10:334–350, 2001.

[Rod92] John F. Roddick. Schema Evolution in Database Systems – An Annotated Bibliography. *SIGMOD Record*, 21(4):35–40, 1992.

[RT01] Barbara G. Ryder and Frank Tip. Change Impact Analysis for Object-Oriented Programs. In *Proceedings of PASTE*, 2001.

[SSKS95] L. A. Shklar, A. P. Sheth, V. Kashyap, and K. Shah. InfoHarness: Use of Automatically Generated Metadata for Search and Retrieval of Heterogeneous Information. In *Proceedings CAiSe*, pages 217–230, London, UK, 1995. Springer-Verlag.

[Ull76] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the Association for Computing Machinery*, 23:31–42, 1976.

[UML03] Object Management Group (OMG). *Unified Modeling Language (UML) Specification, Version 1.5*, March 2003.

[WKR01] A. Winter, B. Kullbach, and V. Riediger. An Overview of the GXL Graph Exchange Language. *Software Visualization – International Seminar Dagstuhl Castle*, 2001.

[ZR98] Xin Zhang and Elke A. Rundensteiner. Data Warehouse Maintenance Under Concurrent Schema and Data Updates. Technical report, Worcester Polytechnic Institute, 1998.