Context-aware Storage and Retrieval of Digital Content

Database Model and Schema Considerations for Content Persistence

Hans-Werner Sehring
Namics
Hamburg, Germany
e-mail: hans-werner.sehring@namics.com

***Abstract*—In increasingly many information systems that publish digital content, the documents that are generated for publication are tailored for and delivered to users working in different and varying contexts. To this end, the content from which an actual document is created is dynamically selected with respect to a specific context. The task of content selection incorporates queries to an underlying database that hosts data representing content. Such queries are parameterized with a description of the context at hand. This is particularly true for content management applications, e.g., for websites that are targeted at a user's context. The notion of context comprises various dimensions of parameters like language, location, time, user, and user's device. Most data modeling languages, including programming languages, are not well prepared to cope with variants of content, though. They are designed to manage universal, consistent, and complete sets of data. The Minimalistic Meta Modeling Language (M3L) can be applied as a language for content representation. M3L has proven particularly useful for modeling content in context. Towards an operational M3L execution environment, we are researching mappings to databases of different data models, and for each data model schemas to efficiently store and utilize M3L models. This article discusses such schemas for context-aware data representation and retrieval. The main focus lies on efficiency of queries used for M3L evaluation with the goal of context-dependent content selection. This is achieved by expressing context-aware models, in particular M3L statements, by means of existing persistence technology.**

Keywords-data modeling; data schema; databases; content modeling; context-aware data modeling; content; content management; content management systems; context.

I. INTRODUCTION

In many information systems, e.g., web-based ones, data represents *content* to be incorporated into *documents* that are generated on purpose. More often than not content is required to be queried dynamically on document access, calling for adequate content storage and retrieval. First studies on such content persistence have been reported [1]. This article extends the report on the current state of these database schema investigations.

In the digital society [2], data is required to represent all kinds of content, ranging from structured content of text documents to unstructured, typically binary representations of video and audio content. Content is used for many purposes, the most obvious ones being information and commerce. Content is published by means of documents, often multimedia documents incorporating different media that are interrelated to form hypermedia networks. So-called publication channels offer the medium for one kind of publication, e.g., a website, a document file, or a mobile app. Content is typically represented in a channel-agnostic way in order to support multi- or even omni-channel publishing.

It is quite common to deliver content to users in a way that addresses the *context* in which they are when requesting the content. This may include the channel they are using, the working mode they are in, the history of previous usage scenarios, etc. Targeting content to users' contexts can range from simply arranging content in a specific way, over specifically assembled documents, to content that is synthesized for the current requests. Examples are a prominent display of teasers for content that is assumed to be of interest to the user, the production of documents matching a user's native language, adjustment of document quality based on the current network bandwidth and the receiving device, and creating content that represents some base data in knowledgeable form.

For such content targeting scenarios, data needs to be stored in a way that allows generating different views on the content, mainly by selecting content relevant in a certain context. Data representing all forms of content in such a system, therefore, needs to be attributed with the contexts in which it is applicable or preferred. Obviously, some notion of context is required for such representations [3].

Data modeling and programming languages typically do not exhibit features to represent context and to include it in evaluations. Database management systems, being the backbone of practically every information system, are particularly optimized for one connected set of data that is supposed to be consistent and complete. This means that they are not well equipped for dynamic content production, neither regarding content representation nor efficient context-dependent retrieval.

Data retrieval needs particular attention when content is dynamically assembled depending on some context in which it is requested. For the tasks of context-aware content management, complex collections of data to be used as content are requested frequently. A context-aware schema has

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

to efficiently support the underlying queries that are employed to identify relevant content.

For the discussion of data models, we consider content in contexts as it is expressible using the Minimalistic Meta Modeling Language (M3L). This language allows expressing content in a straightforward way. Being a modeling language, there is no obvious mapping to established data structures, though.

The rest of this paper is organized as follows. Section II reviews related work in the area of context-aware data and content models. Sections III and IV give a brief overview over the M3L and describe those parts of the language that are required for the discussion in this paper. Section III describes the static aspects of the M3L used to define application models. Section IV focuses on the dynamic evaluation of such models. The architecture of a current M3L implementation is discussed in Section V in order to clarify the scope of M3L persistence. Section VI presents a first conceptual model of an internal representation of M3L concepts. Section VII makes this model more concrete by means of logical representations, comparable to the logical view on databases. Aspects of M3L persistence implementations based on different data models are touched in Section VIII. The conclusion and acknowledgment close the paper in Section IX.

II. Related Work

Context is important in the area of content management, but also in other modeling domains. This section names some existing modeling approaches to contextual information.

A. Content Management Products

Most commercial content management products have introduced some notion of context in their models and processes. They utilize context information to *target* content to users. Some use the term *personalization*, which is similar to, but different from contextualization [4].

In most cases, there are publication *rules* associated with content, similar as discussed in [5]. These rules are based on so-called *segments*. Every user is assigned one or more segments. When requesting content, the rules are evaluated for the actual segment(s) in order to select suitable content.

Content authors and editors maintain the content rules. Segments are assigned to users automatically by the systems based on the users' behavior (user interactions), the user journey (e.g., previously visited sites and search terms used for finding the current website), and context information (e.g., device used and location of the user).

Segments offer a rather universal notion of context, though there is no explicit context model.

B. Context-aware Data Models

Parallel to the notion of context used for content, there exists some work on the influence of environments on running applications. In mobile usage scenarios, context refers mainly to such environmental considerations, e.g., network availability, network bandwidth, device, or location.

Context changes are incorporated dynamically into evaluations in these scenarios [6].

Context-awareness is not limited to data models. It is also used for adaptable or adaptive software systems, e.g., to map software configurations to execution environments [7], or to control the behavior of a generic solution [8].

C. Concept-oriented Content Management

Concept-oriented Content Management (CCM) [9] is an approach to manage content reflecting knowledge. Such content does not represent simple facts, but instead is subject to interpretation. Furthermore, the history of things is described by content, not just their latest state.

CCM is not directly concerned about modeling context. Instead, it aims to introduce a form of pragmatics into content modeling that allows users on the one hand to express differing views by means of individual content models, and on the other hand to still communicate by exchanging content between individualized models.

CCM uses a notion of personalization that goes far beyond the one of content management systems (see above).

It is similar to contextualized content usage, although the system does not know about the context of a user. Instead, users carry out personalization (in CCM terms) manually.

A CCM system reacts to model changes and relates model variants to each other. The basis for this ability is systems generation: based on the definitions of users, schemas, APIs, and software modules are generated.

Some aspects of the considerations presented in Section VIII were gained from the research on the generation of CCM modules that map content to external data, e.g., content representations stored in databases.

III. The Minimalistic Meta Modeling Language

The *Minimalistic Meta Modeling Language* (*M3L*, pronounced "mel") is a modeling language that is applicable to a range of modeling tasks. It proved particularly useful for context-aware content modeling [10].

For the purpose of this paper, we only introduce the static aspects of the M3L in this section. Dynamic evaluations that are defined by means of different rules are presented in the subsequent section.

The descriptive power of M3L lies in the fact that the formal semantics is rather abstract. There is no fixed domain semantics connected to M3L definitions. There is also no formal distinction between typical conceptual relationships (specialization, instantiation, entity-attribute, aggregation, materialization, contextualization, etc.).

A. Concept Definitions and References

A M3L definition consists of a series of definitions or references. Each definition starts with a previously unused identifier that is introduced by the definition and may end with a semicolon, e.g.:

```
Person;
```

A reference has the same syntax, but it names an identifier that has already been introduced.

We call the entity named by such an identifier a *concept*.

The keyword `is` introduces an optional reference to a *base concept*, making the newly defined concept a *refinement* of it.

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

A specialization relationship as known from object-oriented modeling is established between the base concept and the newly defined derived concept. This relationship leads to the concepts defined in the context (see below) of the base concept to be visible in the derived concept.

The keyword `is` always has to be followed by either `a`, `an`, or `the`. The keywords `a` and `an` are synonyms for indicating that a classification allows multiple sub-concepts of the base concept:

```
Peter is a Person; John is a Person;
```

There may be more than one base concept. Base concepts can be enumerated in a comma-separated list:

```
PeterTheEmployee is a Person, an Employee;
```

The keyword `the` indicates a closed refinement: there may be only one refinement of the base concept (the currently defined one), e.g.:

```
Peter is the FatherOfJohn;
```

Any further refinement of the base concept(s) leads to the redefinition ("unbinding") of the existing refinements.

Statements about already existing concepts lead to their redefinition. For example, the following expressions define the concept `Peter` in a way equivalent to the above variant:

```
Peter is a Person;
Peter is an Employee;
```

B. *Content and Context Definitions*

Concept definitions as introduced in the preceding section are valid in a context. Definitions like the ones seen so far add concepts to the top of a tree of contexts. Curly brackets open a new context, e.g.:

```
Person { Name is a String; }
Peter is a Person{"Peter Smith" is the Name;}
Employee { Salary is a Number; }
Programmer is an Employee;
PeterTheEmployee is a Peter, a Programmer {
 30000 is the Salary;
}
```

We call the outer concepts the *context* of the inner, and we call the set of inner concepts the *content* of the outer.

In this example, we assume that concepts `String` and `Number` are already defined. The sub-concepts created in context are unique specializations in that context only.

As indicated above, concepts from the context of a concept are inherited by refinements. For example, `Peter` inherits the concept `Name` from `Person`.

M3L has visibility rules that correlate to both contexts and refinements. Each context defines a scope in which defined identifiers are valid. Concepts from outer contexts are visible in inner scopes. For example, in the above example the concept `String` is visible in `Person` because it is defined in the topmost scope. `Salary` is visible in `PeterTheEmployee` because it is defined in `Employee` and the context is inherited. `Salary` is not valid in the topmost context and in `Peter`.

C. *Contextual Amendments*

Concepts can be redefined in contexts. This happens by definitions as those shown above. For example, in the context of `Peter`, the concept `Name` receives a new refinement.

Different aspects of concepts can explicitly be redefined in a context, e.g.:

```
AlternateWorld {
 Peter is a Musician {
  "Peter Miller" is the Name;
 }
}
```

We call a redefinition performed in a context different from that of the original definition a *conceptual amendment*.

In the above example, the contextual variant of `Peter` in the context of `AlternateWorld` is both a `Person` (initial definition) and a `Musician` (additionally defined). The `Name` of the contextual `Peter` has a different refinement.

A redefinition is valid in the context it is defined in, in sub-contexts, and in the context of refinements of the context (since the redefinition is inherited as part of the content).

D. *Concept Narrowing*

There are three important relationships between concepts in M3L.

M3L concept definitions are passed along two axes: through visibility along the nested contexts, and through inheritance along the refinement relationships.

A third form of concept relationship, called *narrowing*, is established by dynamic analysis rather than by static definitions like content and refinement.

For a concept c_1 to be a narrowing of a concept c_2, c_1 and c_2 need to have a common ancestor, and they have to have equal content. Equality in this case means that for each content concept of c_2 there needs to be a concept in c_1's content that has an equal name and the same base classes.

For an example, assume definitions like:

```
Person { Sex; Status; }
MarriedFemalePerson is a Person {
 Female is the Sex;
 Married is the Status;
}
MarriedMalePerson is a Person {
 Male is the Sex;
 Married is the Status;
}
```

With these definitions, a concept

```
Mary is a Person {
 Female is the Sex;
 Married is the Status;
}
```

is a narrowing of `MarriedFemalePerson`, even though it is not a refinement of that concept, and though it introduces separate nested concepts `Female` and `Married`.

E. *Semantic Rule Definitions*

For each concept, one *semantic rule* may be defined.

The syntax for semantic rule definitions is a double turnstile ("|=") followed by a concept definition. A semantic rule follows the content part of a concept definition, if such exists.

A rule's concept definition is not made effective directly, but is used as a prototype for a concept to be created later.

The following example redefines concepts `MarriedFemalePerson` and `MarriedMalePerson`:

```
MarriedFemalePerson is a Person {
 Female is the Sex;
 Married is the Status;
} |= Wife
MarriedMalePerson is a Person {
 Male is the Sex;
 Married is the Status;
} |= Husband
```

The concepts `Wife` and `Husband` are not added directly, but at the time when the parent concept is evaluated. Evaluation is covered by the subsequent section.

Concepts from semantic rules are created and evaluated in different contexts. The concept is instantiated in the same context in which the concept carrying the rule is defined. The context for the evaluation of a rule (evaluation of the newly instantiated concept, that is) is that of the concept for which the rule was defined.

In the example above, the concept `Wife` is created in the root context and is then further evaluated in the context of `MarriedFemalePerson`.

Rules are passed from one concept to another by means of inheritance. They are passed to a concept from (1) concepts the concept is a narrowing of, and (2) from base classes. Inheritance happens in this order: Only if the concept is not a narrowing of a concept with a semantic rule then rules are passed from base concepts.

E.g., `Mary` as defined above evaluates to `Wife`.

F. Syntactic Rule Definitions

Additionally, for each concept one *syntactic rule* may be defined.

Such a rule, like a grammar definition, can be used in two ways: to produce a textual representation from a concept, or to recognize a concept from a textual representation.

A semantic rule consists of a sequence of string literals, concept references, and `the name` expressions that evaluate to the current concept's name.

During evaluation of a syntactic rule, rules of referenced concepts are applied recursively. Concepts without a defined syntactic rule are evaluated to/recognized from their name.

E.g., from definitions

```
WordList {
 Word; Remainder is a WordList;
} |- Word " " Remainder;
OneWordWordList is a WordList |- Word;
Sentence { WordList; } |- WordList "."
HelloWorld is a Sentence {
 Words is the WordList {
  Hello is the Word;
  OneWordWordList is the Remainder {
   World is the Word;
} } }
```

the textual representation

```
Hello World.
```

is produced.

Syntactic rule evaluation is not covered in this article.

IV. Concept Evaluation

As pointed out, there is no fixed generic semantic of M3L constructs. Nevertheless, concrete models receive semantics by means of semantic rules and their evaluation. After definition, each concept (in the root context) is evaluated in a way described in this section.

Concept evaluation is based on (a) narrowing (see Section III.D) and (b) semantic rules (Section III.E).

This section gives a semi-formal description of these means to assign semantics to M3L models. We present as many definitions as are required to derive the main database operations that drive the evaluation process in database-driven M3L implementations.

Throughout this section, let $\mathbb{C}$ be the set of concepts, $\mathbb{S}$ be the set of sets of concepts, and $\mathbb{R}$ be the set of semantic rules. Let $\mathbb{T}$ be the set of root concepts (concepts that do not have another concept as their explicit context).

A. Concept Relationship Access Functions

First, we define typical access functions to the components of a M3L model.

The function *context* returns the context of a concept as defined by a concept definition, or $\perp$, if the given concept is a root concept:

$$\mathit{context}: \mathbb{C}\rightarrow\mathbb{C}. \tag{1}$$

The reverse relation, *content*, returns the content of a concept:

$$\mathit{content}: \mathbb{C}\rightarrow\mathbb{S}: c\mapsto\{c'\in\mathbb{C} \mid \mathit{context}(c')=c\},\ c\neq\perp,$$
$$\mathit{content}: \mathbb{C}\rightarrow\mathbb{S}: \perp\mapsto\mathbb{T}. \tag{2}$$

The *base* relationship maps a concept to its base concepts:

$$\mathit{base}: \mathbb{C}\times\mathbb{C}\rightarrow\mathbb{S}. \tag{3}$$

Since the set of base concepts may be extended by contextual concept amendments, the relation is evaluated relative to a context, given by the context-defining concept (second parameter), or by $\perp$ if base concepts as defined in the root context are requested.

The inverse, the *refine* relationship, maps concepts to the concepts derived from them in a given context x:

$$\mathit{refine}: \mathbb{C}\times\mathbb{C}\rightarrow\mathbb{C}: (c,x)\mapsto\{c'\in\mathbb{C} \mid c'\in \mathit{base}(c,x)\}. \tag{4}$$

Let *semanticRule* be a projection function that returns the semantic rule defined for a concept in a given context x. If none is defined in x or any parent context, the function returns $\perp$.

$$\mathit{semanticRule}: \mathbb{C}\times\mathbb{C}\rightarrow\mathbb{R} \tag{5}$$

Likewise, let *concept* be the function that returns the concept that is defined by a rule definition:

$$\mathit{concept}: \mathbb{R}\rightarrow\mathbb{C}. \tag{6}$$

E.g., for a concept *Concept* in the root context defined as

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

```
Concept |= NewConcept {Content;}
```

the function application *concept(semanticRule(Concept,⊥))* returns *NewConcept*.

B. Computed Relationships

On the basis of the accessor functions defined in the previous subsection, some computed relationships can be defined. In this subsection we define helper functions required to define narrowing in the subsequent subsection, and to finally define evaluation in Section IV.D: the set of transitive base concepts $base^T$, the set of transitive refinements $refine^T$, and the *bottom* of a concept set.

Chained base relationships are retrieved by

$$base^T: \mathbb{C}\times\mathbb{C}\rightarrow\mathbb{S},$$
$$base^T: (c,x)\mapsto base(c,x) \cup \{base^T(c',x) \mid c' \in base(c,x)\}. (7)$$

Likewise, transitive refinement is defined by:

$$refine^T: \mathbb{C}\times\mathbb{C}\rightarrow\mathbb{S},$$
$$refine^T: (c,x)\mapsto \{c\} \cup \{refine^T(c',x) \mid c' \in refine(c,x)\}. (8)$$

The function *bottom* removes concepts from a concept set if these are already subsumed by other contained concepts. These are concepts that are refined by a concept in the set and are themselves not refining that concept:

$$bottom: \mathbb{S}\times\mathbb{C}\rightarrow\mathbb{S},$$
$$bottom: (S,x)\mapsto S \setminus \{c\in S \mid \exists c_2\in S: c_2\in refine^T(c,x) \wedge c\notin refine^T(c_2,x)\}. (9)$$

C. Concept Narrowing

One central point in the process of evaluating concepts is to compute their narrowing. In order to define narrowings, we first introduce some helper functions.

Let R_c be the set of root concepts that (transitively) are base concepts of a concept c, $R_c = \mathbb{T} \cap base^T(c,x)$. A superset of c's narrowing is easily computed using

$$narrowCandiateList: \mathbb{C}\times\mathbb{C}\rightarrow\mathbb{S}$$
$$narrowCandiateList: (c,x)\mapsto \{refine^T(c',x) \mid c' \in R_c\}, (10)$$

meaning that all narrowings are found in the set consisting of all concepts from all content hierarchies to which the concept belongs.

In order to remove candidates for narrowings, helper functions to examine a concept's "type" are required. Two functions help analyzing whether a concept c is a refinement of a base concept b, (interpreted in the context of concept x):

$$hasType: \mathbb{C}\times\mathbb{C}\times\mathbb{C}\rightarrow Bool$$
$$hasType: (c,b,x)\mapsto base^T(c,x) \supseteq base^T(b,x), \quad (11)$$

and whether two concepts c_1 and c_2 are the same with respect to their set of base concepts:

$$sameType: \mathbb{C}\times\mathbb{C}\times\mathbb{C}\rightarrow Bool: (c_1,c_2,x)\mapsto c_2\in base^T(c_1,x) \vee c_1=c_2 \vee hasType(c_1,c_2,x). (12)$$

Besides these static type checks, we also need structural matching of concepts (sometimes called "duck typing" [11]):

$$hasWholeContent: \mathbb{C}\times\mathbb{C}\times\mathbb{C}\rightarrow Bool$$
$$hasWholeContent: (c,candidateBaseConcept,x)\mapsto \forall c_1\in content(candidateBaseConcept): \exists c_2\in content(c): sameType(c_2,c_1,x). (13)$$

The function *hasWholeContent* determines for two concepts c and *candidateBaseConcept* whether (interpreted w.r.t. the context of concept x) the whole content of c is also part of the context of *candiateBaseConcept*, meaning that there is a concept with an equal set of base classes.

With the helper functions (10)-(13) we define the narrowing of a concept c in the context of a concept x as:

$$narrowing: \mathbb{C}\times\mathbb{C}\rightarrow\mathbb{S}: (c,x)\mapsto refine^T(c,x) \cup \{c' \in narrowCandiateList(c,x) \mid hasType(c,c',x) \wedge hasWholeContent(c,c',x)\}. \quad (14)$$

D. Semantic Rule Application and Concept Evaluation

At the core of the concept evaluation lies the productive application of semantic rules as described in Section III.E.

During the evaluation process, semantic rules are applied by instantiating the concept named in a rule. We express this by a function *apply* as

$$apply: \mathbb{R}\times\mathbb{C}\rightarrow \mathbb{C}:$$
$$apply: (r,x)\mapsto concept(r) \text{ in } context(x), \text{ if it exists,}$$
$$apply: (r,x)\mapsto \text{deep copy of } concept(r) \text{ in } context(x), \text{ interpreted in } x, \text{ else.} \quad (15)$$

With narrowing and rule application we can define M3L concept evaluation as

$$evaluate: \mathbb{C}\times\mathbb{C}\rightarrow\mathbb{S},$$
$$evaluate: (c,x)\mapsto bottom(evaluate(apply(semanticRule(narrowing(c,x),x),x),x),x), \text{ if some concept in } narrowing(c,x) \text{ has a rule,}$$
$$evaluate: (c,x)\mapsto bottom(evaluate(refine^T(c,x),x),x), \text{ if some concept in } refine^T(c,x) \text{ has a semantic rule}$$
$$evaluate: (c,x)\mapsto bottom(refine^T(c,x),x), \text{ else.} \quad (16)$$

For the sake of brevity, we use extensions to set-valued parameters to relationships (5), (15), and (16).

V. Anatomy of the M3L Environment

This section outlines the architecture of a first M3L implementation. It is studied here in order to determine base functions that require an efficient implementation for concept evaluation. This leads to the requirements on the persistence layer that is the subject of this article.

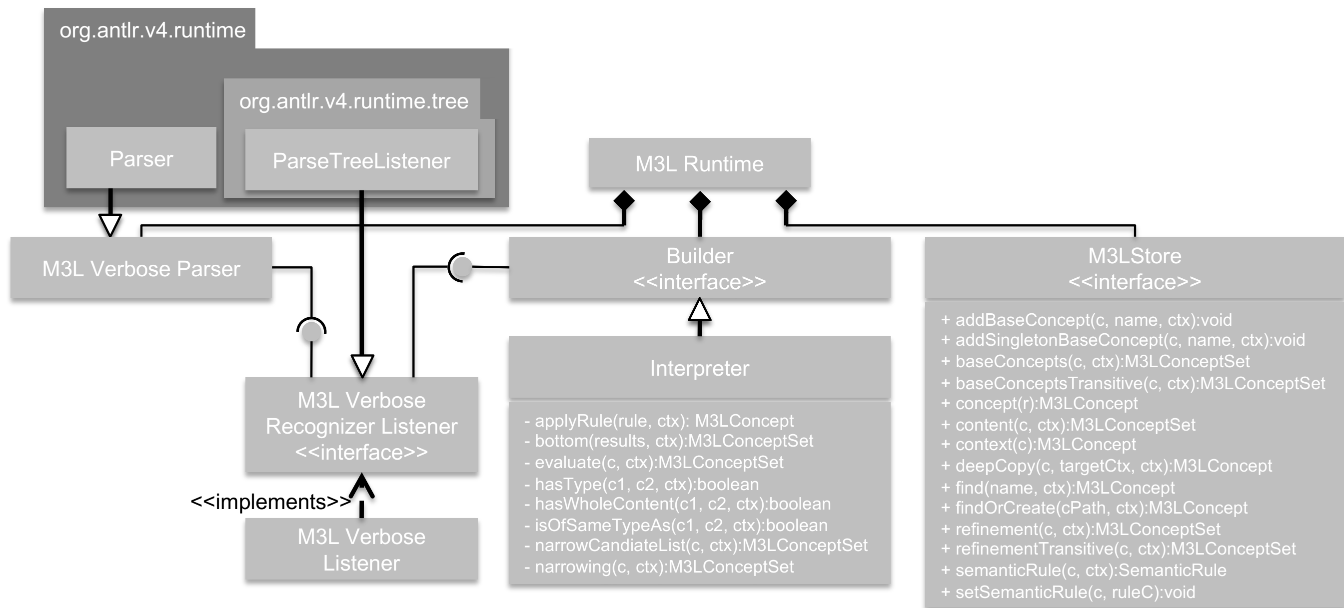

Figure 1. Architecture of the current M3L implementation.

When implementing concept evaluation (16) and all supporting functions (1)-(15), one notices that there are the basic accessor functions (1)-(6). Other functions are defined on top of these basic functions. Therefore, an efficient implementation will place the accessor functions (1)-(6) and those making heaviest use of them, (7) and (8), close to the data layer, while the others can be implemented in a storage-agnostic way. These assumptions lead to an architecture as the one presented in this section.

The current M3L runtime environment is an application that is based on several components. These components are interchangeable in order to be applicable in a wide range of configurations, namely different M3L syntaxes (compact or verbose), interpretation of files or interactive input, or compilation to different target languages and different persistence technology for concept storage and retrieval.

The UML class diagram in Fig. 1 illustrates this M3L implementation. For the method signatures shown in the diagram assume *M3LConcept* to be an interface for concept representations and *M3LConceptSet* to be a set of those.

For brevity, the types of method arguments are omitted. In the figure, *c* denotes a *M3LConcept* to perform an operation on, *ctx* a *M3LConcept* giving the context of the operation, *name* a String giving a concept name, and *r* a semantic rule.

At the frontend, a Parser recognizes M3L statements and creates an abstract syntax tree (AST) for further processing. The parser is based on a parser generated by the *ANTLR (ANother Tool for Language Recognition)* parser generator [12]. The grammar of the M3L is quite simple. Still, this powerful parser generator is employed because it plays an important role for the handling of syntactic rules (see Section III.F) at runtime and thus is part of the setup anyway.

In fact, there are different parsers and listeners for different syntaxes of the M3L we are experimenting with. Fig. 1 shows the *M3LVerboseParser* for the syntax used in this article.

In the next stage of M3L processing, a *Builder* creates an internal representation of the parsed M3L definitions.

Using the AntLR framework, a *Parser* and a *Builder* are connected by an observer, here the *M3LVerboseListener*, that receives callbacks whenever the parser recognized a syntactic construct.

In order to receive notifications, the observer implements methods defined by the AntLR API in the interface *ParseTreeListener*. The interfaces are not shown in detail but illustrated in UML by the "lollipop". In turn an observer uses an interface provided by *Builder* implementations (again represented by a lollipop) to pass information to them.

These interfaces allow different *Builder* implementations. Most notably, there are interpreters and compilers. The *Interpreter* acts directly. It contains generic code for the creation and evaluation of concepts. This code is based on operations provided by a *M3LStore* (see below). The inner working of the *Interpreter* is outlined by the private methods shown on the diagram in Fig. 1. The methods implement those functions from Section. IV that are expressed using the more basic functions.

A compiler creates equivalent code for the creation and evaluation of concepts that can (repeatedly) be executed.

Every concrete *Builder* implements the methods defined in the *Builder* interface that decorate the AST and pass the intermediate representation to a *M3LStore*. These methods are omitted in Fig. 1 in the shown *Interpreter*. Additionally, concrete builder implementations typically define methods for the functions (9)-(16) for concept evaluation. In Fig. 1 such methods are listed as private methods of *Interpreter*.

Analysis of these functions unveils the functionality to be provided by a *M3LStore*. According to this design, *M3LStore* implementations deliver the base functionality required for the builders, namely the required access functions as well as computed relationships that use them most (1)-(8).

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

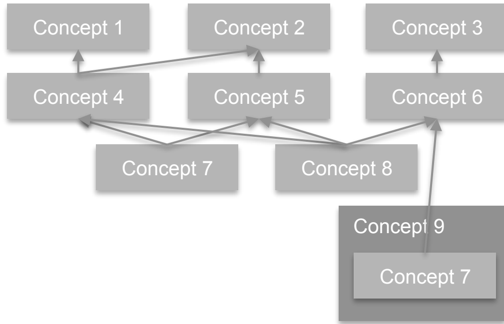

Figure 2. M3L concept refinements and contexts.

The *M3LStore* interface shown in Fig. 1 consists of methods used by a *Builder* to build up a model during parsing, and the abovementioned methods that implement the base functions used during concept evaluation (1)-(8).

For an efficient implementation, we lay an emphasis on the responsibilities of the *M3LStore*. The remainder of this article discusses mappings to some established persistence technologies that can be used as a basis of *M3LStore* implementations.

VI. A Conceptual Model for Content Representations

A conceptual model, as known from database modeling, serves as a first step towards data models for context-aware content. The notion of "concept" is ambiguous here: The aim is a model of (M3L) concepts. A conceptual model for this allows us to abstract from the M3L as a language. The model is not supposed to address practical properties such as operational complexity.

A set of M3L concept definitions can be viewed as a graph with each node representing a concept, labeled with the name of the concept. There are two kinds of edges to represent specialization and contextualization. In fact, such a graph forms a hypergraph to account for contextualization. Every node can contain a graph reflecting definitions as the concept's content.

The following subsections detail specialization and contextualization relationships, as well as contextual redefinitions.

A. Representing Specialization

Conceptually, a specialization/generalization relationship can straightforward be seen as a many-to-many relationship between concepts. Fig. 2 shows an example.

Arrows with filled heads, directed from a concept to its base concepts, represent specialization relationships in the figure. For example, *Concept 4* is a refinement of *Concept 1* and *Concept 2*.

Fig. 2 furthermore indicates an amendment in a context, namely *Concept 9*. While *Concept 7* is a refinement of *Concept 4* and *Concept 5* in the default context, it is additionally a refinement of *Concept 6* in the context of *Concept 9* (if it is an `is a`/`is an` definition; otherwise, *Concept 7* would only be a refinement of *Concept 6* in the context of *Concept 9*).

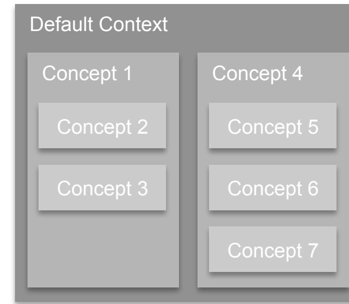

Figure 3. M3L concept definitions in contexts.

B. Representing Context

Since contexts form a hierarchy, contextualization can be represented by a one-to-many relationship between concepts in the roles of context and content.

Fig. 3 represents such a hierarchy by nested boxes shown for concepts. The contextualization relationship is thus visually represented by containment. For example, *Concept 2* is part of the content of *Concept 1*, or *Concept 2* is defined in the context of *Concept 1*.

The outermost context is the default context. There is no corresponding concept for this context.

C. Representing Contextual Information

Specialization and contextualization act together. Refinements of a concept inherit its content; concepts from that content are valid also in the context of the refinement. Each context allows concept amendments. These are a second way to add variations of concepts.

In order to represent contextualized redefinitions, we introduce two kinds of context definitions: *Initial Concept Definition* and *Contextual Concept Amendment*. Both can be placed in any context.

An initial concept definition is placed in the topmost context in which a concept is defined. Redefinitions of concepts are represented by contextual amendments inside the concept in whose context the redefinition is performed.

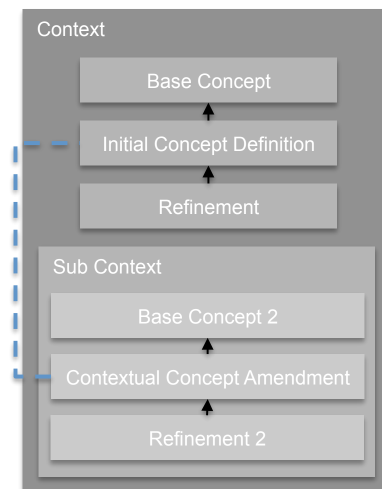

Figure 4. M3L concept amendments in contexts.

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

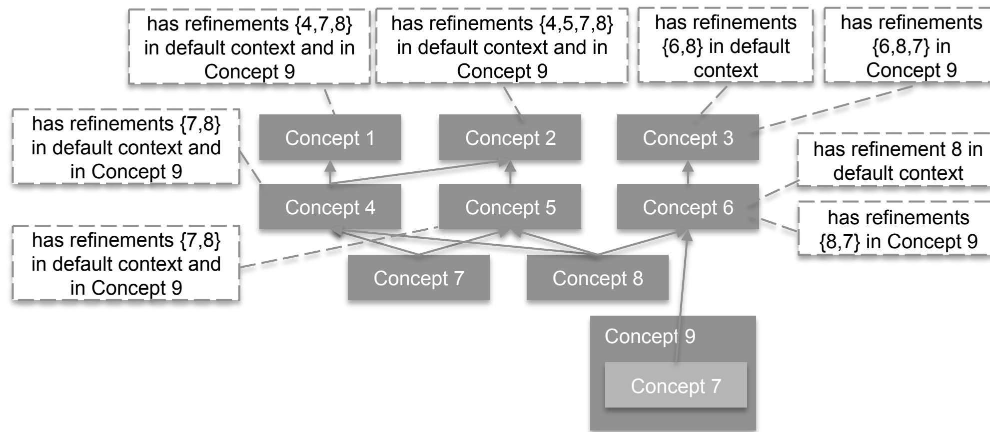

Figure 5. Representation of refinements using materialized transitive refinement relationships.

Fig. 4 illustrates such a concept redefinition scenario. As before, contexts are depicted as nested boxes. There is one *Context* and a *Sub-Context*. Both show a *Concept* that has originally been defined as a refinement of *Base Concept* and is itself refined to *Refinement*. In the context on *Sub-Context*, the concept gets the additional base concept *Base Concept 2*, and there is another refinement *Refinement 2*. These additions are recorded in the contextual amendment of *Concept* in *Sub-Context*. This is, of course, transparent on model level.

Amendments have a reference to the next higher definition. This reference is called *Original*. In Fig. 4, it is shown by the dotted line.

Traversal of the original references allows collecting all definitions in order to determine the effective definition.

VII. LOGICAL CONTENT REPRESENTATION

This section refines contextual content representation models to a level similar to that of a logical data model. This way it discusses properties of data representations without taking implementation details into account.

The complexity of lookups is of major importance for the schema design. During the evaluation of M3L statements, many graph traversals are required to find all valid contexts, all base concepts (to determine content sets) and all refinements (to narrow down concepts before applying rules; this evaluation process is not laid out in this paper).

The most important design decision is the degree of (de)normalization of the schema. The basic assumption is that content is mainly queried, so that creation and update cost is less important than lookup cost.

We consider two designs of denormalized schemas: materialization of reference sets and storage of relationships in a way that allows efficient queries. Efficient storage is based on the usage of numeric IDs to reference concepts and computing relationships based on ID sets. An example of such an approach is the BIRD numbering scheme for trees [13] that allows range queries to determine subtrees.

A. Storing Refinements

Compared to the straightforward conceptual model, the logical schema is denormalized in order to avoid repeated navigation of specialization relationships when collecting the set of (transitive) base concepts or refinements of a concept.

Two approaches are investigated: aggregated data and transitive refinement relationships.

Aggregated data collects necessary information to avoid nested queries for refinements. All base concepts and all refinements are stored in an object representing the concept definition in a certain context. Context-dependent content is added in contextual concept amendments (s.a.) that are stored as part of the context hierarchy. These aggregate the definitions effective in all parent contexts.

The description objects additionally reference each other via original references.

Alternatively, just transitive refinement relationships are materialized for every concept in every context. This way, transitive refinements are directly available, and base concepts can be collected using a simple query.

Fig. 5 shows an example for the sample from Fig. 2. The dashed boxes show the transitive refinements per relevant context. Base concepts can be determined by queries.

For example, the (transitive) base concepts of *Concept 4* are those concepts that have this concept as a refinement. Specifically, these are *Concept 1* and *Concept 2* (in both the default context and in the context of *Concept 9*).

Storing the context together with the refinement relationships is vital for handling singleton (`is the`) relationships, in particular the unbinding of concepts.

B. Storing Context Hierarchies

Performance is particularly important for the retrieval of the hierarchy of contexts a concept is defined or amended in. The effective definition of a concept (including aggregated base concepts and content) relies on this concept hierarchy.

By blending in the context information into the transitive refinements, as shown in the previous subsection, the situation is leveraged to a large degree. Still, the content that a concept has in a certain context is also relevant to concept evaluations.

As for the specialization/generalization relationships, two approaches are discussed here: materialized content collections in all contexts and information about paths in the context hierarchy.

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

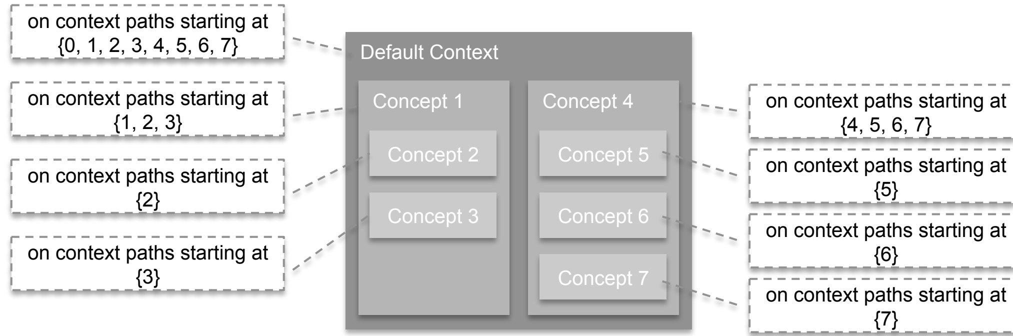

Figure 6. Representation of context hierarchies my materializing paths.

The materialization of contextual definitions works the same way as that of refinements: with every concept definition amendment, we store the effective content in the respective context. This has to be computed on definition.

For the second approach, Fig. 6 illustrates the attribution of paths to the schematic example of Fig. 3. For each concept, we note down the concepts lying on the path in the context hierarchy from the default context to a specific context. For example, *Concept 1* lies on the paths from the default context to itself, to *Concept 2*, and to *Concept 3*.

We used numeric IDs to reference the concept (with the ID 0 given to the pseudo-concept for the default concept). IDs have to be ordered from the default context to sub-contexts. By querying for all concepts on the path of a concept, ordered by ID, we retrieve the path to that concept.

VIII. PHYSICAL CONTENT STORAGE MODELS

This section briefly discusses some implementation approaches to context-aware content models using different data models. Specifically, we present the basics of a mapping to relational databases, to a document-oriented database, a content repository, and a graph database.

A. Mapping M3L to a Relational Database

There is a range of approaches to storing trees and graphs in relational databases [14]. On the basis of these, we add materialized transitive relationships as described above.

Relational tables for the transitive context hierarchy can be defined by statements like (with numeric type INT):

```
CREATE TABLE concept (id INT PRIMARY KEY);
CREATE TABLE paths (
 concept_id       INT REFERENCES concept(id),
 terminal_concept INT REFERENCES concept(id),
 PRIMARY KEY (concept_id, terminal_concept));
```

The table *concept* holds concepts (both initial definitions and amendments) with artificial IDs (other data is omitted here). The second table holds the path information as indicated in Fig. 6. *concept_id* refers to the concept, *terminal_concept* refers to the concept on whose path the concept lies.

Data stored this way can be queried by, e.g.,

```
SELECT c.* FROM concept c, paths p
 WHERE c.id = p.concept_id
 AND p.terminal_concept = i
 ORDER BY p.concept_id DESC;
```

to retrieve the path to concept *i*.

Transitive refinements can be stored in a table:

```
CREATE TABLE transitive_refinements (
 base_concept_id INT REFERENCES concept(id),
 refinement_id   INT REFERENCES concept(id),
 context_id      INT REFERENCES concept(id),
 PRIMARY KEY (base_concept_id, refinement_id,
             context_id));
```

The base concepts of, e.g., *Concept 4* can be queried by:

```
SELECT base_concept_id
 FROM transitive_refinements
 WHERE refinement_id = 4 AND context_id = 0;
```

in the default context (with ID 0), or by:

```
SELECT base_concept_id
 FROM transitive_refinements
 WHERE refinement_id = 4 AND context_id = 9;
```

for the context of *Concept 9*.

B. MongoDB

As an example of so-called NoSQL approaches, we conduct ongoing experiments with MongoDB [15], a widely used document-oriented database management system.

The definition of concept relationships is done in a similar way as in relational databases: records have IDs, and records store IDs for references. There are no distinct relation structures, though. References are stored as document fields.

In contrast to a purely relational structure, documents allow representing nested contexts in a natural manner by embedded documents.

As an example of a schema, the *insert* statement shown in Fig. 7 stores the whole graph of Fig. 2.

This structure can be queried as required. For example, to find concepts with base concept *Concept 6* in the context of *Concept 9*, the *aggregate* statement in Fig. 7 can be applied.

C. Content Repository for Java Technology API (JCR)

In an attempt to define a content-specific database, the *Content Repository for Java Technology API* (*JCR*) standard has been set up in *Java Specification Requests* JSR-170 [16] and JSR-283 [17]. The standard is employed by some commercial content management system products.

The API implies a content model to be supported by JCR implementations. The data model behind JCR is similar to XML: It features hierarchies of *nodes*, where each node can have *properties*, attributes of one out of a set of predefined base types.

```
db.concept.insert({ name: "Default Context", content: [
 { name: "Concept 1", baseConcepts: null,                                content: null },
 { name: "Concept 2", baseConcepts: null,                                content: null },
 { name: "Concept 3", baseConcepts: null,                                content: null },
 { name: "Concept 4", baseConcepts: ["Concept 1", "Concept 2"],          content: null },
 { name: "Concept 5", baseConcepts: ["Concept 2"],                       content: null },
 { name: "Concept 6", baseConcepts: ["Concept 3"],                       content: null },
 { name: "Concept 7", baseConcepts: ["Concept 4", "Concept 5"],          content: null },
 { name: "Concept 8", baseConcepts: ["Concept 4", "Concept 5", "Concept 6"], content: null },
 { name: "Concept 9", baseConcepts: null,                                content: [
  {name: "Concept 7", baseConcepts: ["Concept 4", "Concept 5", "Concept 6"], content: null,
   original: "Concept 7" } ] } ] })
db.concept.aggregate([
{$unwind:"$content"},{$replaceRoot:{newRoot:"$content"}},{$match:{name:"Concept 9"}},
{$unwind:"$content"},{$replaceRoot:{newRoot:"$content"}},{$match:{baseConcepts:"Concept 6"}}])
```

Figure 7. Document definitions to map M3L to MongoDB and a sample query.

With these characteristics, M3L concept models can be mapped to JCR in a straightforward manner: nodes represent concepts, and concept relationships are expressed in the node hierarchy as well as by properties of type *reference*.

Context hierarchy in M3L is reflected by the node hierarchy in JCR. This way, the API allows direct access to context by `Node#getParent()` and access to content by `Node#getNodes()`.

Relationships to base concepts are represented by a (multi-valued) reference property *base-ref*.

Contextual concept amendments are represented as nodes on their own as outlined in Section VI.C. Nodes for amendments link to the node representing the definition they add to by a reference property *original*.

A semantic rule is represented by a reference from a concept to the one that is defined by its rule. To this end, rule concepts that are instantiated on rule application are stored outside of concept hierarchy.

With this mapping, a *M3LStore* can be expressed easily using the JCR API. Functions regarding the context hierarchy are directly reflected in the node hierarchy, base concept references are expressed by reference properties of nodes.

Only refinement relationships require special consideration for navigating base concept references against their direction. E.g., for transitive refinements, Java code like the following is included (with `c` a node representing the concept for which to compute the transitive refinement, `ctx` a node representing the context in which to evaluate it, `refinement` a set in which to collect nodes):

```
outer: for (Node c2 : allConcepts()) {
  Node[] baseConcepts
  = baseConceptsTransitive(session, c2, ctx);
  for (Node bC : baseConcepts)
    if (bC.getPath().equals(c.getPath())) {
      refinement.add(c2);
      continue outer;
    }
}
```

This code is the core of the *refinementTransitive()* method of a *M3LStore* for JCR.

D. Mapping M3L to a Graph Database

Graph database management systems [18] organize data as graphs of different types.

In this section, the DMBS *Neo4J* [19] is considered as a representative of graph database management systems. It allows data modelling using directed colored graphs with labelled nodes. Data manipulation and querying is performed using the language *Cypher* [20].

In Neo4j, we model M3L concepts as nodes. Following the conceptual model from Section VI, we introduce types (labels) *CONCEPT* and *CONCEPTAMENDMENT* for initial concept definitions and for conceptual content amendments. For each contextual definition, an explicit node with a label is created. Edges representing concept relationships are set to and from nodes representing concepts in specific contexts.

For the different concept relationships occurring in M3L models, we add edges of different types. To express context, we use an edge of kind *CONTEXT* from a node representing a concept to a node representing the context of that concept. The relationship between a refinement and its base concept is represented by an edge of kind *BASE*. We record a reference from a contextual concept amendment to the concept it is redefining using an *ORIGINAL* edge. The semantic rule of a concept is expressed by a *SEMANTICRULE* edge from the concept to the new concept the rule defines.

Fig. 8 shows a database resulting from the concept definitions in the example of Person entities from Section III. It is a screen shot taken from the tool *Neo4j Browser*.

The node color shows the label assigned to a node: green for initial concept definition, blue for conceptual amendment.

Cypher allows expressing transitivity directly, e.g., using the path `()-[:BASE*]-()` for (7) ($base^T$). Therefore, the basic concept definitions and access functions can be mapped to Cypher in a straightforward way.

Root level concepts are defined by a simple CREATE directive:

```
CREATE (c:CONCEPT) SET c.name='concept name'
```

In the mapping examples in this section, italicized terms are placeholders for parameter values. In the create directive this is the name of the concept to be created.

Nested concepts are defined in a given context by:

```
MATCH (ctx{name: 'context's concept name'})
      -[:CONTEXT]->...(t)
WHERE NOT (t)-[:CONTEXT]->()
CREATE (c:CONCEPT) SET c.name='concept name'
CREATE (c)-[:CONTEXT]->(ctx)
```

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

Figure 8. Representation of M3L definitions in a graph database.

The MATCH selects the node representing the concept in whose content to create the new concept. In order to select the right node, the complete path to that node is given. The path has to be generated for the context concept at hand, exemplified by the ellipsis. The WHERE clause constrains the match to paths whose end node does not have any further outgoing edges of kind CONTEXT (which are the root nodes).

A concept refinement is defined by:

```
MATCH (c{name: 'concept name'})
      -[:CONTEXT]->...(t)
WHERE NOT (t)-[:CONTEXT]->()
MATCH (b{name: 'base concept name'})
      -[:CONTEXT]->...(t)
WHERE NOT (t)-[:CONTEXT]->()
CREATE (c)-[:BASE]->(b)
```

Here, *c* matches the refinement and *b* matches the base concept using generated paths.

Concept amendments are defined by creating a node and relating it to the redefined node with an ORIGINAL edge as described in Section III.C. This is done by:

```
MATCH (ctx{name: 'context's concept name'})
      -[:CONTEXT]->...(t)
WHERE NOT (t)-[:CONTEXT]->()
MATCH (o{name: 'concept name'})
      -[:CONTEXT]->()
      <-[:CONTEXT|:BASE*]-(ctx)
CREATE (c:CONCEPTAMENDMENT)
SET c.name='concept name'
CREATE (c)-[:CONTEXT]->(ctx)
CREATE (c)-[:ORIGINAL]->(o)
```

The node *ctx* representing the context is found by matching a path as in the above statements. The concept definition *o* to be amended is found by the second MATCH directive. It looks for a node with the right name in the context of a node that is reachable from the context via context (following CONTEXT edges) or via refinement (following BASE edges) relationships, meaning that *o* it is either visible in an outer context or is inherited.

A new node is created with the same name as the original node *o*, labelled with type CONCEPTAMENDMENT, put in the context *ctx*, and related to the original.

Semantic rules are set using:

```
MATCH (c{name: 'concept name'})
      -[:CONTEXT]->...(t)
WHERE NOT (t)-[:CONTEXT]->()
CREATE (r:CONCEPT)
SET r.name='rule concept name'
CREATE (c)-[:SEMANTICRULE]->(r)
```

The concept *c* to which the rule is assigned is matched by name and context path again. The concept *r* that is instantiated on rule application is created like any concept and then related using SEMANTICRULE.

Interpreting the concept definitions from Sections III.B and III.E on a *M3LStore* with this kind of mapping from M3L to Cypher leads to a graph database as shown in Fig. 8.

IX. CONCLUSION

This section sums up the paper and gives an outlook on future work towards M3L concept persistence and querying.

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org

A. Summary

This article lays out approaches to context-aware content management, in particular using the Minimalistic Meta Modeling Language (M3L). Semantics is given to content by rules that allow M3L concept evaluation.

The architecture of a current testbed implementation is presented. The architecture description concentrates on basic functions required for M3L concept evaluation in a data layer. Since content bases typically become large in data volume, persistency has to be provided by this data layer.

Though it is easily possible to map context representations to existing data management approaches, care has to be taken to enable efficient querying for M3L concept evaluation.

A logical schema for the representation of contextual content is presented that introduces first optimizations that are independent of the target data model and the database management systems used.

First sketches of implementations using different data models are conducted. These demonstrate the feasibility of concept persistence using these data models.

Representative technologies for each data model are used to present schemas that can serve as a starting point of the discussion and evaluation of M3L implementations.

B. Outlook

The work on data model mappings for M3L concept definitions is ongoing work. There is ample room for further optimizations of the relational database schema with respect to query execution. The mappings to other data models, document-oriented, tree, and graph databases, need elaboration before significant comparisons between these can be conducted.

The utilization of databases to support M3L concept evaluation is an open issue. Currently, base functions are implemented by database queries while the overall evaluation process is performed in a generic way by application code. Other functions required for concept evaluation may be implemented efficiently in certain database models. One example is the computation of candidate lists for narrowings (10) that may be formulated using database-specific queries.

Experiments with different implementations are ongoing. Data models have yet to be rated based on practical results. To this end, implementations need to be optimized.

For comparison, a kind of test suite needs to be defined. Models and rule sets that address realistic scenarios will guide the investigations in the future. Data of significant volume has to be generated as concept instances according to such models.

ACKNOWLEDGMENT

Though the ideas presented in this paper are in no way related to the work at Namics, the author is thankful to his employer for letting him follow his research ambitions based on experience made in customer projects.

Discussions with colleagues, partners, and customers are highly appreciated.

Thanks go to the reviewers of the original conference paper as well as to those of this journal article.

REFERENCES

[1] H.-W. Sehring, “Schemas for Context-aware Content Storage,” Proc. Tenth Int. Conference on Creative Content Technologies (CONTENT 2018), pp. 18-23, Sep. 2018.

[2] M. Gutmann, “Information Technology and Society,” Swiss Federal Institute of Technology Zurich / Ecole Centrale de Paris, 2001.

[3] C. Bolchini, C. A. Curino, E. Quintarelli, F. A. Schreiber, and L. Tanca, “A Data-oriented Survey of Context Models,” ACM SIGMOD Record, vol. 36, pp. 19-26, December 2007.

[4] A. Zimmermann, M. Specht, and A. Lorenz, “Personalization and Context Management,” User Modeling and User-Adapted Interaction, vol. 15, pp. 275-302, Aug. 2005.

[5] S. Trullemans, L. Van Holsbeeke, and B. Signer, “The Context Modelling Toolkit: A Unified Multi-layered Context Modelling Approach,” Proc. ACM Human-Computer Interaction (PACMHCI), vol. 1, June 2017, pp. 7:1-7:16.

[6] G. Orsi and L. Tanca, “Context Modelling and Context-Aware Querying (Can Datalog Be of Help?),” Proc. First International Conference on Datalog Reloaded (Datalog '10), Mar. 2010, pp. 225-244.

[7] D. Ayed, C. Taconet, and G. Bernard, “A Data Model for Context-aware Deployment of Component-based Applications onto Distributed Systems,” GET/INT, 2004.

[8] S. Vaupel, D. Wlochowitz, and G. Taentzer, “A Generic Architecture Supporting Context-Aware Data and Transaction Management for Mobile Applications”, Proc. International Conference on Mobile Software Engineering and Systems (MOBILESoft '16), May 2016, pp. 111-122.

[9] J. W. Schmidt and H.-W. Sehring, “Conceptual Content Modeling and Management,” Perspectives of System Informatics, vol. 2890, M. Broy and A.V. Zamulin, Eds. Springer-Verlag, pp. 469-493, 2003.

[10] H.-W. Sehring, “Content Modeling Based on Concepts in Contexts,” Proc. Third Int. Conference on Creative Content Technologies (CONTENT 2011), pp. 18-23, Sep. 2011.

[11] C. Diggins: *Explicit Structural Typing (Duck Typing)*. [Online]. Available from http://www.drdobbs.com/architecture-and-design/explicit-structural-typing-duck-typing/228701413

[12] T. Parr, The Definitive ANTLR 4 Reference. Pragmatic Bookshelf, 2013.

[13] F. Weigel, K. U. Schulz, and H. Meuss, “The BIRD Numbering Scheme for XML and Tree Databases – Deciding and Reconstructing Tree Relations using Efficient Arithmetic Operations,” Proc. Third international conference on Database and XML Technologies (XSym'05), Aug. 2005, pp. 49-67.

[14] V. Tropashko, SQL Design Patterns: The Expert Guide to SQL Programming. Rampant Techpress, 2006.

[15] K. Chodorow and M. Dirolf, MongoDB: The Definitive Guide. O’Reilly Media, Inc., 2010.

[16] Day Software AG: *Content Repository for Java Technology API Specification (JSR-170)*. [Online]. Available from http://docs.adobe.com/content/docs/en/spec/jcr/1.0/index.html

[17] Oracle Corporation: *JSR 283: Content Repository for JavaTM Technology API Version 2.0*. [Online]. Available from https://jcp.org/en/jsr/detail?id=283

[18] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, “Foundations of Modern Query Languages for Graph Databases,” in ACM Computing Surveys vol. 50, September 2017.

[19] J. Baton, R. Van Bruggen, Learning Neo4j 3.x. Packt, 2017.

[20] I. Robinson, J. Webber, and E. Eifrem, Graph Databases: New Opportunities for Connected Data. O'Reilly Media, 2015.

2018, © Copyright by authors, Published under agreement with IARIA - www.iaria.org