



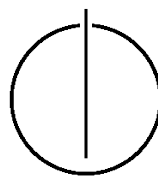
FAKULTÄT FÜR INFORMATIK

DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

**Design and prototypical implementation of a
model-based structure for the definition and
calculation of Enterprise Architecture Key
Performance Indicators**

Thomas Reschenhofer





FAKULTÄT FÜR INFORMATIK

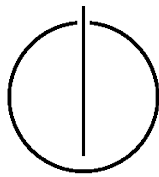
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatik

Design and prototypical implementation of a
model-based structure for the definition and
calculation of Enterprise Architecture Key
Performance Indicators

Konzeption und prototypische Implementierung
einer Modell-basierten Struktur zur
Definition und Berechnung von
Unternehmensarchitekturkennzahlen

Author: Thomas Reschenhofer
Supervisor: Prof. Dr. Florian Matthes
Advisor: Dipl. Inform.-Univ. Ivan Monahov
Date: September 10, 2013



I assure the single handed composition of this master's thesis only supported by declared resources

München, September 10, 2013

Thomas Reschenhofer

Abstract

Due to the increasing complexity in Enterprise Architectures (EA), organization-specific Key Performance Indicators (KPI) are an important tool for the measurement of certain EA characteristics, and hence support the understanding of the EA's structure and its dynamics. Therefore, our group developed a method for defining EAM KPIs based on a uniform and configurable structure as well as a catalog consisting of concrete KPIs gathered from literature and industry. Furthermore, in previous research, we implemented the prototype of a model-based expression language (MxL), which empowers business users to formally define KPIs allowing tool-supported evaluation of EAM KPIs.

However, the design method for defining EAM KPIs is still lacking a proper implementation, which supports the enterprise architect in selecting, instantiating, and configuring KPIs based on the structure and catalog developed by our group. Moreover, the implementation has to support the adaption of the KPIs to a given organization-specific context. Therefore, this thesis examines the shortcomings of the initial language design of MxL and its implementation regarding their suitability for the implementation of the EAM KPI definition design method. Based on these shortcomings, the thesis focuses on the reengineering of MxL and its implementation on the one hand, and the extension of the prototype to allow the implementation of both the EAM KPI structure and the EAM KPI Catalog on the other hand.

The result of the thesis is a prototype supporting all steps of the design method for defining EAM KPIs, i.e. enterprise architects can choose from existing KPIs from the catalog and adapt them to organization-specific needs without violating the environments consistency. Hence, the prototype is a first step towards an integrated EAM environment supporting enterprise architects to define, document, evaluate, and interpret EAM KPIs.

Contents

Abstract	vii
I. Introduction	1
1. Introduction & Motivation	3
1.1. Enterprise Architecture Management	3
1.2. Key Performance Indicators	4
1.3. Motivation & Approach of the thesis	5
2. Foundations	7
2.1. EAM KPI structure	7
2.1.1. General structure elements	7
2.1.2. Organization-specific structure elements	9
2.1.3. Design method for defining EAM KPIs	11
2.2. Tricia	12
2.2.1. Architecture	12
2.2.2. Hybrid wikis	14
2.3. MxL 1.0	15
2.3.1. Fundamentals of MxL 1.0	16
2.3.2. Use cases of MxL 1.0	18
2.4. Shortcomings of existing foundations	19
2.4.1. No compile-time analysis of MxL expressions	19
2.4.2. Tight coupling between MxL 1.0 and Tricia	20
2.4.3. Missing type-based template engine in Tricia	20
2.4.4. Insufficient deployment support in Tricia	20
II. Contribution of the Thesis	23
3. MxL 2.0	25
3.1. Fundamentals of MxL 2.0	25
3.1.1. MxL 2.0 types	25
3.1.2. Basic language constructs	25
3.1.3. Higher-order functions in MxL 2.0	29
3.1.4. Sequence functions	30
3.1.5. MxL Asset Hierarchy	35
3.1.6. Querying the information model	37
3.2. Interpretation	39

3.2.1.	Scanner & Parser	39
3.2.2.	AST & Expression objects	40
3.2.3.	MxL Connector	42
3.2.4.	Type Checker	42
3.2.5.	Evaluation engine	49
3.3.	TxL 2.0 - MxL 2.0's implementation in Tricia	50
3.3.1.	MxL Connector for Tricia	50
3.3.2.	Basic MxL infrastructure in Tricia	51
3.3.3.	Derived attributes and custom functions	52
3.3.4.	Embedded expressions	54
3.3.5.	Compile-time analysis of MxL expressions in Tricia	56
4.	Towards Living KPIs	59
4.1.	Implementation of a type-based template engine	59
4.1.1.	Existing template engine	59
4.1.2.	Type-based template engine	59
4.1.3.	Example of a page template	62
4.2.	Deployment of Tricia applications	62
4.2.1.	Existing import types	64
4.2.2.	Initial data definition	65
4.2.3.	Initial data processing	68
4.2.4.	Back tracking of changes on initial data sets	70
4.3.	Prototype of the Living KPIs	70
4.3.1.	Implementation of KPIs as MxL 2.0 custom functions	71
4.3.2.	Page template for EAM KPI descriptions	72
4.3.3.	EAM KPI Catalog data	72
III.	Results	77
5.	Summary & Conclusion	79
5.1.	Summary	79
5.2.	Conclusion	80
6.	Outlook & Future research	83
6.1.	Authorization in MxL 2.0	83
6.1.1.	Problem / Open issue	83
6.1.2.	Proposed solution	83
6.2.	Evaluating identity	84
6.2.1.	Problem / Open issue	84
6.2.2.	Proposed solution	85
6.3.	Evaluation strategy	85
6.3.1.	Problem / Open issue	85
6.3.2.	Proposed solution	86
6.4.	History of evaluation results	87
6.4.1.	Problem / Open issue	87

6.4.2. Proposed solution	87
6.5. Visualization of evaluation and type checking results	88
6.5.1. Problem / Open issue	89
6.5.2. Proposed solution	89
6.6. Query processing	90
6.6.1. Problem / Open issue	90
6.6.2. Proposed solution	91
6.7. Fully supported Living KPIs	91
6.7.1. Problem / Open issue	92
6.7.2. Proposed solution	92
Bibliography	95

Part I.

Introduction

1. Introduction & Motivation

This chapter motivates the thesis by introducing the area of Enterprise Architecture Management (Section 1.1) explaining the relevance of Key Performance Indicators in this area, and emphasizing the need for an integrated environment for the management and calculation of these Key Performance Indicators and their information model (Section 1.2). Subsequently, Section 1.3 highlights the gap between existing theoretical foundations and proper tool support, and states how this thesis wants to close this gap.

1.1. Enterprise Architecture Management

An Enterprise Architecture (EA) is the holistic structure of an enterprise as a socio-technical system embodied in its components and their relationships [1], covering all areas of an enterprise from business to information technology (IT) aspects. Holism is a system-theoretical property meaning that a system is more than the sum of its parts, i.e. the system and its behavior cannot be deduced from the properties of its elements alone. Applied onto an EA holism means, that it is not sufficient to observe just the single elements of the EA to understand the behavior of the whole system.

Since the enterprise's environment changes continuously (e.g., variable customer demands, technology innovation, and changing legal conditions), it's necessary to adapt the EA to these changes[2]. However, the ever-growing complexity of an EA as well as its holism makes it difficult to understand the EA and its dynamics on the one hand, and hence to adapt the EA to a new environment on the other hand. This yields to several downsides, e.g., loss of transparency, increased complexity costs and risks, or distraction from core business problems.

Enterprise Architecture Management (EAM) is a holistic way to plan, develop and control an EA's evolution to ensure its flexibility, efficiency and transparency[2]. Therefore, EAM supports the enterprise architect in his understanding of the EA and its behavior and – in the end – has a positive impact onto the business performance, as shown in Figure 1.1.

Because of the growth of complexity of EAs, tools supporting EAM are becoming more and more important [3]. These EAM tools [4] provide methods for gathering the EA model's data, modeling techniques for the EA and guidelines for its visualization [5, 6, 7].

However, although the information sources in EAM are very unstructured (e.g., spreadsheets, slides, documents) and shared among a multitude of stakeholders [8], the information structures and collaboration mechanisms provided by prevalent EAM tools are rather rigid, which yields to a major problem in EAM [9]. Therefore a wiki-based approach to EAM called Wiki4EAM was developed at the chair for Software Engineering for Business Information Systems (sebis)¹ at the Technical University Munich (TUM) in 2010, which allows an incremental and collaborative enrichment of initially unstructured information

¹<http://www.matthes.in.tum.de>

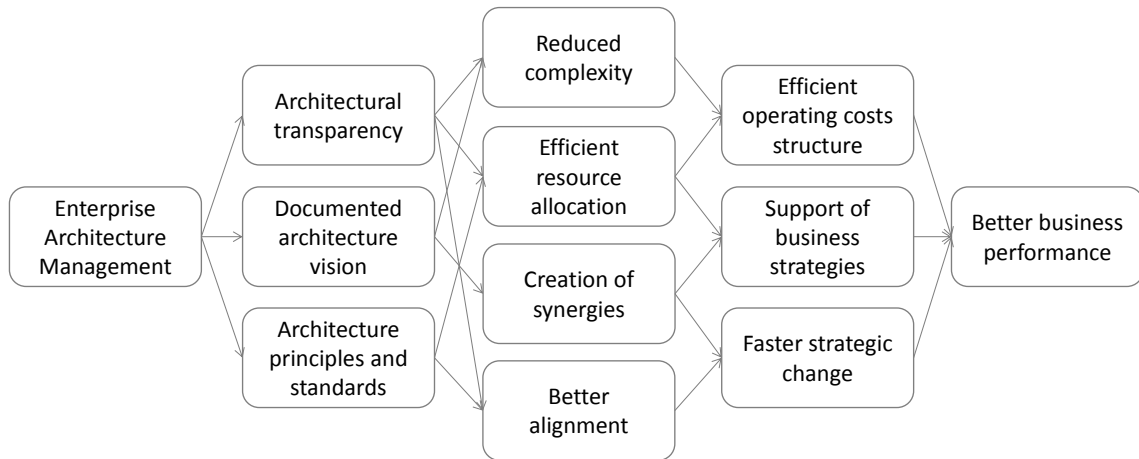


Figure 1.1.: The effects of Enterprise Architecture Management according to Ahlemann et al. [2]

sources with attributes, types and integrity rules. The model-based enterprise wiki system Tricia², which is used for this approach, as well as the so-called Hybrid Wikis are subject in Section 2.2.

1.2. Key Performance Indicators

As already stated, controlling is an important function of EAM. However, in order to control the EA, it's necessary to monitor, measure, and evaluate certain performance-related EA characteristics [10]. Such metrics are called Key Performance Indicators (KPIs) and are able to qualitatively assess the EA itself as well as the achievement of predefined EAM goals.

However, although KPIs are essential for the validation of goal achievements and the development of meaningful value propositions [11], there exists no common structure tailored to the definition and documentation of EAM KPIs [12]. Therefore, our group developed an uniform and configurable EAM KPI structure consisting of general structure elements as well as organization-specific structure elements. Furthermore, our group published a EAM KPI Catalog [13] (in the remainder of the thesis I refer to this source as "catalog"), which defines 52 KPIs gathered from literature and industry partners based on the EAM KPI structure. This structure as well as the catalog enable enterprise architects, IT managers, and business domain experts to quantify and measure their EAM goal achievements and compare them to each other. Moreover, Matthes et al. [14] introduced a design method for the definition of EAM KPIs, guiding all the stakeholders involved in the definition of EAM KPIs through a process consisting of four design steps and based on both the EAM KPI structure and catalog. In Section 2.1, the EAM KPI structure as well as the design method for defining EAM KPIs will be described in more detail.

Although each KPI in the catalog consists of a very detailed and structured description, a

²<http://infoAsset.de>

formal specification of how to calculate a KPI was still missing until recently. For this purpose, a domain-specific language (DSL) named *Model-based Expression Language* (MxL) was developed by Monahov et al. [15]. By the use of this DSL, all the catalog's KPIs are definable in a formal way by stating the calculation prescription as MxL expression, which furthermore enables an automated and tool-supported evaluation of these KPIs. Section 2.3 describes the language MxL itself as well as its prototypical implementation and integration into the model-based wiki system Tricia.

1.3. Motivation & Approach of the thesis

On the one hand, although the EAM KPI Catalog provides a structured set of 52 EAM KPIs gathered from industry and literature, it still lacks a proper implementation and tool integration.

On the other hand, the model-based wiki system Tricia already provides the modeling capabilities required for the implementation of the catalog's integrated information model as well as a DSL (MxL) to perform queries and arithmetic calculations onto this model's data. However, MxL in its current version does not provide mechanisms to analyze MxL expressions to determine the information model elements required to successfully evaluate the KPI represented by an expression. Furthermore, Tricia does not provide an engine to define a flexible and easily adaptable template for the EAM KPI structure.

To close this gap, this thesis covers the design and prototypical implementation of a model-based EAM environment, which is

integrated, i.e., there is one common platform for the collaborative gathering of the EA's data, its management, and the definition, management, and automated evaluation of proper KPIs

based on the EAM KPI structure and the catalog, i.e., all the catalog's KPIs and their descriptions as well as the catalog's integrated data model are initially implemented in the platform. Furthermore, the layout and design of the KPI descriptions in the platform should be deduced from the EAM KPI structure as developed by Matthes et al. *Ma12e*, i.e. the platform is an approach towards "Living KPIs" and hence immediately familiar to users who already know the EAM KPI structure or the catalog.

flexible/adaptable in the sense that the EAM environment and its information model are easily adaptable to a organization-specific context (e.g., by removing KPIs and/or model elements not needed by the enterprise) while retaining the integrity of the information model and the formal specifications of each KPI's computation prescription

By fulfilling these three requirements, this environment will be able to support the entire design method for defining EAM KPIs as described in Subsection 2.1.3.

In the remainder of the thesis this environment will be called the "Living KPIs". While the next chapter describes the foundations on which the Living KPIs will be based on, as well as their shortcomings, the whole Part II will cover the main contribution of the thesis, which is the implementation of a prototype of the Living KPIs.

2. Foundations

As stated in Chapter 1, the goal of the thesis is the implementation of a prototype for the Living KPIs - an integrated and flexible EAM platform based on the EAM KPI structure. The foundations of this platform are the EAM KPI structure (c.f. Section 2.1), the model-based wiki system Tricia (c.f. Section 2.2), and the domain-specific language MxL (c.f. Section 2.3).

However, Section 2.4 will come to the conclusion, that these three components are still lacking some essential features in order to reach the goal of the thesis.

2.1. EAM KPI structure

As stated in *section:kpi*, the EAM KPI structure [12] enables an uniform description, definition, and documentation of EAM KPIs, whereas the EAM KPI Catalog [13] defines 52 KPIs based on this structure. Figure 2.1 shows an exemplary instance of one of the catalog's KPIs.

The catalog's KPIs are organized by an uniform KPI description template providing general structure elements (GSEs) as well as organization-specific structure elements (OSSEs) [12] for the structured description of each KPI. This structure ensures consistency among all of the catalog's KPIs and simplifies and unifies the adaption of a KPI to an organization-specific context.

2.1.1. General structure elements

The following GSEs are independent from the enterprise, in which the KPIs should be applied:

Title A unique name and at the same time a very short description of the purpose of the KPI

Description A more detailed description of the KPI and its purpose

Goals Each KPI is related to at least one of ten distinct EAM goals, which are based on the findings of Buckl et al. [1].

Calculation The calculation of a KPI provides a textual and informal prescription, how the KPI has to be calculated based on a certain information model

Source The origin of the KPI may be either literature or practice (industry partner)

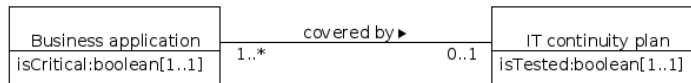
Layers Each of the catalog's KPIs can be assigned to one or more EA layers and cross functions, which are depicted in Figure 2.2 and based on Wittenburg [16]. This assignment may be helpful to decide whether or not a KPI can be employed at all.

Application continuity plan availability

Description

A measure of how completely IT continuity plans for business critical applications have been drawn and tested up for the IT's application portfolio.

Information model



Organization-specific instantiation

Mapping:

Name in model	Mapped name	Contacts	Data sources
Business application	Application	J. Doe	EA repository
isCritical	criticality	J. Doe	EA repository
covered by	covers	R. Miles	Risk Mgmt. rep.
IT continuity plan	Disaster plan	R. Miles	Risk Mgmt. rep.
isTested	tested	R. Miles	Risk Mgmt. rep.

Properties:

KPI property	Property value	Best-practice
Measurement frequency	Yearly	Quarterly
Interpretation		Good > 80% Normal 60% - 80% Problematic < 60%
KPI consumer	J. Smith	
KPI owner	J. Doe	
Target value	100% in 2015	80%
Planned value(s)	25% in 2013 75% in 2014	70%, 75%
Tolerance value(s)	5%	
Escalation rule	n.a.	

Goals

Ensure compliance
Foster innovation
Improve capability provision
Improve project execution
Increase disaster tolerance
Increase homogeneity
Increase management satisfaction
Increase transparency
Reduce operating cost
Reduce security breaches

Calculation

The number of critical applications where tested IT continuity plan available divided by the total number of critical applications.

Code

EAM-KPI-0001

Sources

CobiT 4.0

Layers

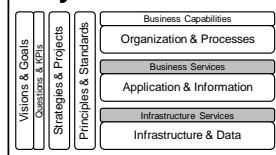


Figure 2.1.: An exemplary KPI instance of the EAM KPI Catalog [13] demonstrating the EAM KPI structure [12]

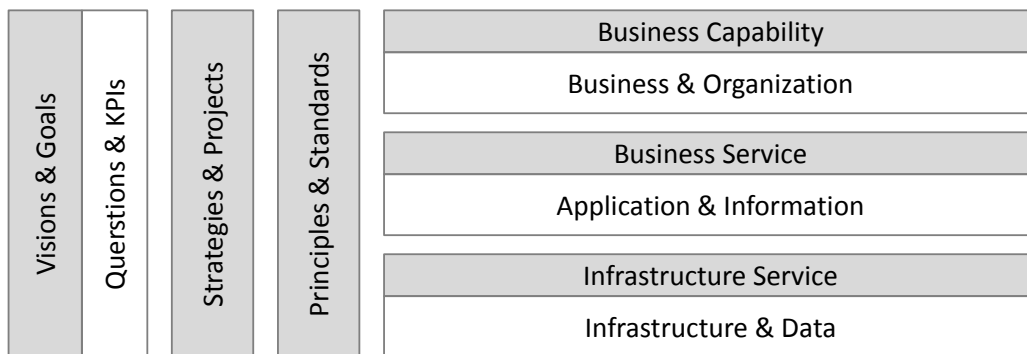


Figure 2.2.: EA layers and cross function based on Wittenburg [16]

Information model A KPI's information model describes the entities, relations, and attributes, which are required to compute the KPI. Figure 2.3 depicts an integrated data model combining the information model of each of the catalog's KPIs.

2.1.2. Organization-specific structure elements

In contrast to GSEs, OSSEs are dependent on the organizational context, i.e. the values of OSSEs will differ from enterprise to enterprise, since they are organization-specific properties. The OSSEs of the catalog are the following:

Measurement frequency This structure element defines, how often the KPI has to be evaluated

Interpretation The interpretation is a short description of how the calculated KPI value has to be interpreted, which value is good, acceptable, or bad (which could be visualized by traffic lights)

KPI consumer The KPI consumer is the person who is actually interested in the value of the KPI

KPI owner The KPI consumer is the person who is responsible for the KPI, i.e. the owner has to ensure the availability of the KPI's input data and the KPI's applicability

Target value This is the value, which has to be achieved

Planned value(s) This are milestones between the current value and the target value

Tolerance value(s) This element determines the allowed deviations from planned and target values

Escalation rule The escalation rule specifies the steps to be taken in case of a foreseeable non-achievement of the target EAM goals

In addition to these organization-specific properties, each KPI consists of a mapping table allowing the linking from the catalog's information model elements to organization-specific concepts. Of course, since each of the catalog's KPIs operates on different information model elements, this mapping table differs from KPI to KPI.

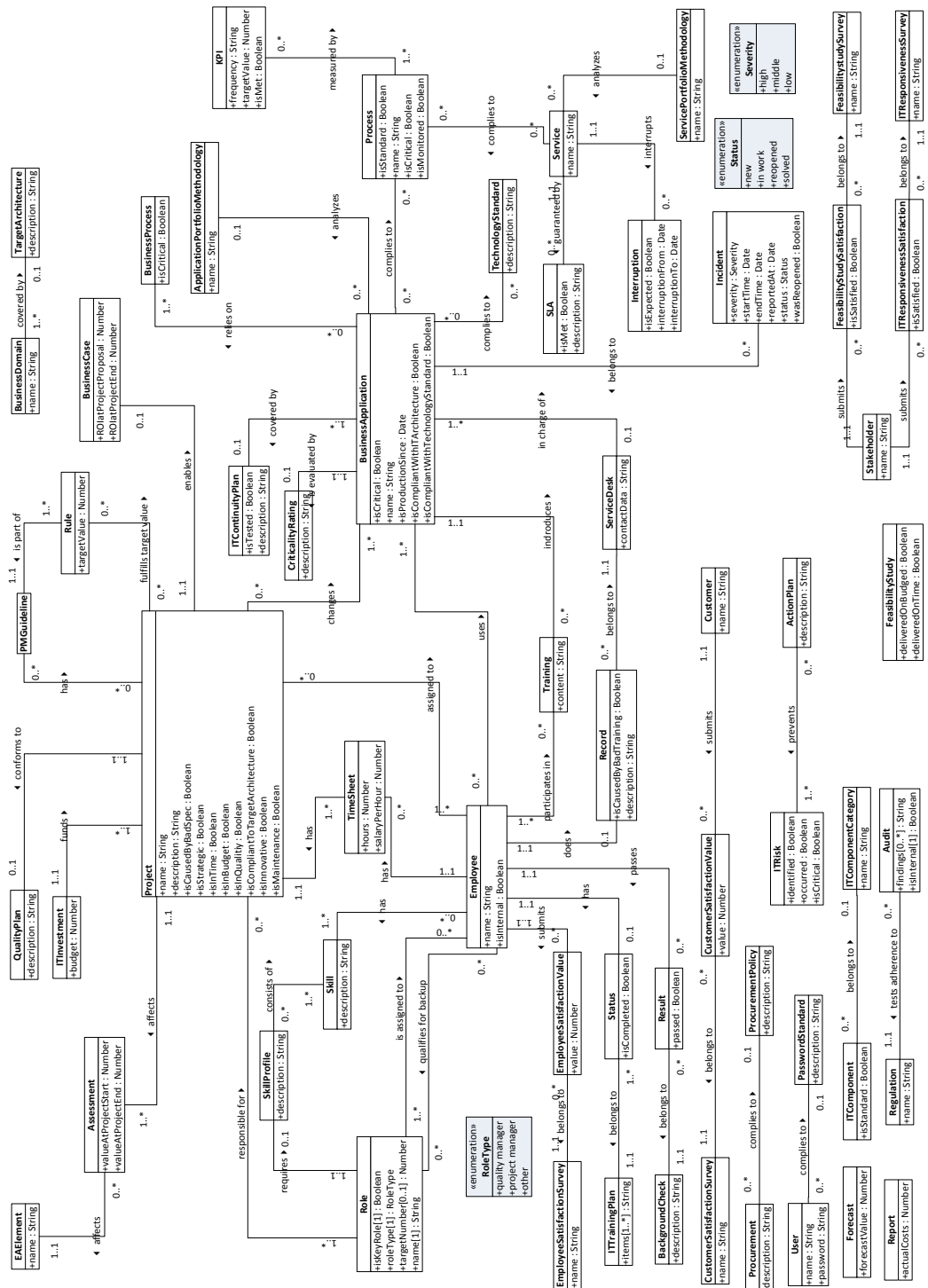


Figure 2.3.: UML class diagram [17] of the EAM KPI catalog's integrated data model

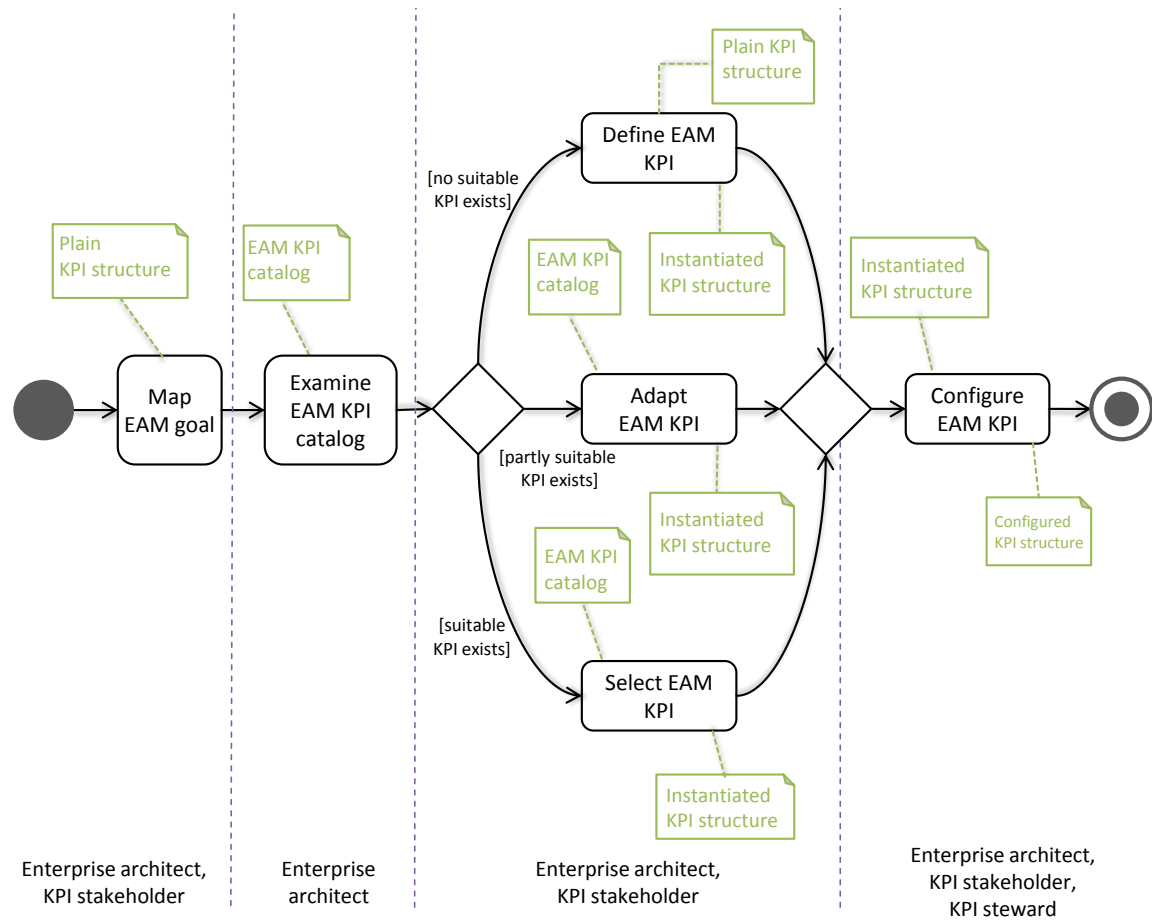


Figure 2.4.: UML activity diagram [17] showing the design method for defining an EAM KPI by Matthes et al. [14]

2.1.3. Design method for defining EAM KPIs

In practice, an enterprise architect won't implement all KPIs from the catalog, but he will search for the "right" candidates KPIs to measure the achievement of his organization-specific goals. In order to support the definition of EAM KPIs based on the EAM KPI structure as well as the catalog, our group developed a design method (depicted in Figure 2.4) consisting of the following four design steps [14]:

Map EAM goal The first activity is the determination of the EAM goals, which have to be achieved

Examine EAM KPI catalog Subsequently, the enterprise architect is able to search in the catalog for suitable EAM KPIs which are related to the selected EAM goals.

Instantiate EAM KPI If no suitable KPI can be found, a new EAM KPI has to be defined. Otherwise, the architect either accepts suitable KPIs from the catalog, or adapts partly suitable KPI to his specific needs.

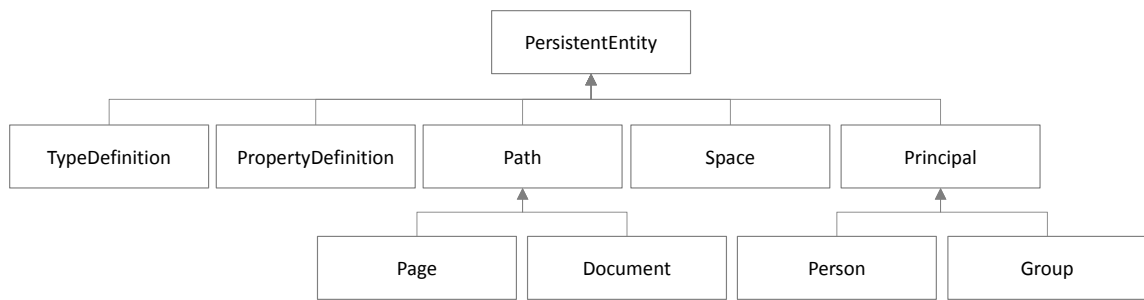


Figure 2.5.: UML class diagram [17] showing the basic model hierarchy of Tricia

Configure EAM KPI The last step is the tailoring of the KPIs to the organization-specific context by the configuration of their organization-specific structure elements.

2.2. Tricia

Tricia is a Java-based enterprise 2.0 wiki system initially developed by the sebis chair of the Technical University of Munich and now owned by the software company infoAsset¹ located in Munich. Its main purpose is a collaborative information management based on a flexible data model [18]. Moreover, its capabilities allow the employment as a collaborative EAM tool, as shown by Matthes et al. [9].

2.2.1. Architecture

The architecture of Tricia is based on the Model-View-Controller pattern, i.e., the web application's view (user interface) and its model (data) are basically decoupled and brought together by a controller component. This pattern ensures the separation of the user interface and the application's logic on the one hand, and facilitates the reuse of models and views on the other hand.

Models in Tricia

Tricia uses a flexible persistence layer to store the application's data, since it is possible to register an arbitrary database system, which will be used as the data storage for the whole data of Tricia. Therefore, the model layer, which abstracts the persistence layer, is independent of the underlying database system.

There is a set of basic models in Tricia, which are extensible by derivation from one of the existing model classes. The basic model hierarchy is depicted in Figure 2.5, whereas its parts are:

TypeDefinition & PropertyDefinition TypeDefinitions and PropertyDefinitions allow the definition of information model's schema at runtime, hence we call them schema objects. Each TypeDefinition contains an arbitrary number of PropertyDefinitions

¹<http://www.infoasset.de>

Master's Thesis Thomas Reschenhofer

Last modified by Thomas Reschenhofer Aug 9

student project master's thesis siemens masterarbeit masterthesis thesis

Design and prototypical implementation of a model-based structure for the definition and calculation of Enterprise Architecture Key Performance Indicators

Abstract

Due to the increasing complexity of Enterprise Architectures (EA), organization-specific Key Performance Indicators (KPI) are an important tool for the measurement of certain EA characteristics, and hence support the understanding of an EA's structure and its dynamics.

Therefore, our group published a catalog containing 52 KPIs gathered from industry and Enterprise Architecture Management (EAM) literature to support enterprise architects and business domain experts in the documentation of EAM KPIs. Furthermore, in previous research, we implemented the prototype of a domain-specific language (DSL) empowering business users to formally define KPIs.

Attributes of this Student Project

Title (de)	Konzeption und prototypische Implementierung einer Modell-basierten Struktur zur Definition und Berechnung von Unternehmensarchitekturkennzahl
Title (en)	Design and prototypical implementation of a model-based structure for the definition and calculation of Enterprise Architecture Key Performance Indicators
Project	EAM KPI Catalog
Type	Master's Thesis
Status	Completed
Student	Thomas Reschenhofer
Advisor	Ivan Monahov
Supervisor	Prof. Dr. Florian Matthes

Figure 2.6.: A Tricia page and its basic parts.

and can be assigned to information objects (e.g., pages, documents). Subsection 2.2.2 will cover this topic in detail.

Page & Document Pages are the main information objects in Tricia. As shown in Figure 2.6, they are consisting at least of a name, tags, an unstructured rich-text content, as well as of a type, attributes and relations. The attributes and relations of the page are either defined by the assigned TypeDefinition (by corresponding Property-Definitions) or free attributes (i.e., arbitrary name-value-pairs attached to the page). Since pages have both structured (type, attributes, and relations) and unstructured (rich-text content) data, they are called *Hybrid Pages* (c.f. Subsection 2.2.2 for more information).

Documents are very similar to pages, i.e., they are also consisting of the mentioned properties. However, in contrast to pages, they are directly related to some file uploaded to Tricia.

Space Spaces are containers for pages and documents as well as for TypeDefinitions and PropertyDefinitions, i.e., each information and schema object is part of exactly one space. Spaces are comparable to Java packages and are defining an own namespace, i.e., there may be two types with the same name in two different spaces, while all the types of one space have to have unique names. Moreover, Tricia allows the export and import of whole spaces and their objects.

Person & Group Since Tricia is a collaborative tool with a multitude of users, an authentication and authorization mechanism is obligatory. Therefore, after a common authentication process (by providing username and password), a user has either read-only access to certain Tricia objects (if the user is just a reader), or even write access (if the user is a writer of the object, which of course implies read access).

Controllers and Views in Tricia

Since Tricia is a web application, the MVC controller handles HTTP requests as shown in Figure 2.7:

1. The web server is responsible for the authentication of users and the creation (if new) or restoration of sessions
2. Based on the request URL, the web server forwards the request to a certain handler (controller)
3. A handler checks, if the current user is allowed to perform the requested handlers action
4. If the user is authorized, the handler loads appropriate models and performs the associated business logic (e.g., update the model)
5. Subsequently, the handler returns the view, which has to be presented to the requesting client. This view can be based on a template defining the layout and design of a view, while its content is instantiated by the view itself.

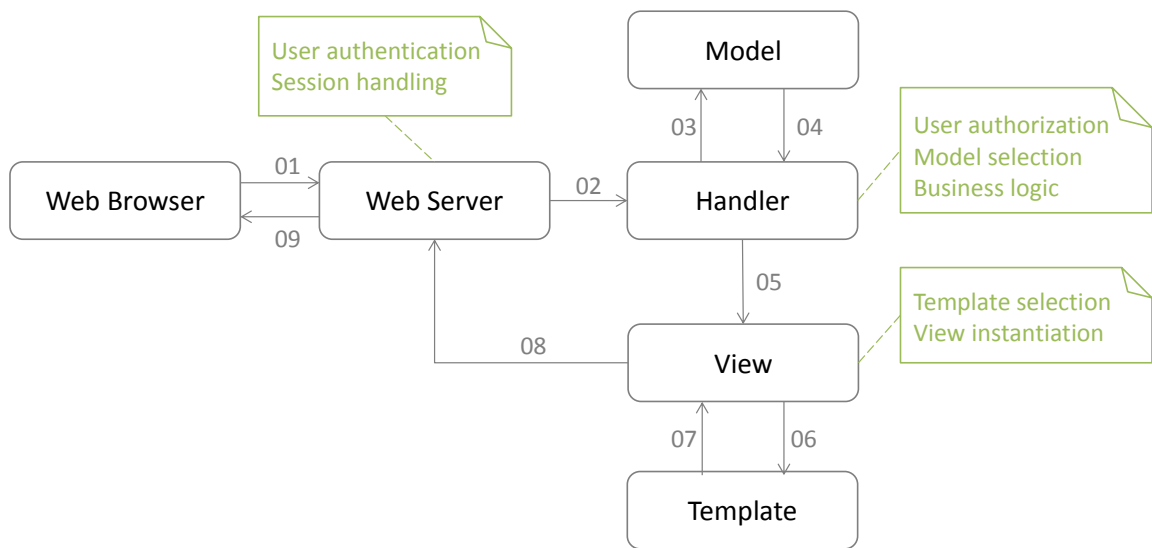


Figure 2.7.: Basic processing of a HTTP request to Tricia.

2.2.2. Hybrid wikis

The so-called hybrid wikis are one of the core concepts of Tricia. In this context, the term hybrid refers to an emergent enrichment of unstructured content (e.g., free text or documents, c.f. left side of Figure 2.6) with structure (types, attributes, and relations, sc.f. right side of Figure 2.6).

As already mentioned in the previous Subsection 2.2.1, two of Tricia's default models are

Attributes of type *Employee*





Attribute Name	Attribute Type	Multiplicity	Action
 Last name	Text	Exactly one value	Edit Delete
 Location	Reference <i>Department</i>	Exactly one value	Edit Delete
 Picture	Image	Exactly one value	Edit Delete
 Salary	Number	Exactly one value	Edit Delete

Figure 2.8.: An exemplary TypeDefinition *Employee* consisting of several PropertyDefinitions.

the type definition and the property definition. The type definition may consist of several property definitions, which in turn may define certain integrity rules:

Type If the property definition defines an attribute type, the attribute value of each of the type definition's instances has to be of this type, otherwise a warning is displayed in the instance. Tricia provides a basic set of attribute types, e.g., *Text*, *Number*, *Date*, *Boolean*, and *Reference* (Relation to other instances, optional restricted to instances of a certain type). For example, the PropertyDefinition *Location* in Figure 2.8 is of type *Reference*, whereas the referred object has to be of type *Department*.

Multiplicity If the property definition defines a multiplicity, the attribute of each of the type definition's instances has to have the number of values as defined by the property definition, otherwise a warning is displayed in the instance. The multiplicities provided by Tricia are *Any number*, *At least one*, *Exactly one*, and *Maximal one*. For example, all PropertyDefinitions in Figure 2.8 are defined with multiplicity *Exactly one*, so that each instance of type *Employee* has to provide exactly one value for each of its attributes.

The relations between the schema objects (type definition and property definition) and the information objects (page) are depicted in the hybrid wikis data model in Figure 2.9.

2.3. MxL 1.0

Until recently, the EAM KPI structure [12] lacked a formal computation prescription for its KPIs. Therefore, Monahov et al. [15] designed a DSL capable of defining all the catalog's KPIs. While this DSL was named Model-based Expression Language (MxL), its prototypical implementation in the wiki system Tricia is called Tricia Expression Language (TxL). Moreover, since this thesis covers further development of MxL, we call the initial version

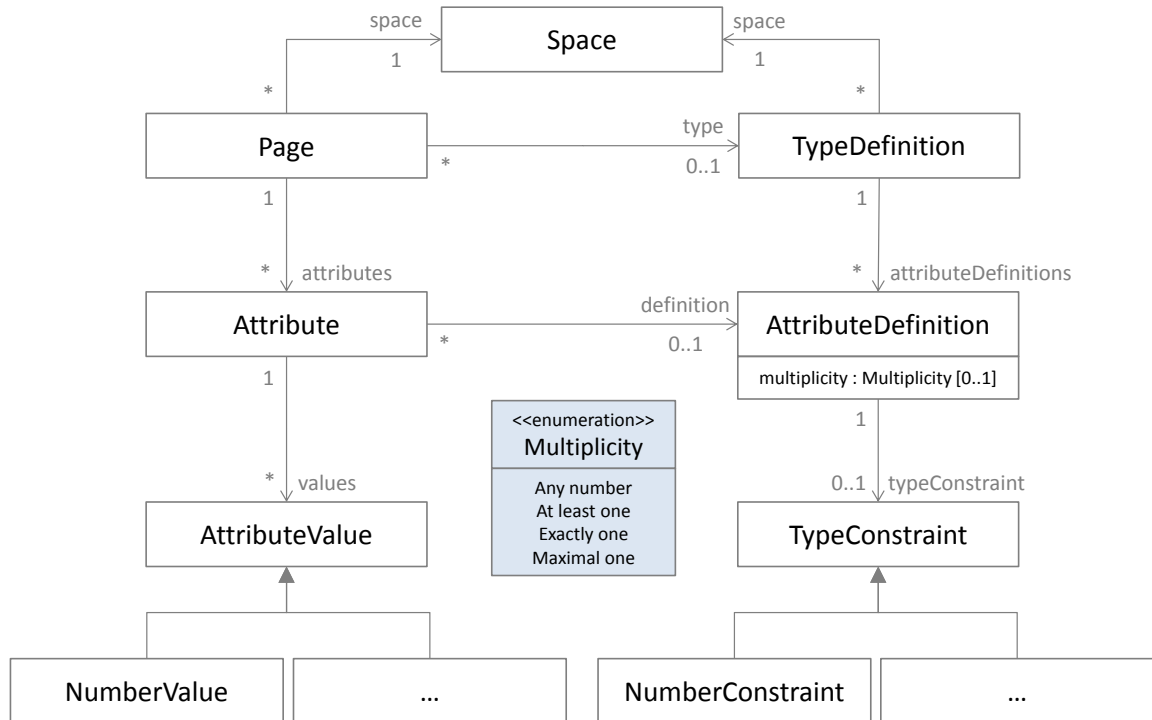


Figure 2.9.: UML class diagram [17] showing the abstract hybrid wikis data model by Matthes et al. [19].

designed by Monahov et al. [15] *MxL 1.0* (or *TxL 1.0*), while the version developed throughout this thesis (especially in Chapter 3) will be named *MxL 2.0* (or *TxL 2.0*).

2.3.1. Fundamentals of MxL 1.0

MxL 1.0 was inspired by both the Object Constraint Language (OCL) [20] and Microsoft’s Language Integrated Query (LINQ) [21]. However, it was tailored to the EAM domain. The most important properties of MxL 1.0 are:

Functional programming Functional programming is characterized by the absence of side effects and furthermore implies some other language features like higher-order functions and recursion [22, 23]. Higher-order functions are functions, which can take other functions as arguments. This is especially useful for MxL 1.0’s basic operators covered later in this subsection.

Object-orientation Since Tricia’s information objects have types, attributes, and relations, an object-oriented language allows their representation by complex objects, which allows a convenient access to the information object’s data. However, one of most-important concepts of object-orientation – namely inheritance – is not supported by MxL 1.0. Reuse of functionality is achieved by the concept of delegation [22].

Sequence-orientation One important purpose of MxL 1.0 is querying the underlying data model. Hence, MxL 1.0 has to handle whole sequences (ordered multi-sets) of data objects by applying filters, projections, etc.

Dynamic type system MxL 1.0 is dynamic typed [22, 23], i.e., the type of objects in an expression are determined at runtime. Technically, the MxL 1.0 interpreter misses a type checker component, i.e., the MxL 1.0 interpreter consists of a scanner, a parser, and an evaluation engine.

Dynamic binding If a function is invoked on an object, the evaluation engine looks up the proper function at runtime [22, 23].

MxL 1.0 was designed to be minimal in the sense of providing just a minimal set of language constructs. As a result, MxL 1.0 does not provide constructs for arithmetic operations (addition, subtraction, multiplication, division), string concatenation, or comparison operations (equality and inequalities). However, MxL 1.0 supports the conditional, a name-binding-construct, lambda-expressions (anonymous functions), as well as a comment-construct:

```
/* conditional */
<condition> ? <ifbranch> : <elsebranch>

/* let-construct */
let <name> = <value> in <scope>

/* lambda */
? (<parameters>) <method-stub>
```

Name	Description
String	A character sequence enclosed by quotation marks
Number	Both integers and decimals, as well as strings representing a number
Boolean	The language constructs <i>true</i> and <i>false</i> , as well as the strings "true", "false", "yes", and "no"
Date	Strings representing a date
Sequence	A ordered multi-set of data, i.e., the order of elements matters and duplicates are allowed
Map	A fixed collection of key-value-pairs, whereas the notation for the definition of maps is similar to the JavaScript Object Notation (JSON)
Function	Since MxL 1.0 is a functional language, functions are first-class objects [22] of type Function
Entity	Each object of the underlying information model is of this type

Table 2.1.: Basic types of MxL 1.0

Custom MxL Function

STATIC::applicationContinuityPlanAvailabilityKPI

Type	
Name	getApplicationContinuityPlanAvailabilityKPI
Parameters	
Description	A measure of how completely IT continuity plans for business critical applications have been drawn & tested up for the IT's application portfolio
Method Stub	<pre>// Determine all critical business applications let criticalApplications = find("Business Application", "is critical", "yes") in // Determine all critical business applications // with tested IT continuity plan let criticalApplicationsWithCoveringContinuityPlan = criticalApplications.where(? (ca) (ca.hasTestedContinuityPlan())) in // Calculate proportion of critical business applications criticalApplicationsWithCoveringContinuityPlan.count() .div(criticalApplications.count())</pre>

Figure 2.10.: The implementation of the EAM KPI depicted in Figure 2.1 as a MxL 1.0 custom function [15].

Since MxL 2.0 will provide similar constructs, Subsection 3.1.2 will cover this topic in more detail.

Furthermore, in addition to the types defined in the underlying information model, MxL 1.0 also provides a set of basic types, which is listed in Table 2.1. Since handling and processing sequences is one of the main purposes of MxL 1.0, the language provides a set of basic functions to filter, sort, group, or to apply other query operators onto a arbitrary sequence of objects. The set of functions provided by MxL 1.0 is based on Microsoft's *Standard Query Operators* [24] and contains *i.a.*, common query functions (e.g., *where*, *select*), aggregation functions (e.g., *count*, *sum*), and set functions (e.g., *concat*, *intersect*). These functions are also subject in Subsection 3.1.4, where they will be explained in further detail.

2.3.2. Use cases of MxL 1.0

While the motivation for the design of MxL 1.0 was the formal definition of EAM KPIs, its implementation in Tricia enables many use cases:

Custom functions In order to reuse MxL expressions, users can create so-called custom functions, e.g., for each of the catalog's KPIs a custom function was created formally defining the KPI's computation (c.f. Figure 2.10). Therefore, to evaluate the KPI, the user just has to invoke the corresponding custom function.

Derived attributes While the values of common attributes are persisted in Tricia's database, the values of derived attributes are computed according to a MxL 1.0 expression.

Therefore, derived attributes are able to make dependencies between information model elements explicit by defining the dependency a proper MxL 1.0 expression.

Embedded expressions MxL 1.0 expressions can be embedded in the rich-text content of a page, which enables the dynamic generation of HTML-based visualizations, e.g., based on a certain condition, the MxL 1.0 expression returns an HTML image showing either a green, yellow, or red traffic light.

In addition to the evaluation of MxL 1.0 by the implementation of all the catalog's KPIs, this language and its prototypical implementation were also deployed in an EU project called *SmartNet Navigator* [25, 26], whereas the main purpose of MxL 1.0 in this project was the dynamic generation of a visualization of a project status. This shows the applicability of MxL 1.0 in areas outside of EAM.

2.4. Shortcomings of existing foundations

As stated in Section 1.3, the goal of the thesis is the prototypical implementation of the Living KPIs – an integrated and flexible EAM platform based on the EAM KPI structure. The Living KPIs prototype will be based on Tricia as well as MxL 1.0, since these technologies already provide a set of useful features.

For example, since Tricia implements the Hybrid Wikis concept (c.f. Subsection 2.2.2), it supports the collaborative creation of a data model, i.e., a multitude of users is capable of contributing to the emergence of the data model. Furthermore, Tricia implements the MxL 1.0 prototype, wherefore there is already the possibility to formally define and automatically evaluate the EAM KPIs.

However, there are still some shortcomings regarding the implementation of the Living KPIs prototype.

2.4.1. No compile-time analysis of MxL expressions

The Living KPIs have to be easily adaptable to an organization-specific context, i.e., it should be possible to delete or rename certain information model elements without violating the integrity of the model.

However, since the formal definitions of the KPIs represented by MxL 1.0 expressions refer to these model elements (like the custom function in Figure 2.10, which refers to a type with name "Business Application" and an attribute with name "is critical"), deleting or renaming model elements would make these expressions invalid. Moreover, since MxL 1.0 is dynamically typed and uses dynamic dispatching (as mentioned in Subsection 2.3.1), the system would not even recognize the expression's invalidity until a re-evaluation leads to a runtime exception.

Chapter 3 tackles this problem by developing MxL 2.0, the successor of MxL 1.0. In contrast to MxL 1.0, the 2.0 version will be type safe, which enables further analysis of the expression, e.g., which information model elements the expression refers to. This analysis allows to determine all expressions referring to a certain information model element and to handle the model element's deletion or renaming in a proper way.

2.4.2. Tight coupling between MxL 1.0 and Tricia

While the Living Catalog is prototypically implemented in Tricia, there are many other possible use cases requiring a DSL capable of defining queries onto an information model and performing calculations on the obtained results (e.g., the mentioned EU project *Smart-Net Navigator* [25, 26]). However, these use cases may require an implementation in other tools than Tricia, which – because of the tight coupling between MxL 1.0 and Tricia – rules out the usage of MxL.

Therefore, MxL 2.0 is decoupled from Tricia and is available as a separate Java archive (mxl.jar). Hence, MxL 2.0 is easily integratable in other tools by the implementation of one of MxL 2.0's components (MxL Connector, c.f. Subsection 3.2.3), which manages the interaction between the language and the implementing tool.

As a consequence, Sections 3.1 and 3.2 are completely independent from MxL 2.0's implementation in a tool, while just Section 3.3 focuses on its implementation in Tricia.

2.4.3. Missing type-based template engine in Tricia

In order to facilitate the familiarization into the EAM platform – especially for people already familiar with the EAM KPI structure or the catalog – the layout and design of the Living KPIs has to be based onto the EAM KPI structure as defined by Matthes et al. [12]. Hence each page representing one of the catalog's KPIs has to look like the KPI description in Figure 2.1.

However, while the Hybrid concept of Tricia (c.f. 2.2.2) allows the definition of integrity rules for a type, so that each of the type's instances will be checked according to these rules, it is not possible to influence the appearance a certain type's instances. This leads to the following problems:

- If the EAM KPI structure changes (e.g., by adding or removing a structure element, reordering of structure elements, ...), each page representing a KPI has to be updated.
- If the Living KPIs have to be extended by a new KPI, the creator of the KPIs has to take care of the page's layout.
- Managing structure elements both as attributes of the page (to allow structured access) and parts of rich-text content (to provide a familiar layout) yields to redundancies.

Section 4.1 covers the solution for this problem, which is a light-weight and type-based template engine. This allows the definition of a template for each type definition, whereas this template is applied onto each of the type's instances.

2.4.4. Insufficient deployment support in Tricia

The idea of the EAM KPI Catalog is to provide practice proven KPIs, which are adaptable to and employable in their organization-specific environment. The Living KPIs environment's goal is to take this idea to another level by the provision of an integrated and flexible EAM platform implementing all KPIs from the EAM KPI Catalog. Therefore, the Living

KPIs environment has to provide the EAM KPI structure's elements, all the EAM KPI Catalog's KPIs, as well as its whole integrated information model.

However, Tricia does not support an intuitive and descriptive definition of initial data sets. Since a hard-coded approach (definition of the structure, EAM KPIs, and integrated information model in Java code) is rather inflexible and unintuitive, Section 4.2 covers the implementation of a descriptive approach for the definition of initial data sets in Tricia as well as the finalization of a prototype of the Living KPIs.

Part II.

Contribution of the Thesis

3. MxL 2.0

With the development of MxL 1.0 by Monahov et al. [15], there is already a language capable of formally defining the KPIs of the EAM KPI Catalog. However, since MxL 1.0 uses a dynamic type system as well as a dynamic dispatching mechanism, an analysis of a MxL 1.0 expression at compile time is not possible.

However, as stated in Subsection 2.4.1, a compile-time analysis is inevitable for the implementation of the Living Catalog. Therefore, this chapter covers the design and implementation of an advanced MxL version, namely MxL 2.0. Doing this, Section 3.1 will explain the basics of MxL 2.0, Section 3.2 will focus on its technical aspects. While these two chapters about MxL 2.0 are rather independent from implementation aspects, Section 3.3 goes into more detail on TxL 2.0, the implementation of MxL 2.0 in Tricia.

3.1. Fundamentals of MxL 2.0

While the type checker is certainly the main feature added to MxL 2.0 (c.f. Section 3.2), there are many other changes and improvements in MxL 2.0 compared to its predecessor.

3.1.1. MxL 2.0 types

While MxL 1.0 does not support inheritance, MxL 2.0 uses this fundamental concept of the object-orientation paradigm in order to reuse functionality. However, the type hierarchy of the basic types is rather simple, since each of them derives from type *Object* (except *Object* itself). All basic types of MxL 2.0 are listed in Table 3.1.

This set of basic types may be extended by a specific implementation of MxL 2.0, e.g., TxL 2.0 adds the types *Page*, *Document*, *Principal*, *Person*, and *Group* (based on Tricia's models, cf Subsection 2.2.1), whereas each of them derives (directly or indirectly) from type *Entity*. The resulting type hierarchy of TxL 2.0 is depicted in Figure 3.1.

3.1.2. Basic language constructs

While MxL 1.0 desired simplicity in the sense of a minimal set of language constructs, MxL 2.0 focuses a clear syntax by providing common constructs.

Comments

To explain certain parts of an arbitrary MxL 2.0 expression, they can be annotated with textual comments by the following construct:

```
/* This is a simple textual comment */
```

Name	Description
Object	Each element of MxL's underlying information model is of type <i>Object</i>
String	Each character sequence encapsulated in quotation marks is a value of type <i>String</i> , e.g., "hello world"
Number	Represents both integers and decimals, e.g., 123.456
Boolean	true and false, but also language specific string like "yes" and "no"
Date	A date consisting of day, month, and year. Can be constructed by the date-function and the date's string representation. The current date can be determined by the global identifier <i>Today</i> . The components of a date are accessible via <i>day</i> (e.g., <i>Today.day</i>), <i>month</i> (e.g., <i>Today.month</i>), and <i>year</i> (e.g., <i>Today.year</i>)
Map	A fixed collection of key-value-pairs. The notation is similar to the JavaScript Object Notation (JSON), e.g., { <i>number</i> : 1, <i>title</i> : "hello world"}
Entity	An entity is a complex object, i.e., an object with attributes and/or relations to other other entities
Sequence	An ordered multi-set of values, written as [<i>element1</i> , <i>element2</i> , ...]. An ordered multi-set is a collection, whose order matters and which allows duplicates. The type <i>Sequence</i> can be parametrized to determine the type of the sequence's elements, e.g., the type <i>Sequence</i> < <i>Number</i> > defines a sequence of numbers. The elements of a sequence are accessible via [] and the element's index, whereas the index is zero-based.
Function	Because MxL 2.0 allows higher-order functions, there are objects of type <i>Function</i> . Again, this type can be parametrized to determine the function's signature (parameter types and return type), e.g., the type <i>Function</i> < <i>Number</i> , <i>Number</i> , <i>Boolean</i> > defines a function with two parameters of type <i>Number</i> returning an object of type <i>Boolean</i> . Moreover, parameter types can be defined as optional by a question mark (the function can be invoked without optional parameters), e.g., <i>Function</i> < <i>Number</i> , <i>Number</i> ?, <i>Boolean</i> > can be invoked for either one or two parameters.
Type	A meta-type representing types, e.g., the types <i>Number</i> and <i>String</i> are also objects of type <i>Type</i>
Space	Represents a workspace/package consisting of types, static functions, and instances (comparable to Java packages)

Table 3.1.: Basic types of MxL 2.0, whereas each of them derives from type *Object* (except *Object* itself)

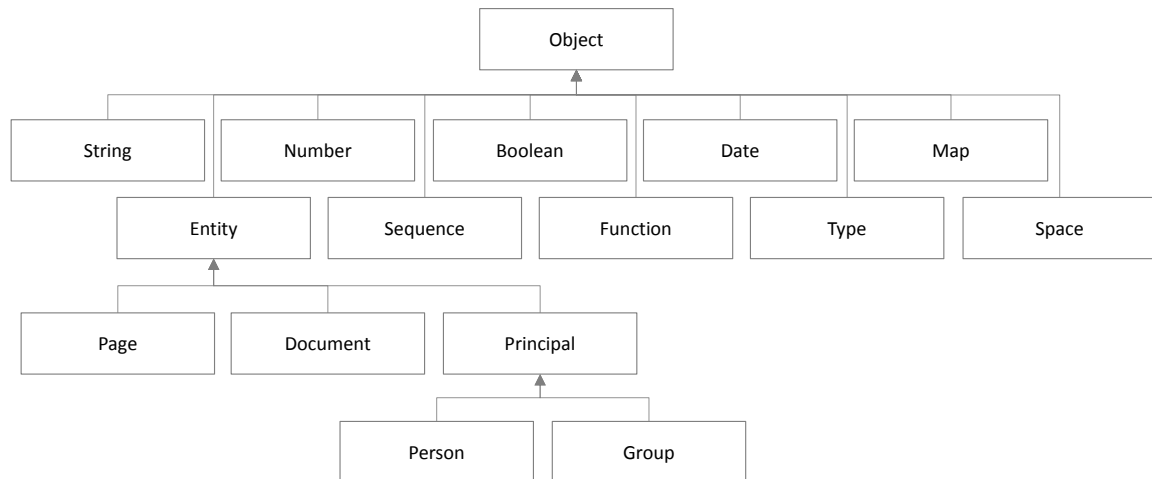


Figure 3.1.: UML class diagram [17] showing the type hierarchy of MxL 2.0 extended by TxL 2.0's basic types *Page*, *Document*, *Principal*, *Person*, and *Group*.

Arithmetic operators

MxL 2.0 provides constructs for the arithmetic addition, subtraction, multiplication, division, and exponentiation:

```

1.0 + 2.0 /* Addition */
3 - 0.1 /* Subtraction */
3.14 * (2 * 3) /* Multiplication */
1.0 / 0 /* Division, denominator = 0 yields to an exception */
2 ^ 8 /* Exponentiation */

```

Of course, the parser takes care of the common operator precedence [22], e.g., $1 + 2 * 3$ will be evaluated as $1 + (2 * 3)$.

If the first operand of the plus operator is not a number, but a string, the string concatenation is applied, e.g., `"Hello " + "World"` will be evaluated to `"Hello World"`.

Furthermore, the minus operator can also be applied onto two objects of type *Date* to calculate the difference between them in terms of days, e.g., `Today - date("15.03.2013")` retrieves the number of days passed since 15th March 2013.

Comparison and logical operators

In order to compare certain instances, MxL 2.0 provides also a basic set of common comparison operators, namely the equality (`=`) and the inequalities (`<>`, `>`, `>=`, `<=`, `<`). These operators can be applied on both numbers and dates.

Furthermore, to combine multiple comparisons, MxL 2.0 provides the logical inversion (not), conjunction (and), and disjunction (or), whereas inversion will be evaluated before the conjunction, which in turn will be evaluated before the disjunction. Hence, the expression

```
1 > 1 or 5 <> 3 + 2.1 and not false
```

will be evaluated as

```
(1 > 1) or ((5 <> (3 + 2.1)) and (not false))
```

Conditional

The conditional construct consists of three expressions (condition as well as if-branch and else-branch), whereas the construct evaluates either the if-branch (if the condition evaluates to *true*) or the else-branch (if the condition evaluates to *false*). The result of the conditional construct is the result of the evaluated branch.

Compared to MxL 1.0, the conditional's syntax has changed from a ternary operator (*<condition>? <ifbranch>: <elsebranch>*) to a traditional if-then-else-statement:

```
if <condition>
then <ifbranch>
else <elsebranch>
```

Name-value binding

To simplify certain expressions, it may be beneficial to bind a certain value to a certain name, whereas this name can be used in the remaining expression.

The syntax of this construct is the same as in MxL 1.0, i.e.:

```
let <name> = <value> in
<scope>
```

The MxL 2.0 interpreter evaluates the value and make it accessible in the scope via the given name.

Lambda expressions

A lambda-expression is an anonymous function-expression.

MxL 2.0 introduces a new syntax for lambda expressions, which is based on the syntax of the C# version of Microsoft's LINQ[21]:

```
/* syntax */
(<paramname1>:<paramtype1>, <paramname2>:<paramtype2>, <...>)
=> <methodstub>
```

```
/* example */
let addition = (a:Number, b:Number) => a + b in
addition(2,3)
```

The parameters of the function are defined before the arrow as name-type-pairs, whereas the method stub, which may access the parameters by their identifiers, is defined afterwards. If there is only one parameter, the brackets around the parameter can be omitted:


```
/* syntax */
<paramname>:<paramtype> => <methodstub>
```

```
/* example */
let inc = a:Number => a + 1 in
inc(1)
```

A function without parameters is defined by empty brackets:

```
/* syntax */
() => <methodstub>
```

```
/* example */
let one = () => inc(0) in
one()
```

Furthermore, if the parameter types can be inferred by the environment, they can be omitted (as shown in Subsection 3.1.4).

Type Checking and Type Casting

The implementation of a type checker in MxL 2.0 also implies the need for constructs for checking an object's type as well as for casting an object to a certain type.

The binary type checking operator (*<object>is <type>*) checks at runtime if the given object is of the given type and returns either true or false, e.g.:

```
"Hello World" is String /* true */
"Hello World" is Object /* true */
"Hello World" is Number /* false */

[1,2,3] is Sequence /* true */
[1,2,3] is Sequence<Number> /* true */
[1,2,3] is Sequence<String> /* false */
```

The binary type casting operator (*<object>as <type>*) tries to cast the given object to the given type. If this is not possible, this construct would throw an exception. This construct is especially useful for map-values, since the *Map* type does not provide parametrization (to determine the types of the map's attributes) and each of the map's value is of type *Object*, e.g:

```
"Hello World" as String /* OK */
{text: "Hello World", number: 4.0}.text as String /* OK */
{text: "Hello World", number: 4.0}.text as Number /* Exception */
```

3.1.3. Higher-order functions in MxL 2.0

As described later on in Subsection 3.1.4, higher-order functions in MxL 2.0 are an important tool for the definition of queries against an information model, since they provide a flexible mechanism to define custom predicates or other functions as parameters to the

query operators.

Higher-order functions are functions, which can take functions as parameters, e.g., a function *applyFunc* of type *Function<Function<Date, Number>, Date, Number>* is a higher-order function expecting a function and a date as parameter and returning a number. An exemplary application of the *applyFunc*-function could look like follows:

```
let getDay = (d:Date) => d.day in
applyFunc (getDay , Today)
```

Another possibility is to pass the lambda expression directly as parameter to the *applyFunc*-function:

```
applyFunc ((d:Date) => d.day , Today)
```

However, since the type of the lambda expression can be inferred by the expected parameter type of the *applyFunc*-function, which is *Function<Date, Number>*, the parameter types of the lambda expression can be omitted:

```
applyFunc(d => d.day , Today)
```

Implicit lambdas

The previous expression can be even further shortened by the use of so-called "implicit lambdas":

```
applyFunc (day , Today)
```

Since the *applyFunc*-function expects a function of type *Function<Date, Number>* as its first parameter, but realizes that *day* is neither a function nor a known identifier, it will try to interpret this parameter as the method stub of a lambda, whereas the identifier *day* – since not a known global identifier – will be implicitly evaluated as a member of an implicit lambda parameter. Because the application of an implicit lambda yields to a valid expression, the type checker will implicitly interpret the expression *applyFunc(day,Today)* as

```
applyFunc(<implicit param> => <implicit param>.day , Today)
```

However, since there can be just one implicit lambda parameter, implicit lambdas just work for higher-order functions expecting either a one-dimensional function (functions with one parameter) or functions without parameters (which obviously do not have an implicit parameter).

The advantage of implicit lambdas is a very intuitive spelling of certain higher-order functions (c.f. Subsection 3.1.4), which for example is comparable to clauses of the well-known Structured Query Language (SQL).

3.1.4. Sequence functions

One of MxL's main purposes is the definition of queries against an information model [15]. Therefore, based on the sequence of objects of a specific type, MxL has to be able to apply certain filters, projections, aggregation, etc. onto this sequence. For this purpose, MxL 2.0 provides an extensive set of sequence functions based on Microsoft's *Standard Query*

Operators [24] and the evaluation of MxL 1.0 by Monahov et al. [15]. All sequence functions provided by MxL 2.0 are listed in Tables 3.2, 3.3, 3.4, 3.5, 3.6, and 3.7.

Name	Parameters & Return type	Description
select	map : Function<T, V> returns : Sequence<V>	Applies the map-function to each element of the source sequence and returns a sequence containing the results of each individual application
selectMany	map : Function<T, Sequence<V>> returns : Sequence<V>	Similar to the select-function, however, in selectMany, the map-function returns a sequence for each element. The concatenation of all sequences forms the result of the selectMany-function.
where	pred : Function<T, Boolean> returns : Sequence<T>	Filters the source sequence by the given predicate, i.e., all elements fulfilling the predicate remain in the sequence.
groupby	keySel : Function<T, Object> f : Function<Sequence<T>, Object>? returns : Map	Groups the elements of the source list by the keySel-Function and applies the (optional) f-function on the elements of each single group
orderby	keySel : Function<T, Object>? descending : Boolean? returns : Sequence<T>	Sorts the source sequence by the (optional) keySel-function, whereas a natural order will be applied. The (optional) descending parameter determines, if the elements should be ordered ascending or descending.

Table 3.2.: MxL 2.0's common query functions. In this table, all functions are applied on Sequences of type *Sequence<T>*, whereas *T* and *V* are arbitrary MxL 2.0 types.

Name	Parameters & Return type	Description
any	pred : Function<T, Boolean> returns : Boolean	Returns <i>true</i> , if at least one element of the source sequence fulfills the given predicate, otherwise <i>false</i>
all	pred : Function<T, Boolean> returns : Boolean	Returns <i>true</i> , if each element of the source sequence fulfills the given predicate, otherwise <i>false</i>
none	pred : Function<T, Boolean> returns : Boolean	Returns <i>true</i> , if no element of the source sequence fulfills the given predicate, otherwise <i>false</i>
contains	element : T returns : Boolean	Returns <i>true</i> , if the given element is contained in the source sequence, otherwise <i>false</i>
isEmpty	returns : Boolean	Returns <i>true</i> , if the source sequence has no elements, otherwise <i>false</i>
isNotEmpty	returns : Boolean	Returns <i>true</i> , if the source sequence has at least one element, otherwise <i>false</i>

Table 3.3.: MxL 2.0's quantifier functions returning a boolean value. In this table, all functions are applied on Sequences of type *Sequence<T>*, whereas *T* is an arbitrary MxL 2.0 type.

Name	Parameters & Return type	Description
distinct	returns : Sequence<T>	Removes all duplicates of the source sequence
except	other : Sequence<T> returns : Sequence<T>	Returns a sequence with all elements contained in the source sequence, but not in the other one
intersect	other : Sequence<T> returns : Sequence<T>	Returns a sequence with all elements contained in the source sequence and in the other one
concat	other : Sequence<T> returns : Sequence<T>	Concatenates the source sequence with the other one, i.e., the resulting sequence contains all elements of the source sequence, followed by all elements of the other one

Table 3.4.: MxL 2.0's set functions produce a sequence based on the presence or absence of an equivalent element within the same or another sequence. In this table, all functions are applied on Sequences of type *Sequence<T>*, whereas *T* is an arbitrary MxL 2.0 type.

Name	Parameters & Return type	Description
first	pred : Function<T, Boolean>? returns : T	Returns the first element of the source sequence (or the first element satisfying the predicate). If there is not such an element, this function throws an exception
last	pred : Function<T, Boolean>? returns : T	Returns the last element of the source sequence (or the last element satisfying the predicate). If there is not such an element, this function throws an exception
single	pred : Function<T, Boolean>? returns : T	Returns the only element of the source sequence (or the only element satisfying the predicate). If there is not such an element, or if there is more than one element, this function throws an exception

Table 3.5.: MxL 2.0's element functions choosing a certain element of the source sequence. In this table, all functions are applied on Sequences of type *Sequence<T>*, whereas *T* is an arbitrary MxL 2.0 type.

Name	Parameters & Return type	Description
rest	returns : Sequence<T>	Returns the source sequence without the first element
take	n : Number returns : Sequence<T>	Returns a sequence with the first <i>n</i> elements of the source sequence
takeWhile	pred : Function<T, Boolean> returns : Sequence<T>	Returns all elements of the source sequence until an element does not satisfy the predicate
skip	n : Number returns : Sequence<T>	Returns a sequence without the first <i>n</i> elements of the source sequence
skipWhile	pred : Function<T, Boolean> returns : Sequence<T>	Skips all elements of the source sequence as long as these elements satisfy the predicate, and returns the rest

Table 3.6.: MxL 2.0's partitioning functions dividing the source sequence into two sections and return one of them. In this table, all functions are applied on Sequences of type *Sequence<T>*, whereas *T* is an arbitrary MxL 2.0 type.

Name	Parameters & Return type	Description
count	pred : Function<T, Boolean>? returns : Number	Counts all elements of the source sequence (or counts the elements satisfying the predicate)
ratio	pred : Function<T, Boolean> returns : Number	Returns a number between 0 and 1 representing the ratio of elements fulfilling the given predicate
sum	map : Function<T, Number>? returns : Number	Sums up all numbers of the source sequence. The optional map-function may select a numerical member of each element
average	map : Function<T, Number>? returns : Number	Computes the average of all numbers of the source sequence. The optional map-function may select a numerical member of each element
max	map : Function<T, Object>? returns : T	Determines the maximal element of the source sequence. The optional map-function may select a criterion used for the selection of the maximum
min	map : Function<T, Object>? returns : T	Determines the minimal element of the source sequence. The optional map-function may select a criterion used for the selection of the minimum
aggregate	func : Function<V, T, V> seed : V returns : V	This is a fold-operator aggregating the current sequence to a single value by the given func-function. The func-function is invoked for the result of its previous invocation and each of the source sequence's elements. For the first iteration of the func-function the seed value is used. The result of the last invocation of the func-function is the result of the aggregate-function.

Table 3.7.: MxL 2.0's aggregation functions folding up all elements of the source sequence to a single value. In this table, all functions are applied on Sequences of type *Sequence<T>*, whereas *T* and *V* are arbitrary MxL 2.0 types.

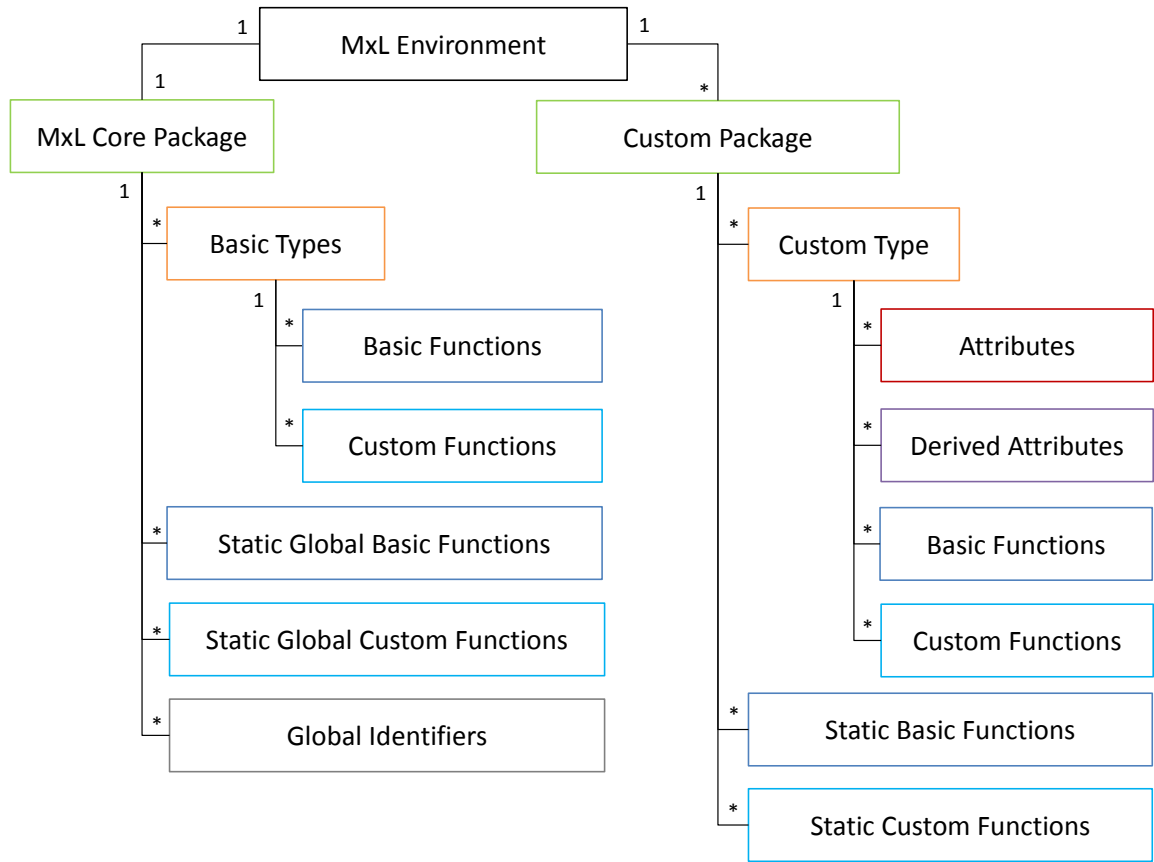


Figure 3.2.: The MxL 2.0 asset hierarchy.

3.1.5. MxL Asset Hierarchy

MxL assets are elements of the MxL environment (e.g., implementing system), which are referable by identifiers in MxL expressions, i.e., these assets are usable in the definition of MxL expressions.

While previous subsections in this chapter already introduced some basic assets of MxL 2.0 (e.g., basic types and sequence functions), there are still some more assets in MxL 2.0. All assets as well as their relations are depicted in the MxL 2.0 asset hierarchy in Figure 3.2.

The root of the hierarchy is the MxL Environment, which is the system the language is implemented in. Hence, TxL 2.0's environment is Tricia.

The only direct components of the MxL Environment are its packages, whereas there is exactly one MxL Core Package and an arbitrary number of additional custom packages. All basic assets of MxL 2.0 are encapsulated in the MxL Core Package. This includes the basic types mentioned in Subsection 3.1.1 as well as the sequence functions described in Subsection 3.1.4, which are part of the basic type *Sequence*.

In general, MxL 2.0 supports two types of functions:

Basic functions These functions are defined in the implementation language of MxL 2.0 – Java. Hence, basic functions are compiled and are not definable at runtime (c.f.

Subsection 3.2.5). However, they allow to extend MxL 2.0's expressiveness by executing arbitrary Java code.

Custom functions Custom functions are defined in MxL 2.0. Hence, these functions can be defined at runtime, which enables reusability of expressions. Of course, custom functions can call other custom (and basic) functions enabling compositions of functions.

Both basic functions as well as custom functions can be defined for a certain owner type (e.g., the owner type of the sequence functions is *Sequence*) allowing the application of the function onto an object of its owner type. Obviously, all functions defined for basic types are also part of the MxL Core package.

An example for a custom function with an owner type would be the function *getCentury* implemented for type *Date*, expecting no parameters, and implemented as follows:

```
(this.year - 1) / 100 + 1
```

An exemplary invocation of the *getCentury*-function would be the following:

```
Today.getCentury() /* returns the current century */
```

Since this custom function has an owner type, the object, on which the function will be applied, is accessible via the *this*-keyword (whose type is obviously the function's owner type). However, MxL 2.0 also supports an "implicit this", i.e., if the type checker finds an unknown identifier, it will check if the identifier is a member of the object represented by *this*. Therefore, the following implementation of the *getCentury*-function would be equivalent to the previous one:

```
(year - 1) / 100 + 1
```

Both the "implicit this"-feature and the "implicit lambda"-mechanism (c.f. Subsection 3.1.3) equip MxL 2.0 with some kind of intelligence by providing "semantic sugar" and facilitating – at least in some cases – a shortened and hence clearer syntax.

Functions without an owner type are called *static*, e.g., the function *date* parsing the string representation of a date to an object of type *Date*. Hence, when implementing a static function, the *this*-object is null. Static functions belonging to the MxL Core Package are also called *global*.

The last remaining assets of the MxL Core Package are the global identifiers representing globally available name-value-bindings. For example, the identifier *Today*, which was already used in previous examples, is a global identifier returning the current date.

Depending on the system MxL 2.0 is implemented in, there may be additional custom packages, which may have custom types. In TxL 2.0, MxL 2.0's implementation in Tricia, this is the case, whereas Tricia call the packages *spaces* and the custom types *type definition* (as introduced in Subsection 2.2.1). In contrast to basic types, custom types may have attributes (also representing relations) as well as derived attributes (in addition to custom functions). While the values of attributes are persisted in Tricia's database, the values of derived attributes are computed based on a MxL expression. For example, a custom type *Employee* may contain two attributes *Salary* and *Hours* as well as a derived attribute *Costs*, which is specified as the product of *Salary* and *Hours*, whereas the definition of a proper derived attribute might look like follows:

`this`.Salary * `this`.Hours

In the definition of derived attributes, the *this* refers to the instance the derived attribute is evaluated for. The use of the previously mentioned “implicit this” allows a shortening of this expression to the following:

Salary * Hours

Custom packages as well as custom types and their attributes and derived attributes share a common characteristic: Their names may contain special characters, e.g., the type *Employee* may have an attribute *Work hours* instead of *Hours*. However, in a MxL expression, special characters may yield to problems, e.g., the identifier of the *Work hours* attribute would be interpreted as two successive identifiers (*Work* and *hours*). Therefore, MxL 2.0 supports the enclosure of identifiers with single quotes, i.e., a valid definition of the derived attribute *Costs* referring to the *Work hours* attribute might look like follows (again, by using an “implicit this”):

Salary * 'Work hours'

Hence, each identifier referring to a custom package, custom type, attribute, or derived attribute, which contains a special character, has to be enclosed with single quotes.

Last but not least, custom packages may also contain static functions. However, they do not differ from static global functions, apart from the fact that they are assigned to a custom package.

3.1.6. Querying the information model

One of MxL’s main purposes is the definition of queries against an underlying information model. The starting point of an query against the information model is very often a sequence of all instances of a certain type, e.g., all instances of type *Employee*. Hence, MxL 2.0 supports the *find* construct, expecting an arbitrary custom type *T* as parameter and returning a sequence of type *Sequence<T>*:

`find`(<any custom type>)

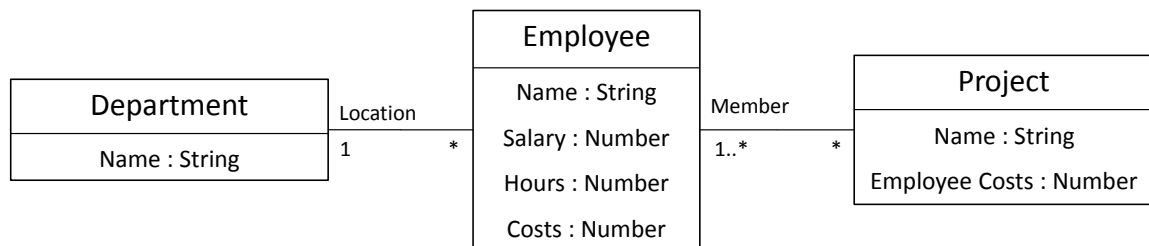


Figure 3.3.: An exemplary UML class diagram [17] consisting of three classes.

Assuming the information model in Figure 3.3, the retrieval of all employees would look like follows:

`find` (Employee)

Based on this sequence, the sequence functions of Subsection 3.1.4 can be used to define certain queries, e.g., the following query would retrieve all employees whose salary is greater than 20:

```
find (Employee)
  .where(e => e.Salary > 20)
```

By applying the previously mentioned “implicit lambda”, this query can be expressed like follows:

```
find (Employee)
  .where(Salary > 20)
```

MxL 2.0 supports the navigation through an information model by using the relations between the types. For example, the following query determines the count of employees working in each project:

```
find (Employee)
  .select(e =>
    {
      ProjectName: e.Name
      DepartmentCount: e.Member.count()
    })
```

Again, this expression can be shortened by the use of an “implicit lambda”:

```
find (Employee)
  .select(
    {
      ProjectName: Name
      DepartmentCount: Member.count()
    })
```

To navigate reversely through the information model, MxL 2.0 supports the *get-whereis* construct:

```
get <related custom type> whereis <reverse relation>
```

For example, the *get-whereis* construct can be used to determine all employees of a certain department, whereas the relation *Location* is used in reverse direction. The following query determines all departments, in which there is at least one employee with a salary greater than 20:

```
find (Department)
  .select(d =>
    d.get Employee whereis Location.any(e => e.Salary > 20))
```

By the “implicit lambda”, this query can be shortened to the following:

```
find (Department)
  .select(
    get Employee whereis Location.any(Salary > 20))
```

3.2. Interpretation

While Section 3.1 discovered MxL 2.0 from the user's perspective, this section will cover rather technical aspects, namely the interpretation and evaluation of MxL 2.0 expressions. Figure 3.4 depicts the process of interpreting and evaluating a MxL 2.0 expression.

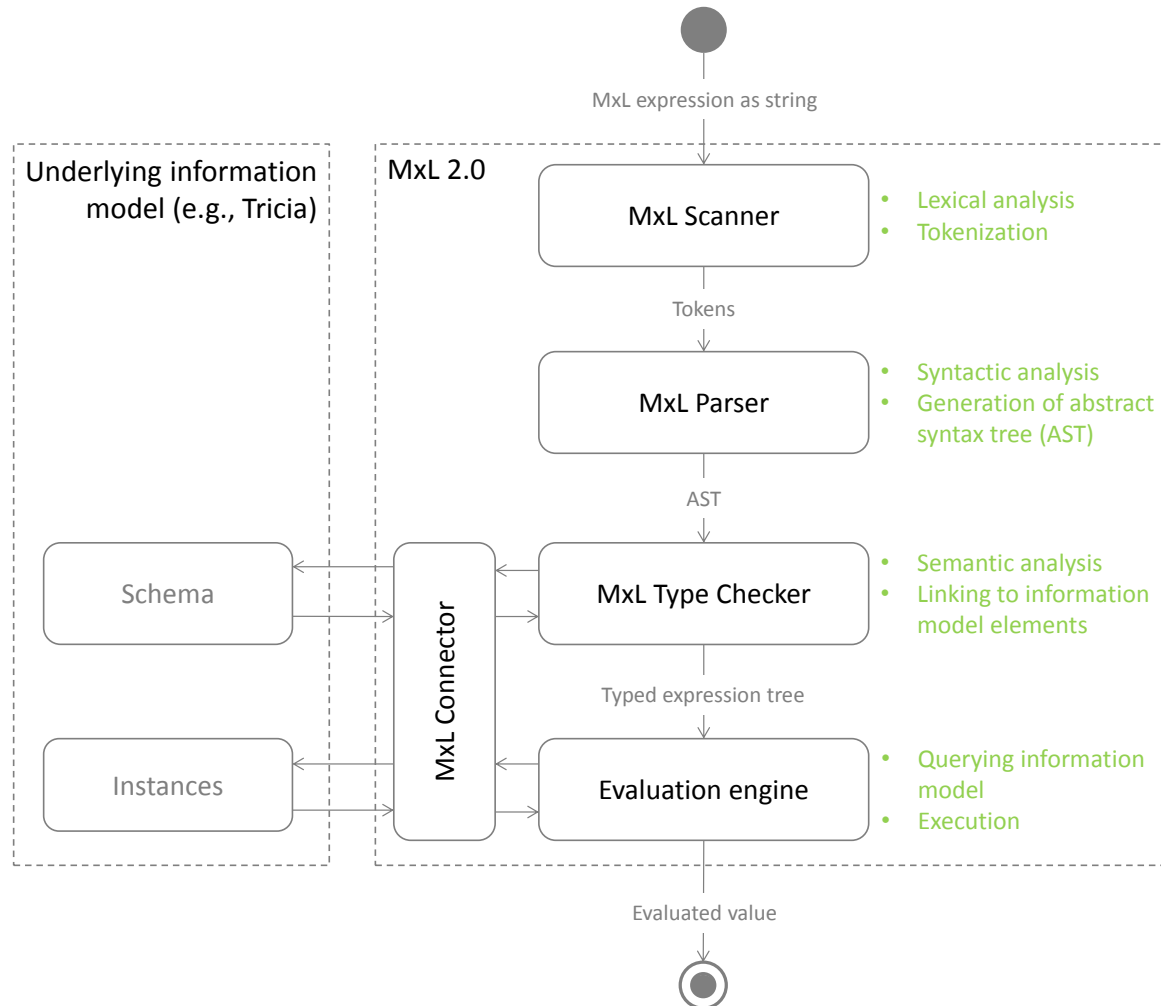


Figure 3.4.: The process of interpreting and evaluating a MxL 2.0 expression. This also shows the interaction between MxL 2.0 and the underlying information model via the MxL Connector.

3.2.1. Scanner & Parser

Apart from some syntax changes (c.f. Subsection 3.1.2), both the MxL 2.0 scanner and the MxL 2.0 parser have not changed that much in comparison to MxL 1.0.

The scanner's input basically is a MxL expression as a stream of characters. A declarative specification of MxL 2.0's lexical grammar defines how to bundle these characters to

proper tokens, e.g., bundling multiple digits to one number, or bundling multiple characters to one identifier. To create the MxL 2.0 scanner, the free Java-based lexical analyzer generator JFlex [27] was used.

These tokens generated by the scanner are the input for the parser, which creates an abstract syntax tree (AST). Similar to the scanner, the MxL 2.0 parser was also generated based on a declarative specification of MxL 2.0's syntax in Extended Backus-Naur Form (EBNF) [28] by the use of the open source LALR [29] parser generator Beaver [30].

Figure 3.5 shows a MxL 2.0 expression and its processing by the MxL 2.0 scanner and parser.

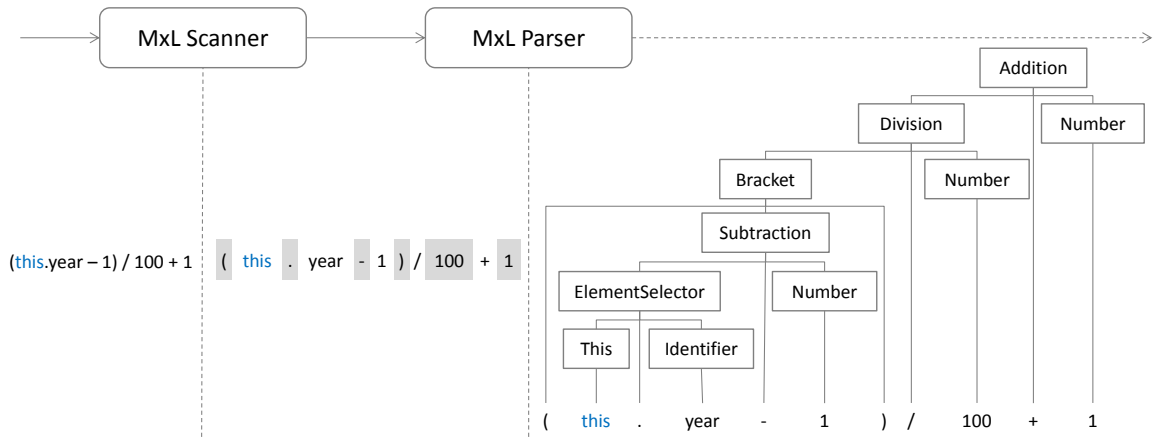


Figure 3.5.: Scanning and parsing of an exemplary MxL 2.0 expression.

3.2.2. AST & Expression objects

The MxL 2.0 AST is implemented as an evaluable expression object containing proper sub-expressions. For example, the addition expression object in Figure 3.5 is the root of the AST and contains a sub-expression for the left operator (a division expression), and a sub-expression for the right operator (a number).

The base of all of MxL 2.0's expression classes is the abstract *Expression* (c.f. Figure 3.6), whereas its most important methods are the following, whereas most of them are abstract and implemented in concrete subclasses:

evaluate This method executes the expression, and evaluates its sub-expression, e.g., the *DivisionExpression* will evaluate its sub-expressions for the left and right operand, and subsequently divides the first value through the second one.

The evaluation is done by the evaluation engine described in Subsection 3.2.5.

checkType This method checks the type of the expression and all its sub-expressions and returns the determined return type (the supposed type of value returned by the evaluate-method), e.g., the *DivisionExpression* will check if the types of its two sub-expressions are numbers and subsequently returns also the type *Number*.

The type checking is done by the type checker described in Subsection 3.2.4.

toJSON To store a type checked expression, an expression object can be serialized to a JSON object. While a serialization of an expression object's original string representation as inputted to the scanner would also be imaginable, an internal representation as JSON object allows the retention of the mapping from the expression's identifiers to the information model's objects determined by the type checking process (c.f. Subsection 3.2.4).

getMxLReferenceProfile After a successful type checking, this method returns a reference profile containing all the MxL assets referenced in the expression and its sub-expressions, e.g., all types, attributes, and functions used in the expression.

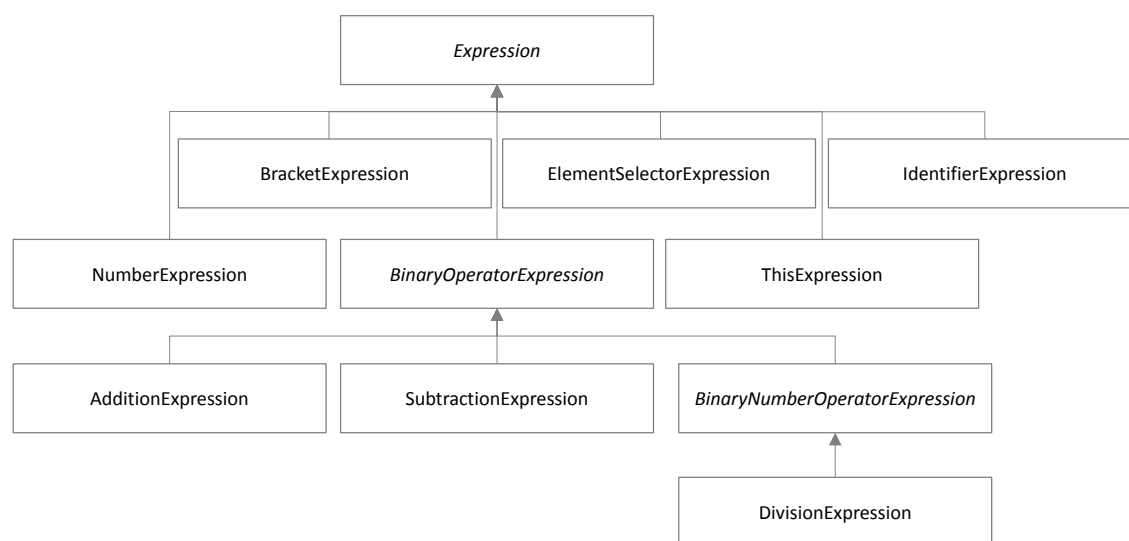


Figure 3.6.: An excerpt of MxL 2.0's expression types containing all the types of the example in Figure 3.5. Since the plus-symbol is both the arithmetic addition and the string concatenation, and the minus-symbol is both the arithmetic subtraction and the date difference operation, the *AdditionExpression* and *SubtractionExpression* classes are not subtypes of the *BinaryNumberOperatorExpression*

In addition to these methods, the *Expression* class also contains two important static functions:

fromJSON This static function generates an expression object by its internal JSON representation

parse This static function generates an expression object by the expression's string representation, i.e., the string is processed by the MxL 2.0 scanner as well as the MxL 2.0 parser as described in Subsection 3.2.1.

Since MxL 1.0 missed a type checker, all these methods except *evaluate* – although even this one has changed a lot – are new in MxL 2.0, whereas they will be explained in more detail in the following Subsections.

3.2.3. MxL Connector

While scanner and parser are rather autonomous components, both the type checker and the evaluation engine have to interact with the underlying information model. However, since MxL 2.0's information model strongly depends on the concrete implementation of MxL 2.0, the MxL 2.0 interpreter consists of a component called "MxL Connector", which abstracts the interaction between MxL and the information model of its concrete implementation. In fact, the connector is the interface between MxL 2.0 and its implementing system carrying out all their interactions. The MxL Connector is a new component in MxL 2.0 and facilitates the implementation of MxL in arbitrary tools.

Basically, a MxL Connector is an abstract class, which has to be implemented in order to connect to a system like Tricia, whereas the connector provides the following methods:

Access to information model The main purpose of the MxL Connector is the abstraction of the access to a concrete information model. Therefore, the connector provides a multitude of methods for gathering an information model's schema data (e.g., get an attribute by its name and owner type) as well as its instances (e.g., get attribute value by its name and owner object).

Mapping of basic types As described in Subsection 3.1.1, MxL 2.0 provides a set of basic types. Consequently, these types have to be supported by each implementation of MxL 2.0, whereas the MxL Connector maps the types of the implementing system to MxL 2.0's types. This mapping is defined by the connector's sub-component *TypeProvider*.

Extensions of global identifiers While MxL 2.0 provides already a (very minimalistic) set of global identifiers (e.g., *Today* returning the current date, c.f. Subsection 3.1.5), the MxL Connector allows the provision of implementation-specific global identifiers. This extensibility is defined by the connector's sub-component *GlobalIdentifierProvider*.

Extensions of basic functions Similar to the global identifiers, MxL 2.0 provides already a set of default basic functions, which also includes the sequence functions from Subsection 3.1.4. However, for certain implementations it might be necessary to extend this set by implementation-specific functions, whereas the MxL Connector supports this extensibility of basic functions. This extensibility is defined by the connector's sub-component *FunctionProvider*.

The MxL Connector and its sub-components are depicted in Figure 3.7.

3.2.4. Type Checker

While the MxL 1.0 interpreter already contains a scanner, parser, as well as an evaluation engine, the type checker, along with the MxL Connector, is new in MxL 2.0. This component of the interpreter takes an AST as input and applies a type check onto the root expression object triggering a cascading type check of the whole AST.

The concrete operation launched by the type checker strongly depends on the type of expression. Trivial expressions (e.g., *NumberExpression*, *StringExpression*, *BooleanExpression*,

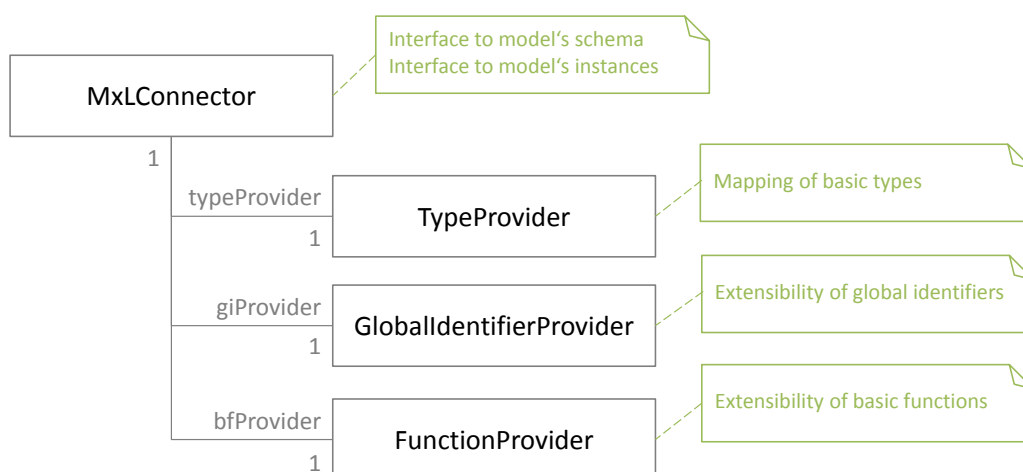


Figure 3.7.: The MxL 2.0 Connector and its sub-components.

NullExpression) do not have to be checked, since their types are fixed, whereas operator expressions (e.g., arithmetic and conditional expressions) are either fixed or purely depending on their operands (e.g., if the operands of the *AdditionExpression* are numbers, the addition itself returns a number too).

However, certain expressions require more effort to determine their return type.

Expressions interacting with the information model

There are basically two expression types interacting with the information model: the *ElementSelectorExpression* (of the form `<context object>.<member>`) and the *IdentifierExpression*.

For example, the derived attribute *Costs* of the type *Employee* from Subsection 3.1.5 was defined as follows (containing two element selectors):

```
this.Salary * this.Hours
```

This expression will be parsed to the AST depicted in Figure 3.8.

An *ElementSelectorExpression* consists of two sub-expressions, namely of a left operand, which establishes the context for the right operand, which has to be an identifier (optionally enclosed by single quotes). Hence, the type checker first determines the type of the left operand and subsequently checks if this type has a member with the name defined by the identifier. The element selector processing steps are done in the following order:

ESPS 1 If the left operand's type is a subtype of *Date*, and the identifier is one of *day*, *month*, or *year*, the return type of the element selector is a number representing the desired part of a date.

ESPS 2 If the left operand's type is a subtype of *Map*, the element selector's return type is *Object* (Since the *Map*-type does not support type parametrization

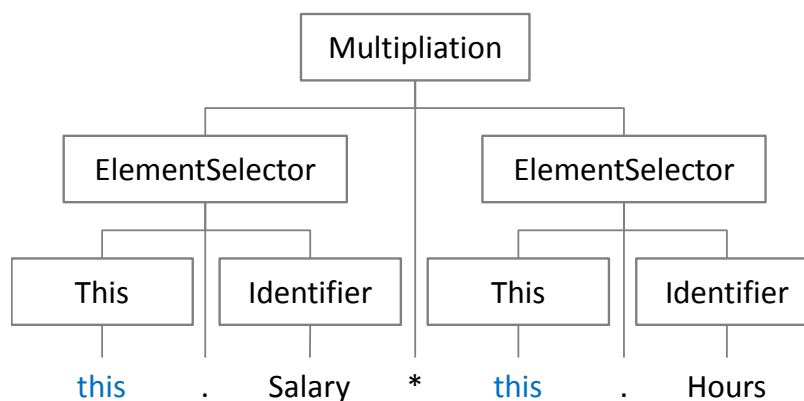


Figure 3.8.: The AST for the expression *this.Salary * this.Hours*

- ESPS 3 If there is a basic function compatible to the left operand's type and having the name defined by the identifier, the element selector's type is the determined basic function's type
- ESPS 4 If there is a custom function compatible to the left operand's type and having the name defined by the identifier, the element selector's type is the determined custom function's type
- ESPS 5 If the left operand's type is a subtype of *Entity*, the type checker obtains this type from the underlying information model and checks for the existence of an attribute or derived attribute with the name defined by the identifier. If yes, the element selector's type is the type of the determined attribute or derived attribute.
- ESPS 6 If the element selector cannot be resolved, a *MxLTypeCheckingException* is fired.

The purpose of this processing order is the resolution of name collisions in case of multiple members with the same name.

The type checking process for the derived attribute *Costs* from above is depicted in Figure 3.9.

As stated in Subsection 3.1.5, the derived attribute *Costs* of the type *Employee* is also definable as follows:

*Salary * Hours*

While this definition is semantically equivalent to the previous one, there are syntactical differences, since the AST generated by the MxL 2.0 parser consists only of a *MultiplicationExpression* whose sub-expressions are two *IdentifierExpressions* instead of *ElementSelectorExpressions*.

The identifier processing steps are done in the following order:

- IPS 1 If there is a name-value-binding with the given name in the current context (e.g., a function's parameter, or a binding created by the *let*-construct), the determined name-value-binding's type is returned
- IPS 2 If there is a global identifier with the given name, its type is returned

IPS 3 If there is a static basic function with the given name, the basic function's type is returned

IPS 4 If there is a static custom function with the given name, the custom function's type is returned

IPS 5 If there is a member of the current *this* object with the given name, this member's type is returned. This is the implementation of the "implicit this" concept introduced in Subsection 3.1.5, which is done by executing the type checker of the element selector, whereas the object identified by *this* is used as the context object.

IPS 6 If the identifier cannot be resolved, a `MxLTypeCheckingException` is fired.

For example, by type checking the derived attribute *Costs*, processing step IPS 5 with *Employee* as the type of the "implicit this" is applied, which basically yields to a very similar process as depicted in Figure 3.9.

Each time the type checker obtains a certain information model element (whether by an *ElementSelectorExpression* or an *IdentifierExpression*), it does not only determine its type, but links the referring identifier of the expression to its corresponding information element by remembering the information model element's ID. Hence, a renaming or other changes of the information model element does not affect the integrity of the MxL expression, since the information model element is still obtainable by the typed expression.

Since an expression is stored by its internal JSON representation, which also serializes the IDs of the information model elements, this linking is also retained on the expression's storage.

For example, the type checking process depicted in Figure 3.9 stores the IDs of both the *Salary* and *Hours* attribute in the corresponding *ElementSelectorExpression*. Hence in MxL 2.0's implementation in Tricia – TxL 2.0 – the type checker determines the IDs of the corresponding *PropertyDefinitions* and assigns them to the expression object, supporting the serialization and storage of the expression object.

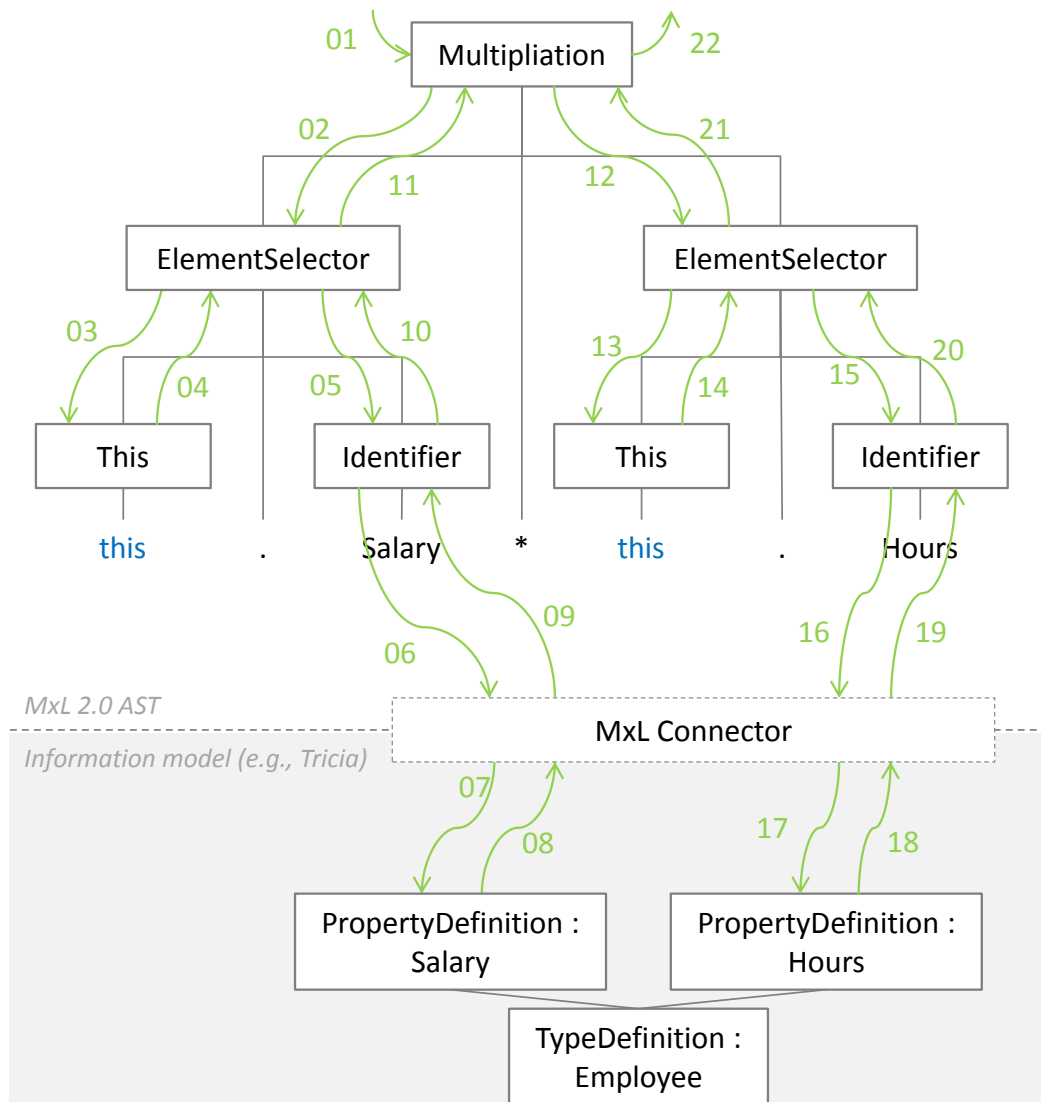


Figure 3.9.: The type checking process for the derived attribute *Costs* is initiated by checking the root of the AST which is the *MultiplicationExpression* (step 01). Checking the left operand of the multiplication causes the determination of the *this* object's type (steps 02-04), which is *Employee*, since this is the derived attribute's owner. Subsequently, the type checker continues with the element selector's processing steps (05), whereas in this example the identifier *Salary* refers to an attribute of type *Employee*. This attribute is obtained via the MxL Connector introduced in Subsection 3.2.3 (steps 06-09). The type of the *Salary* attribute is returned to the *MultiplicationExpression* (steps 10-11), which does the same process again for the right operand (steps 12-21), and finally returns the type *Number* as result of the *MultiplicationExpression* and type checking process as a whole (step 22).

Function application expression

While checking the type of an expression for the trivial application of a function would be rather simple, the following two features of MxL 2.0 are challenging for the type checking process:

- Although MxL 2.0 does not support the implementation of generic functions by the use of type parameters, it provides a mechanism to define dependencies between a function's actual owner type, its actual parameter types and its actual return type. These mechanism is implemented by the two components *FunctionParameterTypeChecker* and *FunctionReturnTypeInferer*.

For example, the *select*-function was introduced in Table 3.2 as a function with owner type *Sequence*<*T*>, parameter type *Function*<*T*,*V*>, and return type *Sequence*<*V*>. Hence, the parameter type depends on the actual owner type, while the return type depends on the actual parameter type. Since MxL 2.0 does not support type parameters in the implementation of functions, the *select* is implemented rather general as function with owner type *Sequence*, parameter type *Function*<*Any*,*Object*>, and return type *Sequence*, whereas *Any* is a pseudo-type, which is a subtype of all other types (also known as *bottom*, since on the bottom of the type hierarchy [31]). This pseudo-type is important for the definition of parameter types, because of the contravariance of function types [31]. However, the *FunctionParameterTypeChecker* is able to concretize the parameter types based on an actual owner type, while the *FunctionReturnTypeInferer* can infer an actual return type based on an actual owner type and actual parameter types.

Both the *FunctionParameterTypeChecker* and the *FunctionReturnTypeInferer* are provided by the the function which has to be executed, i.e., each function may implement arbitrary dependencies between its owner type, parameter types, and return type.

The processing steps of the function application are as follows:

FAPS 1 Determine the executing function's type

FAPS 2 Retrieve and execute the *FunctionParameterTypeChecker*, which determines the actual parameter types *FunctionReturnTypeInferer* based on the actual owner type of the function

FAPS 3 Retrieve and execute the *FunctionReturnTypeInferer*, which returns the actual return type of the function based on the actual owner type and the actual parameter types

For example, the following expression increments each of the sequence's numbers:

```
[1,2,3].select(n => n + 1)
```

While the defined owner type of the *select*-function is *Sequence*, its actual owner type in this example is *Sequence*<*Number*>. Based on this actual owner type, the *FunctionParameterTypeChecker* infers that the parameter type has to be at least of type *Function*<*Number*,*Object*>. However, by inferring the lambdas return type, *Function*<*Number*,*Number*> can be observed as the actual parameter type of the *select*-function. Subsequently, the *FunctionReturnTypeInferer* infers the return type of the *FunctionApplicationExpression*, which is *Function*<*Number*>, since the return type of the lambda is also *Number*.

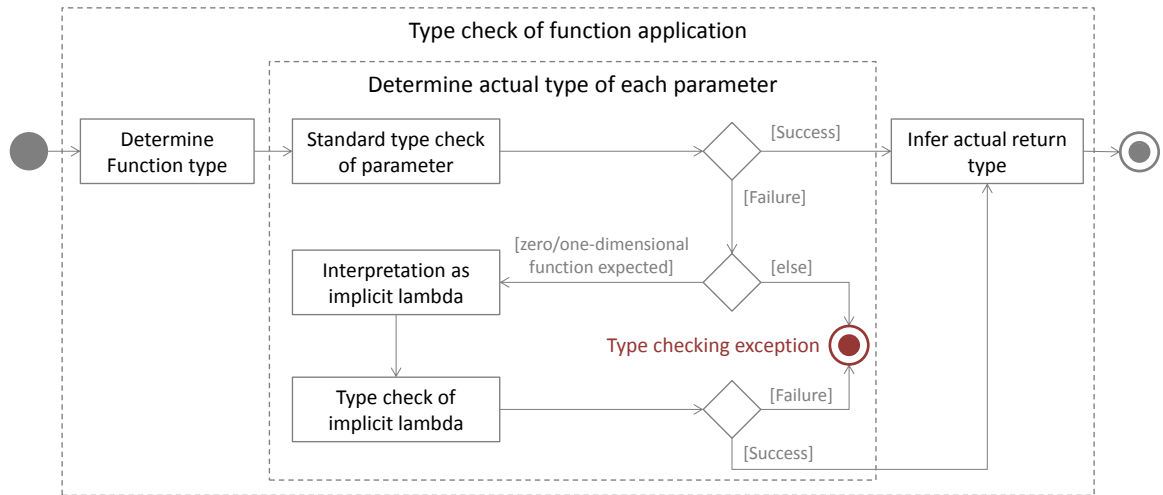


Figure 3.10.: Type checking process of the function application expression depicted as UML activity diagram [17]. This expression uses the default *FunctionParameterTypeChecker*, which implements the “implicit lambda” feature introduced in Subsection 3.1.3.

- As stated in Subsection 3.1.3, MxL 2.0 supports “implicit lambdas”. Hence, if the expected type of a parameter is either a zero-dimensional or one-dimensional function (function with either no or one parameter), the type checker determines the actual type of this parameter. If the actual parameter type is not conform to the expected one, the type checker interprets the parameter expression as the method stub of a lambda expression and rechecks the parameter’s type. This functionality is implemented by the default *FunctionParameterTypeChecker* as shown in Figure 3.10.

Multiple purpose expressions

MxL 2.0 consists of several expression types, which first have to be analyzed to observe their semantic. For example, the *AdditionExpression* is both the arithmetic addition and the string concatenation.

However, the semantic of a multiple purpose expressions can already be determined at compile time. Therefore, a check of an expression’s semantic at runtime is neither necessary nor efficient. To set the concrete operation an expression has to perform at runtime, the type checker has to set the so-called *Executor* of the multiple purpose expression, which implements this operation.

For example, the *AdditionExpression* provides an abstract *AdditionExecutor* as well as two concrete implementations *ArithmeticAdditionExecutor* (performing an arithmetic addition at runtime) and *StringConcatenationExecutor* (performing a string concatenation at runtime), as shown in Figure 3.11. The type checker determines the type of the plus-operator’s operands, and instantiates either an *ArithmeticAdditionExecutor* (if both operands are numbers) or *StringConcatenationExecutor* (if the first operand is a string). At runtime, the evaluation engine calls the *execute*-function of the *Executor* without checking its semantics.

This executor pattern (a variation of the prevalent prototype pattern [32]) is also applied on the *SubtractionExpression* (may be arithmetic subtraction or a date difference operation), certain comparison expressions (either numerical or date comparisons), and the previous mentioned *ElementSelectorExpression* and *IdentifierSelection*, whereas the executors of *ElementSelectorExpression* and *IdentifierSelection* are determined by the mentioned processing steps. For example, the executor for both *ElementSelectorExpressions* in Figure 3.9 is set to *AttributeExecutor*, whereas this executor is parametrized with the ID of the corresponding attribute.

Since the executor is included in the serialization of the expression object, the assignment of the concrete executor implementation is also stored and hence retained through the serialization-deserialization cycle of the expression object.

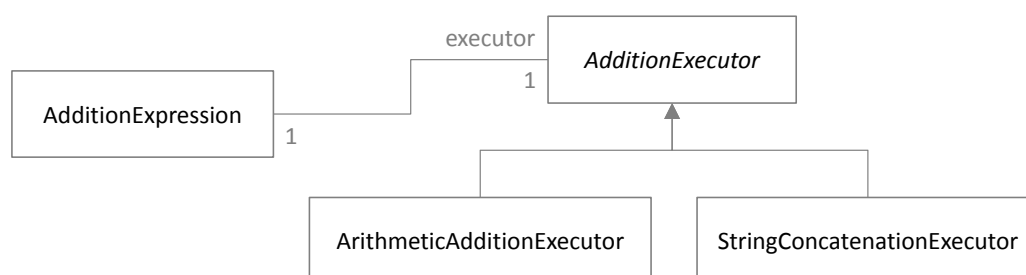


Figure 3.11.: UML class diagram [17] showing the *AdditionExpression* and its executor types

3.2.5. Evaluation engine

Although MxL 1.0 already had an evaluation engine, it does not have much in common with the evaluation engine of MxL 2.0, since in MxL 2.0, type checking is already done at runtime by the type checker, whereas MxL 1.0 was a dynamically typed language. Furthermore, expressions with potentially multiple semantics are already analyzed at runtime, i.e., the operation which has to be performed is already specified by the type checker.

Basically, the process of evaluating an AST is very similar to the process of an AST's type checking, i.e., the evaluation is initiated at the AST's root and cascaded through the whole tree. For example, the evaluation process for the previously defined derived attribute *Costs*, whose AST is depicted in Figure 3.8, nearly looks like the type checking process in Figure 3.9, except that the evaluation engine processes instance data instead of schematic information.

Hence, due to the extensive preliminary work by the type checker, evaluating an expression is rather simple. Most expressions are either directly expressible by proper Java statements (e.g., arithmetic and logical operations) or evaluable by executing the executor as mentioned in the previous subsection (e.g., element selectors and identifiers, eventually triggering a query of the information model through the MxL Connector).

Stack trace

In order to identify potential infinite loops as well as to localize *MxLEvaluationExceptions*, the evaluation engine manages a stack trace.

Each time the evaluation engine executes a function or derived attribute, it pushes the function's or derived attribute's identifier onto a call stack (inclusively some context parameters, e.g., the object a function is applied on). If the execution of the function or derived attribute is completed, the evaluation engine takes off the call stack's upper element. The stack trace can be understood as a snapshot of the current call stack.

Therefore, if an `MxLEvaluationException` occurs while evaluating an expression, the stack trace represents a path to the source of the `MxLEvaluationException`. Moreover, since the call stack also contains some context parameters, the evaluation engine is able to check if a function or derived attribute, which has to be evaluated, is already evaluating with the current parameters, which would indicate an infinite loop. In this case, the evaluation engine stops the evaluation and throws a proper `MxLEvaluationException`.

3.3. TxL 2.0 - MxL 2.0's implementation in Tricia

While Sections 3.1 and 3.2 described the implementation-independent aspects, this section focuses on a certain implementation of MxL 2.0, namely its implementation in Tricia called TxL 2.0.

3.3.1. MxL Connector for Tricia

As described in Subsection 3.2.3, one of MxL 2.0's components – the MxL 2.0 Connector – manages the interaction between MxL 2.0 and a concrete information model. Hence, in order to allow MxL 2.0 to access Tricia's information model, a MxL Connector for Tricia has to be implemented.

However, in order to successfully implement MxL 2.0, Tricia has to meet the following requirements:

- All the information model elements, which have to be accessed via MxL 2.0, as well as all MxL assets as depicted in Figure 3.2 must implement certain MxL 2.0 interfaces. For example, Tricia's `TypeDefinition` (c.f. Subsection 2.2.1) has to implement the MxL 2.0 interface `MxLType`, the `PropertyDefinition` has to implement the interface `MxLAttribute`, the Tricia class `Page` has to implement the interface `MxLEntity`, etc.
- Since MxL 2.0 requires the MxL Core Package (c.f. Subsection 3.1.5) as container for all basic types, basic functions as well as global identifiers, Tricia has to ensure the existence of a corresponding MxL Core Space (Tricia's counterpart of packages)
- Tricia has to provide all basic types required by MxL 2.0 (c.f. Subsection 3.1.1). For this purpose, Tricia has to create all these basic types on the platform's initialization. Moreover, Tricia extends this set of basic types by the tool-specific types `Page`, `Document`, `Principal`, `Person`, and `Group` as depicted in Figure 3.1.

Fulfilling these requirements enables the implementation of a MxL Connector for Tricia, which abstracts the access to Tricia's information model and maps the MxL basic types to Tricia's counter parts.

Although the MxL Connector would support the extension of the set of basic functions as well as the extension of the set of global identifiers, the Tricia connector does not use this

feature.

As stated in Subsection 3.1.6, MxL 2.0 supports a *find* construct returning all instances of a certain type. However, this construct just forwards this command to the MxL Connector, so that each implementing tool has to process the query in its own specific way. Tricia determines all pages of a certain type by its integrated search engine elasticsearch [33].

3.3.2. Basic MxL infrastructure in Tricia

As described in Subsection 3.1.5, MxL 2.0 consists of several assets arranged in a certain structure called the MxL 2.0 asset hierarchy (c.f. Figure 3.2). The root of the hierarchy is the MxL Environment, whereas in TxL 2.0 this environment is represented by Tricia.

A central access point to MxL 2.0 in Tricia is the MxL Core Space, which is not only the container for MxL 2.0's basic assets, but also provides a proper user interface (UI) for its main assets:

MxL Core Space - Homepage The homepage of the MxL Core Space is accessible via a button presented by a magic wand in Tricia's navigation bar at the top. It shows some statistics about the usage of certain MxL assets, i.e., the page consists of a list of types breaking down the number of basic functions, custom functions, and derived attributes, which are defined per type. Types without these MxL assets are omitted in this view.

Basic Types This view lists all basic types of MxL 2.0 as well as TxL 2.0's basic types (c.f. Figure 3.1).

Global Static Functions As stated in Subsection 3.1.5, the MxL Core Package may contain global static functions, whereas the term *global* means, that the function is assigned to the MxL Core Package and accessible in each package. The set of default global static functions provided by default by MxL 2.0 are the date parse function *date*, the exponent function *exp*, and the natural logarithm function *ln*. However, this view also contains a button for the creation of new global static custom functions in order to extend this default set (c.f. Subsection 3.3.3).

Global identifiers Again, this is an overview for one of MxL 2.0's assets, namely its global identifiers. Since TxL 2.0 does not extend MxL 2.0's default set of global identifiers, this page will show only the *Today* identifier returning the current date as well as an identifier representing the mathematical constant *Pi*.

Expression Test In order to test some expression as well as to evaluate its performance, this view offers a UI to submit an arbitrary MxL 2.0 expression and shows the results and duration of its interpretation and evaluation.

Invalid MxL Providers MxL providers are assets defined by at least one MxL expression, e.g., custom functions and derived attributes (c.f. also Subsection 3.3.5). However, since Tricia allows the embedding expressions into a page's rich-text content, pages are also potential MxL providers (c.f. Subsection 3.3.4).

While on creation of an MxL expression its integrity is checked, this integrity can be violated over time, e.g., by deleting certain types or properties.

Hence, the view “Invalid MxL Providers” shows all providers, whose integrity is currently violated. If there are no such providers, this view is hidden.

While the MxL Core Space covers MxL 2.0’s basic assets, the custom packages in Tricia are represented by work spaces. Hence, each work space provides a view “Static Functions” listing all its static functions and allowing the definition of new ones. While the “Static Functions” view was created as part of the implementation of TxL 2.0, the view “Types” is a default view of Tricia. However, the UI of individual types was extended by the following features:

Super type and (direct) sub types While Tricia does not support inheritance, MxL 2.0 does. Therefore, the super type of each custom type definition is TxL 2.0’s basic type *Page* by default (if the type definition applies to pages). However, a type’s super type as well as its (direct) sub types are displayed on the type’s settings view.

Derived attributes As previously mentioned, one of MxL 2.0’s main assets are derived attributes, whose values are not persisted, but computed based on a MxL 2.0 expression. Therefore, a type’s settings page not only lists its common attributes, but also the derived attributes (c.f. Subsection 3.3.3).

Functions As depicted in the MxL Asset Hierarchy in Figure 3.2, a custom type not only consists of attributes and derived attributes, but also of basic and custom functions, which are listed in the view “Functions” of each type, i.e., the list contains all functions whose owner type is the selected one. Moreover, this view also allows the definition of new functions for the selected type (c.f. Subsection 3.3.3).

3.3.3. Derived attributes and custom functions

While spaces, type definitions, and property definitions are default entities in Tricia and able to implement MxL 2.0’s packages, types, and attributes, originally there are no corresponding Tricia entities for derived attributes and custom functions. As a consequence, these two entities were implemented as *DerivedHybridProperties* and *CustomFunctions* as depicted in Figure 3.12. Since global identifiers and basic functions are not definable at runtime, they are not implemented as persistable Tricia entities and hence are not included in this Figure.

As mentioned in the previous section, derived attributes are managed on the settings page of the derived attribute’s owner type. By selecting a derived attribute (c.f. Figure 3.13), its associated MxL expression as well as an inferred return type are shown. Furthermore, so-called incoming and outgoing “MxL References” are listed, which are described in more detail in Subsection 3.3.5. New derived attributes can be created by clicking the “New Derived Attribute” button on the settings view of a type, while there are also hyperlinks for editing and deleting existing ones.

The only properties of derived attributes are its name, a short description, and its definition represented by a MxL expression. While both the name and the description are simple strings, the expression property is of a type *MxLProperty*, which provides the following features:

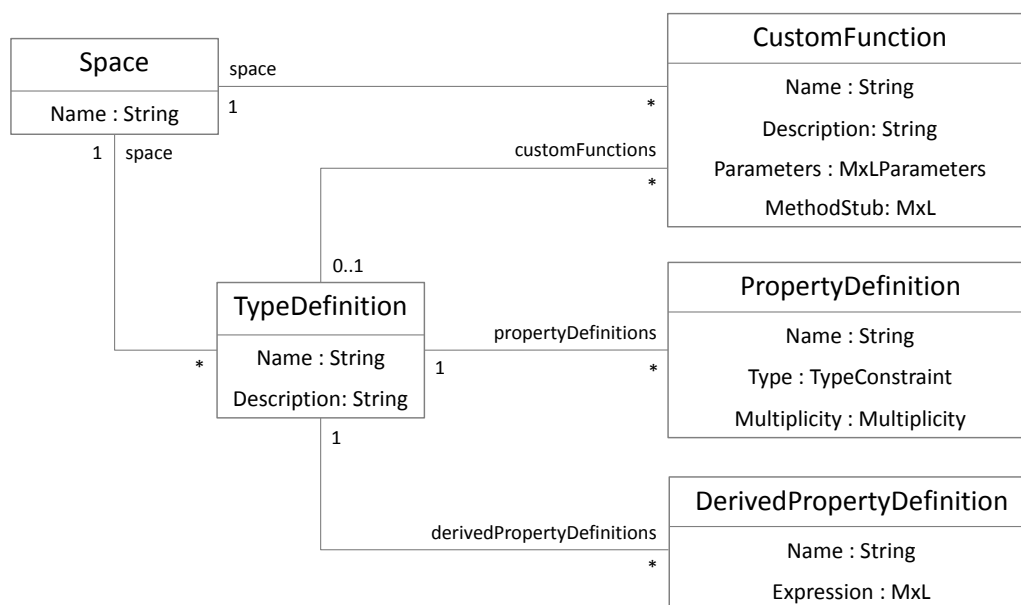


Figure 3.12.: UML class diagram [17] showing Tricia's default entities *Space*, *TypeDefinition*, and *PropertyDefinition* as well as TxL's new entities *DerivedPropertyDefinition* and *CustomFunction*.

Editor When viewing and editing a *MxLProperty*, the in-browser code editor CodeMirror¹ is used in order to provide a proper UI for MxL expressions. This code editor supports syntax highlighting as well as auto completion (c.f. Figure 3.14).

Testing In order to test a defined expression by evaluating it, the UI of the *MxLProperty* offers a "Try expression" button, whereas the *this* object as well as eventual parameter values are instantiated with exemplary data.

Validation On creating and updating an expression, an automated validation of the expression is triggered, i.e., Tricia tries to interpret (including type checking) the expression. If the interpretation is not successful (i.e., if the parser or the type checker throws an *MxLTypeCheckingException*), the function cannot be created or updated. However, in case of an error, a proper error message is displayed in order to support the user to solve the problem.

Analysis As mentioned later on in Subsection 3.3.5, MxL expressions can be analyzed in order to determine outgoing MxL references, whereas the *MxLProperty* provides some useful methods for this purpose, e.g., a method for updating all MxL references of an *MxLProperty*'s expression.

In contrast to derived attributes, a type's custom functions (as well as all basic functions) are listed on a separate view of this type ("Functions"). Again, this view provides a button for the creation of new custom functions as well as hyperlinks for editing and deleting existing ones. By selecting a custom function (c.f. Figure 3.15), its name, description, pa-

¹<http://codemirror.net>

Attributes of type Employee

Attribute Name	Frequency	Attribute Type	Default Values	Multiplicity	Help for Editors	Action
Hours	2/2 (100 %)	Number	n.a.	Exactly one value		Edit Delete
Location	2/2 (100 %)	Reference Department	n.a.	Exactly one value		Edit Delete
Salary	2/2 (100 %)	Number	n.a.	Exactly one value		Edit Delete

Derived Attributes of type Employee

Derived Attribute Name	Help for Editors	Action
Costs		Edit Delete

Derived Attribute Costs of Type Employee

Name: Costs

Help for Editors: Return Type: Number

Expression: Salary * Hours

Incoming MxL References

Custom Functions	Derived Property Definitions
Department::employeesByCosts	Project::Employee Costs

Outgoing MxL References

Property Definitions
Employee::Hours
Employee::Salary

Figure 3.13.: The settings view for the type *Employee* of Figure 3.3 showing its attributes and derived attributes. By selecting a derived attribute, the user will be navigated to the derived attribute’s definition. In this example, the derived attribute *Costs* is defined based on an employee’s attribute *Salary* and *Hours*, whereas an “implicit this” is used to shorten the expression.

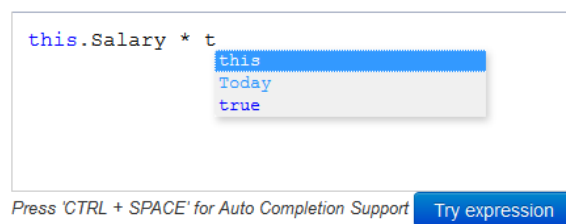


Figure 3.14.: The in-browser code editor for editing *MxLProperties*. This code editor supports syntax highlighting as well as auto completion

rameters and method stub are shown, as well as a inferred return type and the custom function’s MxL References (c.f. Subsection 3.3.5). A function’s name and description are strings, whereas the method stub is again implemented as *MxLProperty*. The type of the parameter property is *MxLParametersProperty*. In contrast to *MxLProperty*, a *MxLParametersProperty* does not allow arbitrary MxL expressions, but just the definition of typed parameters.

3.3.4. Embedded expressions

While the previous subsection covered MxL 2.0’s usage for the formal definition of MxL assets (derived attributes and custom functions), its implementation in Tricia supports yet another use case, namely the embedding of a MxL 2.0 expression in a page’s rich-text content.

In general, rich-text properties (like the content of a wiki page) support so-called *BlockSub-*

Custom Functions of type **Department**

[+ Define a new Custom Function](#)

Name	Parameters	# Used by	Description	Action
employeesByCosts	minCosts: Number maxCosts: Number ?	0	Returns all employees of the department whose Costs is in the given range	Edit Delete

Custom MxL Function Department::employeesByCosts

Name: employeesByCosts

Description: Returns all employees of the department whose Costs is in the given range

Parameters: minCosts: Number
maxCosts: Number ?

Return Type: Sequence < Employee >

Method Stub

```
get Employee whereis Location
  .where(Costs >= minCosts and
    if maxCosts = null then true else Costs <= maxCosts)
```

Outgoing MxL References

Basic Functions

Sequence::where

Derived Property Definitions

Employee::Costs

Property Definitions

Employee::Location

Types

Employee

Number

Figure 3.15.: The functions view for the type *Department* of Figure 3.3 showing all custom functions whose owner type is *Department*. By selecting a custom function, the user will be navigated to the custom function's definition. In this example, the custom function *employeesByCosts* returns all employees of a department, whose *Costs* (the derived attribute from Figure 3.13) is in the given range. The range can be set by two parameters of type *Number*, whereas the second one is optional. Hence, both *anyDep.employeesByCosts(200)* and *anyDep.employeesByCosts(200,300)* are valid applications of the function.

stitutions, which are executable blocks inside the HTML markup defining the properties value. In edit mode of the property, a block's definition is shown, while in view mode Tricia executes the block and substitutes its definition by the result of the execution.

By implementing MxL 2.0 in Tricia, this *BlockSubstitution* feature was extended by the *eval* block. This allows the embedding of MxL expressions as blocks (by serializing them to their internal JSON representation), whereas an *eval* block's execution is the evaluation of the MxL expression.

For example, a content of a page might be defined as follows (for the sake of clarity, the internal JSON representation is reduced to its basic parts):

```
<p>1 + 2 = <strong>${eval()}$
{ "leftOperand": { "value":1, "expressiontype":"Number" },
  "expressiontype":"Addition", "executor":"number",
  "rightOperand": { "value":2, "expressiontype":"Number" } }
$eval]${</strong></p>
```

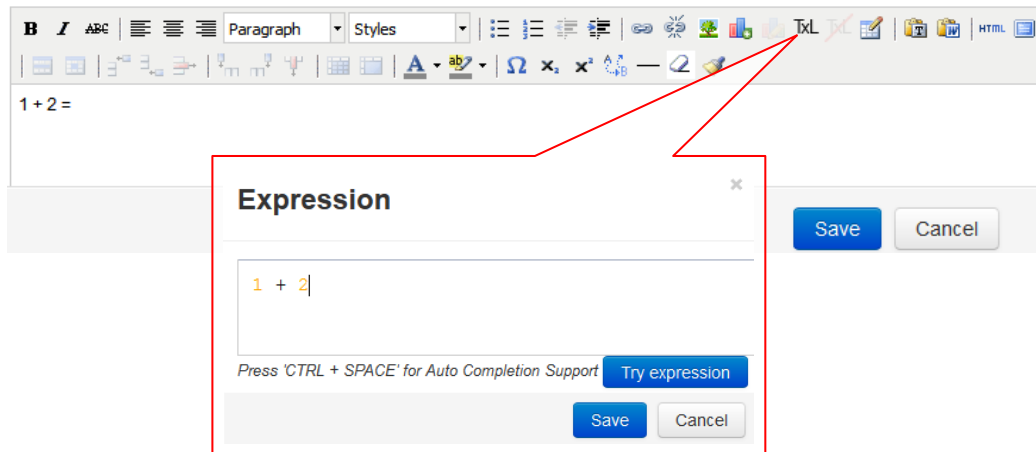


Figure 3.16.: The rich-text editor of Tricia provides a button opening a MxL code editor.

In view mode of the property, Tricia executes the *eval* block by evaluating the MxL 2.0 expression, whereas the result of the evaluation would be the following:

```
<p>1 + 2 = <b>3</b></p>
```

In order to embed a MxL expression, Tricia's rich-text editor (based on the JavaScript WYSIWYG editor TinyMCE [34]) provides a button, which opens a modal dialog containing a MxL code editor (c.f. Figure 3.16). By clicking on "Save", the entered expression will be validated by running the type checker, serialized to its internal JSON representation, and embedded in the HTML markup with a surrounding *eval* block.

The page the content property is part of is accessible via the *this* keyword. Hence, when editing a page of type *Employee* (as defined in Figure 3.3), an embedded expression can access its properties by *this.Salary* or *this.Costs*, enabling the definition of dynamic HTML-based visualizations based on the information model's data. For example, the following expression defines a visualization showing either a green or a red traffic light, dependent on the value of the current employee's *Costs*:

```
if this.Costs > 30
  then "<img src='redlight.jpg'/>"
  else "<img src='greenlight.jpg'/>"
```

More advanced visualizations are subject in Section 6.5.

3.3.5. Compile-time analysis of MxL expressions in Tricia

The main motivation for the development of MxL 2.0 was the capability of a compile-time analysis of a MxL expression in order to implement the Living KPIs, whereas the analysis of an expression mainly consists of the determination of all MxL assets the expression's identifiers refer to.

While MxL 2.0 expressions are already analyzable, Tricia has to provide the infrastructure to manage the references between the MxL assets in order to allow an efficient determination of all MxL providers (assets defined by at least one MxL expression, e.g., custom

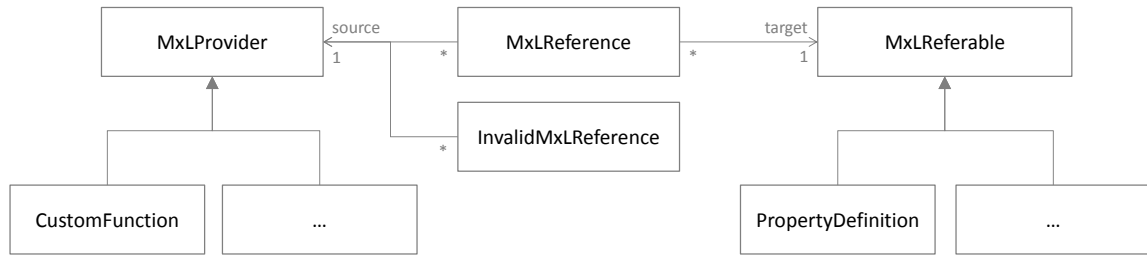


Figure 3.17.: Tricia's conceptual model of the MxL reference infrastructure.

functions or derived attributes) affected by a certain action, e.g., renaming of a property. The conceptual model of the MxL reference infrastructure implemented in Tricia is depicted in Figure 3.17. As already explained, MxL providers are assets defined by MxL expressions, which may refer to other MxL assets through identifiers. In contrast, MxL referables are assets, which can be accessed in MxL expressions by identifiers or element selectors, i.e., *Spaces* (spaces), *Types*, *Basic functions*, *Custom Functions*, *Global identifiers*, *property definitions* (attributes), as well as *Derived attributes* (derived property definitions).

However, since persistable Tricia entities (e.g., *CustomFunction* and *PropertyDefinition*) are already derived from the Tricia class *PersistentEntity*, and Java does not support multiple inheritance, Tricia uses another code reusability pattern, namely *Mixins* [35].

Since MxL references are persistable objects, the determination of all MxL providers referring to a certain MxL referable is rather simple under the requirement of a proper management of these MxL references. Hence, on each creation and update of MxL providers and MxL referables, Tricia checks and eventually rearranges involved MxL references, and creates an *InvalidMxLReference* in case of a failed type check of a MxL provider.

For example, the exemplary information model of Figure 3.13 would yield to the MxL references depicted in Figure 3.18. The MxL reference from the custom function *employeesByCosts* to the type *Number* shows, that not only a custom function's method stub is affected by its analysis, but also its parameter definitions.

So if the attribute *Hours* of type *Employee* is renamed, Tricia determines all its incoming MxL references (from the derived attribute *Costs*) and updates the corresponding identifiers referring to this attribute. Hence, the integrity of the MxL environment is preserved. If the attribute *Hours* is deleted, Tricia allows the user to choose from the following two options:

Delete all MxL providers referring to this attribute However, since the deletion of the derived attribute *Costs* would make the derived attribute *Employee Costs* inconsistent, all MxL providers transitively referring to the attribute are deleted.

Do nothing This option creates an *InvalidMxLReference* for each MxL provider referring to the attribute *Hours*, marking the MxL provider as inconsistent. As mentioned in Subsection 3.3.2, all inconsistent MxL providers are listed in the view *Invalid MxL Providers* of the MxL Core Space.

Due to the capability of embedding MxL expressions into a page's rich-text content, pages are also possible MxL providers. Hence, the deletion of an MxL expression does not delete

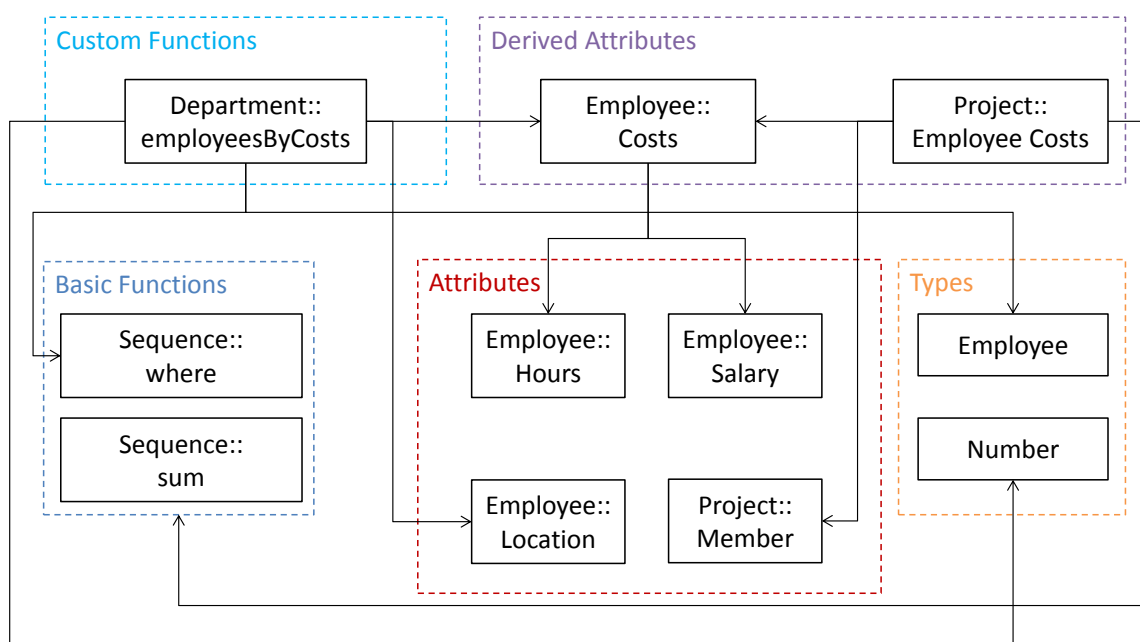


Figure 3.18.: MxL references for the exemplary information model of Figure 3.3 with the derived attributes *Costs* (as defined in Figure 3.13 and *Employee Costs* (as sum of all *Costs* of the project’s members) as well as the custom function *employees-ByCosts* (as defined in Figure 3.15).

the entire page, but just the *eval* block containing the expression. Alternatively, the block can also be replaced by the latest evaluated value of the MxL expression.

As depicted in Figures 3.13 and 3.15, the incoming and outgoing MxL references are shown on each MxL asset’s view, whereas they are grouped by their type. More advanced visualizations of these MxL references are discussed in Section 6.5.

4. Towards Living KPIs

The development of MxL 2.0 equips Tricia with the basic functionality required for the implementation of the Living KPIs by providing an analyzable DSL capable of defining and adapting the KPIs of the EAM KPI Catalog to the organization-specific context.

However, the EAM KPI structure has to be implemented in Tricia in order to project the EAM KPI Catalog's experience to an integrated EAM environment. This includes the EAM KPI structure on the one hand, and the catalog's instances – the KPIs – on the other hand. Hence, Section 4.1 covers the development of a type-based template engine, while the deployment of Tricia applications is subject in Section 4.2. Section 4.3 puts it all together and describes a prototype of the Living KPIs.

4.1. Implementation of a type-based template engine

As already stated in Section 1.3, applying EAM KPI structure as defined by Matthes et al. [12] to Tricia facilitates the familiarization with the Living KPIs. In this context, a page's layout defines the positions and appearance of the page's attributes, whereas a template is the prototype for a page's layout.

Since the catalog's KPI descriptions are implemented as `TypeDefinitions` with the GSEs from Subsection 2.1.1 as well as the OSSEs from Subsection 2.1.2 as its `PropertyDefinitions`, Tricia has to provide a mechanism to apply a certain layout to all instances of a type, otherwise the problems mentioned in Subsection 2.4.3 can occur.

4.1.1. Existing template engine

Although Tricia already supports a HTML-based template engine, there is just one template affecting all pages without taking into account the type definition, which is assigned to them. Moreover, just the appearance of the so-called "built-in" attributes is affectable by the page template, which includes the page's name as well as the page's rich-text content, but excludes all type-specific attributes (e.g., the attributes *Salary* and *Hours* of type *Employee* in Figure 3.3). Since the page's content is implemented as a rich-text property, there are no restrictions regarding the page's layout and design.

4.1.2. Type-based template engine

In order to support type-based templates, Tricia has to be extended to define a template for each of the information model's types in order to define the layout of their instances. Furthermore, since the template is HTML based, the definition of type-based Cascading Style Sheet (CSS) classes allows a central organization of the design of the type's instances. Hence, by implementing a type-based template engine in Tricia, the type definition is extended by the two properties *PageTemplate* and *PageTemplateCSS* defining the appearance

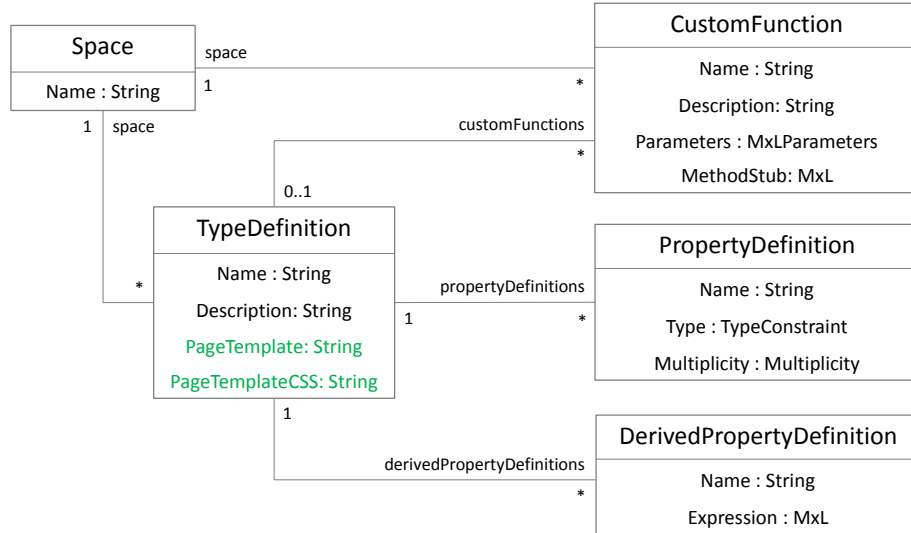


Figure 4.1.: Extension of Tricia's *TypeDefinition* by the properties *PageTemplate* and *PageTemplateCSS*.

of pages which are assigned to this type (c.f. Figure 4.1).

For the definition of the template and the CSS classes a new view "Page Template" for type definitions is available. Initially, the template and the CSS classes are empty, as shown in Figure 4.2. Since the template is internally managed as rich-text property, its editing is done via Tricia's TinyMCE-based rich-text editor. However, in contrast to rich-text contents of page instances, page templates allow the embedding of properties, i.e., they position the corresponding type's attributes. For this purpose, the rich-text editor provides a button for the embedding of properties for the definition of page templates. This button opens a modal dialog containing a list of all the current type's attributes (built-in, regular, and derived attributes). An additional drop-down list allows to select, if either the attribute's name, value, or type has to be inserted in the template (c.f. Figure 4.3). By clicking the save-button, the selected options are serialized to a JSON-object of the following form:

```
{propertyFullIdentifier: "<builtin | regular | derived>propertyid",
propertyDisplayOption: "<Value | Name | Type>"}
```

This JSON-object is embedded in the page template surrounded by a *prop* block (*BlockSubstitutions* were already explained in Subsection 3.3.4). For example, to embed the value of property *Salary* in *Employee*'s page template, the following block has to be inserted (assuming the ID of the attribute is "employeesalary"):

```
$[prop()$ {"propertyFullIdentifier": "regularemployeesalary",
"propertyDisplayOption": "Value"} $prop]$
```

In addition to properties, MxL expression can also be embedded in page templates (as described in Subsection 3.3.4), whereas this expressions are evaluated for the instances of the corresponding type.

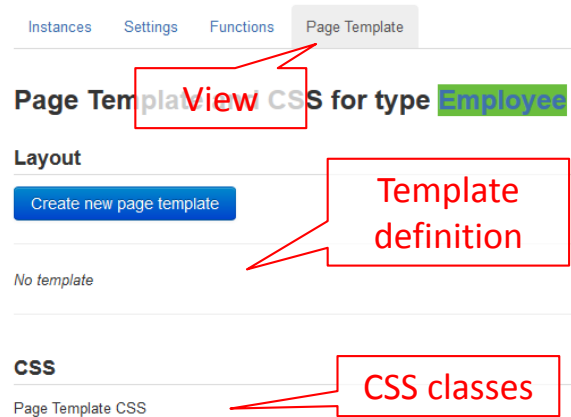


Figure 4.2.: The "Page Template" view of the exemplary type definition *Employee*, whereas neither a template nor CSS classes are defined.

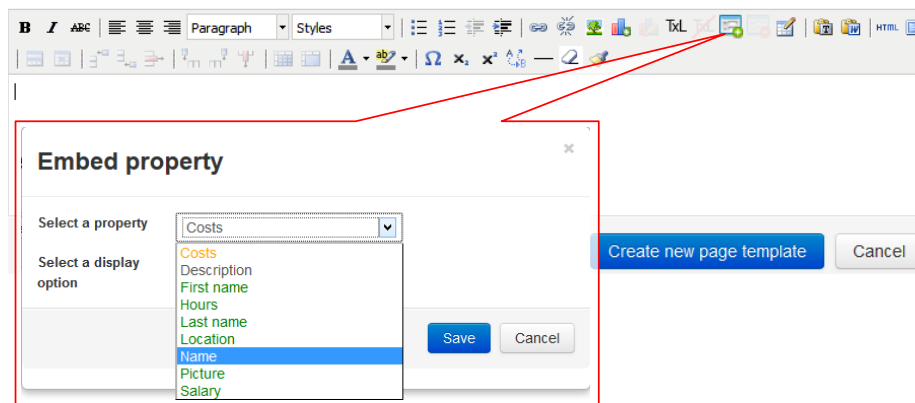


Figure 4.3.: The "Add/Edit property" button for the definition of page templates opens a modal dialog for the selection of a property and its display option (name, value, or the attribute's type). Built-in attributes are colored black, regular attributes blue, and derived attributes are orange

In the view mode of the page template, both the *eval* and *prop* blocks are substituted by a readable description of the embedded property or expression.

Each time a page is requested, it checks if its type provides a page template. If yes, it loads the template, executes all the template's *eval* and *prop* blocks, and inserts the instantiated template into the page in place of its rich-text content.

4.1.3. Example of a page template

In order to define the layout of pages, e.g., of type *Employee* (as defined in Figure 3.3, extended by the attributes *First name*, *Last name*, and *Picture*), an *Employee* page template can be defined as shown in Figure 4.4, whereas the function *projectsAsHtmlList* is defined as follows:

```
"<ul>" +
  get Project whereis Member
    .select(p => "<li>" + p + "</li>")
    .aggregate((a,b) => a + b, "") + "</ul>"
```

The following snippet is an excerpt of *Employee*'s page template:

```
<div class="profilebox profilepicture">
  <span class="embeddedProperty">
    $[prop()]$ {"propertyFullIdentifier":"regularemployeepicture",
               "propertyDisplayOption":"Value"} $prop]$
  </span>
</div>
...
```

By using CSS classes (e.g., "profilebox" and "profilepicture"), recurrent designs can be defined once and used multiple times. Therefore, the type *Employee* defines its CSS classes as follows:

```
.profilepicture { float:left;}
.profilebox { display:inline-table; width:150px; height:230px;
  border-style: solid; border-width: 1px; border-color: #eeeeee;
  padding: 5px; margin: 4px;}
```

By applying the CSS classes onto the template leads to instances as shown in Figure 4.4.

4.2. Deployment of Tricia applications

While all the requirements for the implementation of the Living KPIs prototype are met by the development of both MxL 2.0 (c.f. Chapter 3) and the type-based template engine (as described in Section 4.1), the last remaining issue is the deployment of the Living KPIs environment.

The deployment of the Living KPIs includes the following artifacts:

Integrated information model Initially, the Living KPIs have to provide the whole integrated information model as defined by the EAM KPI Catalog and as depicted in

Template

Value of property 'Picture'

Profile

Name of property 'Last name'

Value of property 'Last name'

Name of property 'First name'

Value of property 'First name'

Name of property 'Location'

Value of property 'Location'

Numbers

Name of property 'Salary'

Value of property 'Salary'

Name of property 'Hours'

Value of property 'Hours'

Calculated

Name of property 'Costs'

Value of property 'Costs'


Projects

Value of expression `projectsAsHtmlList()`

Instance

Thomas Reschenhofer

Created 1 minute ago - Last modified 1 minute ago [View Versions](#)



Profile

Last name:
Reschenhofer

First name:
Thomas

Location:
[Dev Team](#)

Numbers

Salary:
20

Hours:
300

Calculated Costs:
6000

Projects

- [Guided Research](#)
- [Master thesis](#)
- [Bachelor thesis](#)

Figure 4.4.: A template for the exemplary type *Employee* in the template's view mode as well as an exemplary instance.

63

Figure 2.3. Hence, the adaption of the model to an organization-specific context consists of renaming and deleting unnecessary model elements, whereas the MxL 2.0 implementation ensures the consistency of expressions referring to these elements.

EAM KPI descriptions The whole content of the EAM KPI Catalog [13] has to be provided by the Living KPIs, including all the EAM goals, EA layers, as well as the KPI descriptions with their general structure elements as well as organization-specific structure elements.

Templates Not only the EAM KPI Catalog's content, but also its structure has to be implemented in the Living KPIs. Hence, the deployment of the Living KPIs also has to include type-based templates and CSS classes.

KPIs formally expressed by MxL 2.0 All the catalog's KPIs have to be implemented as proper MxL 2.0 expressions in order to analyze the KPI's dependencies on the one hand, and to support a tool-supported computation on the other hand. For the sake of reusability, all the KPIs are implemented as custom functions, so that they are evaluable by invoking the proper function.

Furthermore, the deployment of the Living KPIs has to ensure the possibility of back tracking the changes of the information model in order to retain a relation between the original information model and the adapted one. This allows an analysis of organization-specific adaptations and eventual recurring adaption patterns on the one hand, and the distribution of updates of the catalog on the other hand.

Since a hard-coded approach for the definition of the mentioned artifacts is inflexible and unintuitive, the artifacts, which have to be deployed with the Living KPIs, have to be definable by a descriptive repository, e.g., a text-based file (XML, JSON, ...).

4.2.1. Existing import types

Tricia already supports some mechanism to import data from various data sources, i.a., the following:

Relational data bases By a comprehensive description of how the tables and relations of the relational database management system (RDBMS) have to be mapped to Tricia, this import type allows the import of data of arbitrary SQL-based RDBMSs. However, this import type supports neither the import of MxL 2.0 expressions as custom functions, nor the import of type-based templates.

MS Excel Any existing space can be exported and subsequently imported as a MS Excel file. Again, this import type supports neither the import of MxL 2.0 expressions as custom functions, nor the import of type-based templates. Moreover, the export and import includes only the space's pages, whereas its schema objects (type definitions and property definitions) are omitted, wherefore this import type is not appropriate for the deployment of the Living KPIs.

ZIP-archive Any existing space can be exported as a ZIP-archive containing a XML file describing the exported data as well as eventual documents uploaded to Tricia, which

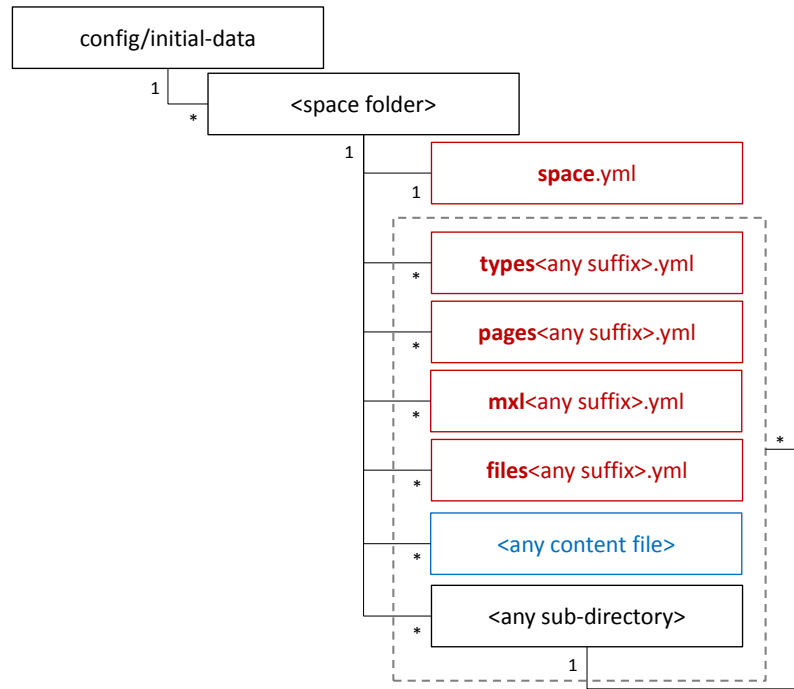


Figure 4.5.: The folder structure of the *config/initial-data* directory.

subsequently can be imported by a proper environment provided by Tricia. Again, this import type supports neither the import of MxL 2.0 expressions as custom functions, nor the import of type-based templates. Moreover, although the definition of a rather big amount of data in a single XML file is a descriptive way of defining data, it yields to a very inflexible, inflated, and confusing XML construct.

In addition to the mentioned weaknesses of these import types, none of them fulfills the requirement of back tracking changes of the information model.

4.2.2. Initial data definition

Because of the weaknesses of existing import mechanisms, the implementation and deployment of the Living KPIs requires another approach, focusing on the definition of data, which has to be initially available on the deployment of the tool. The language used for the descriptive definition of the initial data is YAML, which is a human-readable data serialization language [36]. These files are accessed via the Java-based YAML parser snakeyaml [37], whereas each file is translated to an eventually nested map.

A Tricia application consists of a *config* directory containing various configurations, e.g., accessibility of the Tricia web application or configuration of the integrated search engine. The initial data has to be defined in its sub-directory *initial-data*, whereas its folder structure has to be as depicted in Figure 4.5. The space folders contain both descriptive files (YAML-files) as well as content files (e.g., HTML files defining page templates, CSS files, pictures), whereas these files may be located in arbitrarily named and arbitrarily nested sub-directories.

While the content files do not have to follow a certain naming convention, the name of descriptive files has to have a predefined prefix, which determines the purpose of the file.

space.yml file

In contrast to other descriptive files, there has to be just one YAML file with a "space" prefix.

The *space* file defines both the homepage of the space and the appearance of the space itself (e.g., if the space's types should be shown in the Tricia's navigation).

types.yml* files

The *types* files define the type definitions as well as the property definitions, which have to be created on the initialization of Tricia. The top-level attributes are the names of the types, whereas the uniqueness of the keys of a map ensures the uniqueness of the type names of a space. The possible attributes of a type are *id*, *template* (or *templatefile* referring to a content file containing the template), *css* (or *cssfile* referring to a content file containing the CSS classes), and some attributes defining the appearance of the type's instances (*hideFiles*, *hideTags*, *showHybridTable*, *hideRefBy*). If there is no explicit *id* attribute, the type's ID is inferred by its name by removing all the name's special characters and turning all upper case letters to lower case letters. If a type does not have any attributes, it can be defined as `<typename>: _`

Moreover, a type can also contain the attribute *properties* defining any number of property definitions, which have to be assigned to the type. The syntax of each property definition has to be `<name>: <type><multiplicity><type properties>`. A property's type has to be one of *Text*, *Number*, *Bool*, *Date*, *Link*, *Enum*, *LongText*, or *Image*. The multiplicities are encoded by certain special characters, i.e., the exclamation mark (!) for *exactly one*, question mark (?) for *maximal one*, plus (+) for *at least one*, and the asterisk (*) for *any number of values*. The type property can be used for the *Link* type in order to determine the type of instances the defined relation may refer to, or for the *Enum* type to define the enumeration's elements. For example, the information model depicted in Figure 3.3 can be defined as follows:

```
Employee:
  id: employee
  templatefile: templates/employee.htm
  cssfile: templates/employee.css
  properties:
    First name: Text !
    Last name: Text !
    Picture: Image !
    Salary: Number !
    Hours: Number !
    Location: Link ! Department

Department: _

Project:
  properties:
    Member: Link + Employee
```

In this example, the implicit ID of type *Department* will be "department", whereas the ID of type *Project* will be "project".

Furthermore, the type *Employee* refers to a template file as well as a CSS file defining the type-based page template. These paths are space relative, i.e., inside the space directory there has to be a subfolder named "templates", which has to contain the two files "employee.htm" and "employee.css".

***pages*.yml* files**

While *types* files define the information model's schema, the *pages* files are defining its instances. However, in contrast to types, the top-level attributes of *pages* files are a combination of the page's type and its name, whereas the syntax is `<type>(<name>)`.

The attributes of a page may be *id* (if not explicitly determined, it will be derived from the page's name), *parent* (referring to a parent page by its ID), *content* (or *contentfile* referring to a content file containing the page's richtext content), and some attributes defining the appearance of the page (*hideFiles*, *hideTags*, *showHybridTable*, *hideRefBy*).

Moreover, the page may have an attribute *properties* for the instantiation of the properties defined by the corresponding type definition. For example, the definition of initial pages for the previously defined types may be as follows:

```
Department (Dev Team) : _

Employee(Thomas Reschenhofer) :
  content: This is me
  properties:
    Last name: Reschenhofer
    First name: Thomas
    Picture: picturereschenhofer
    Salary: 20
    Hours: 300
    Location: devteam

Project(Master thesis) :
  properties:
    Member:
      - thomasreschenhofer
```

As shown by this example, the instantiation of references is done by the target page's ID, whereas the IDs of all these pages are implicitly inferred by the corresponding page names.

***mxl*.yml* files**

The *mxl* files are defining both derived attributes and custom functions. Similar to pages, the top-level attributes of these files are a combination of the owner type and the name of the derived attribute or custom function, whereas the syntax is again `<type>(<name>)`.

Both derived attributes and custom functions may define an explicit *id* and a *description*, as well as a numerical *order* attribute to control the order of checking the type of these MxL providers. The *order* attribute is necessary, since a MxL provider A may refer to another MxL provider B, wherefore MxL provider B's type has to be checked first.

If the MxL provider is a derived property definition, it contains an attribute *derive* containing a MxL expression. Otherwise, if the MxL provider is a custom function, it has to have

the attribute *method* defining the method stub of the function. Moreover, custom functions may also contain a *parameters* attribute defining the function's parameters.

For example, the following *mxl* file defines the derived attribute *Costs* from Figure 3.13 as well as the custom function *projectsAsHtmlList* as mentioned in the previous section:

```
Employee(Costs):
  id: employeecosts
  derive: |
    Salary * Hours

Employee(projectsAsHtmlList):
  description: Generates a HTML list of all projects assigned to the employee
  id: projectsashtmllist
  method: |
    "<ul>" +
    get Project whereis Member
      .select(p => "<li>" + p + "</li>")
      .aggregate((a,b) => a + b, "") + "</ul>"
```

For the definition of static functions the pseudo type *STATIC* is used, e.g., *STATIC(doSomething)*. Furthermore, as shown in this example, the pipe (*|*) can be used to define a MxL expression over several lines.

files.yml* files

The *files* files are uploading referred files to Tricia. Again, the top-level attributes are a combination of two attributes. However, this time the first part is a reference to the parent page of the uploading document (by the page's ID), whereas the second part defines the document's name. To define the space's homepage as a document's parent, the token *_home* can be used as the parent page's ID.

The only additional attributes are *id* and a space directory relative *path*, which refers to an arbitrary file, which has to be uploaded to Tricia.

For example, the following *files* file uploads a picture to the space's homepage as well as a PDF to one of the previously defined pages:

```
_home(reschenhofer.jpg):
  id: picturereschenhofer
  path: pictures/reschenhofer.png

thomasreschenhofer(CurriculumVitae.pdf):
  id: cvreschenhofer
  path: files/cv.pdf
```

4.2.3. Initial data processing

Since some assets defined for the creation on the initialization of Tricia may depend on each other, the order of their initialization is relevant.

The initialization process, which is depicted in Figure 4.6, consists of the following steps:

1. For each folder contained in the *config/initial-data* directory, Tricia checks for the existence of a space which is named as the corresponding folder. If yes, the process continues with the next space folder. Otherwise Tricia will create a space with the

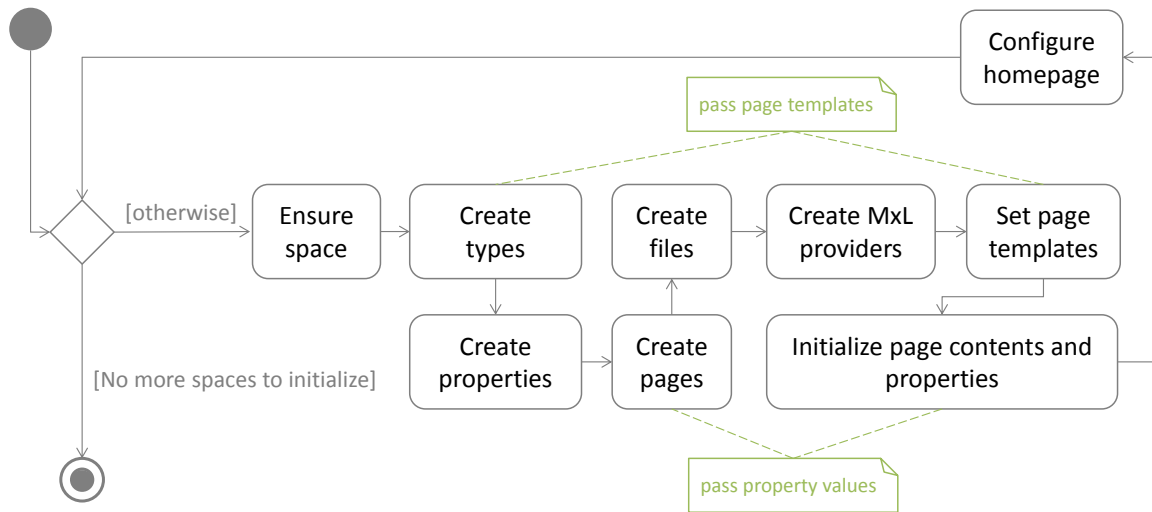


Figure 4.6.: UML activity diagram [17] depicting the process of initializing the data as defined by the content of the *initial-data* directory.

folder's name (its ID is inferred by the name) and continues with the initialization of the space.

2. Tricia searches for *types* files in the space folder and all its sub-folders, and collects all defined types and their property definitions. However, since the property definitions may define relations to other types, Tricia first creates all types before adding their property definitions. Moreover, the page templates are not yet created, since they may contain an embedded expression referring to other MxL assets (properties, functions).
3. Subsequently, the property definitions are created as defined in the *types* files, which were obtained in Step 2.
4. Tricia searches for *pages* files in the space folder and all its sub-folders and collects all defined pages as well as their their content and properties. However, since pages may refer to other pages, and because the rich-text content of a page may contain an embedded expression, the pages are created without initializing their content and properties.
5. Tricia searches for *files* files in the space folder and all its sub-folders, and collects all defined files, uploads them, and attaches them to the defined parent pages.
6. Tricia searches for *mxl* files in the space folder and all its sub-folders, and collects all defined derived attributes and custom functions, and creates them in the order specified by the *order* attribute.
7. Since, all MxL assets are created, the page templates collected in Step 2 are applied to the corresponding types

8. Subsequently, all page contents and properties are initialized with the data obtained in Step 4
9. Finally, the homepage is configured according to the *homepage* attribute in the *space* file.

4.2.4. Back tracking of changes on initial data sets

The implemented deployment mechanism ensures the creation of all assets by a predefined ID, which is either explicitly defined or implicitly inferred (e.g., by the asset's name). Hence, because of the immutability of an ID, an asset is always identifiable, even if it was renamed or its content has changed. This feature enables, i.a., the following actions:

Initial data update If the initial data sets are updated after initializing Tricia, the changes can be propagated to the existing data sets by their ID.

Data reset All the data created on the initialization of Tricia can be reseted to its initial state, while all the data created afterwards remains the same.

Change analysis The deviations of an actual data set from the initial data set can be analyzed in order to optimize the initial data set. For example, if there is a recurrent renaming of a certain asset, it can be renamed in the initial data definition.

4.3. Prototype of the Living KPIs

While Section 2.4 exposed the shortcomings of the foundations of the thesis, the development of MxL 2.0, as well as the implementation of both a type-based template engine and a flexible deployment mechanism rectified the mentioned issues:

- MxL 2.0 supports compile-time analysis of expressions
- MxL 2.0 is decoupled from Tricia and hence integratable in other tools by implementing a proper MxL Connector
- Tricia now supports the definition of type-based page templates to centralize the definition of the layout and design of a type's instances
- Tricia now provides a mechanism to intuitively and descriptively define data, which has to be created on Tricia's initialization, including the information model's schema and instance data, as well as MxL providers like custom functions and derived attributes.

Hence, by the work done in this thesis, a first prototype of the Living KPIs can be implemented.

Custom MxL Function

STATIC::applicationContinuityPlanAvailabilityKPI

Name	applicationContinuityPlanAvailabilityKPI
Description	A measure of how completely IT continuity plans for business critical applications have been drawn & tested up for the IT's application portfolio
Parameters	
Return Type	Number

```

Method Stub      /* Determine all critical applications */
                  let criticalApplications =
                    find('Business Application').where('Is critical') in

                  /* Calculate proportion of covered critical applications */
                  criticalApplications.ratio('Covering continuity plan' <> null)

```

Outgoing MxL References

Basic Functions Sequence::ratio Sequence::where	Property Definitions Business Application::Covering continuity plan Business Application::Is critical	Types Business Application
--	--	--------------------------------------

Figure 4.7.: The implementation of the EAM KPI depicted in Figure 2.1 as a MxL 2.0 custom function (c.f. Figure 2.10).

4.3.1. Implementation of KPIs as MxL 2.0 custom functions

As stated in Subsection 2.3.2, MxL 1.0 was already able to define the catalog's KPIs, as stated by Monahov et al. [15]. For example, the definition of the first KPI of the catalog as a MxL 1.0 custom function is depicted in Figure 2.10.

However, because of MxL 1.0's shortcomings mentioned in Section 2.4, MxL 2.0 was developed and prototypically integrated in Tricia. All the KPIs of the EAM KPI Catalog are implemented as MxL 2.0 custom functions (e.g., Figure 4.7 shows the definition of the previously mentioned KPI *Application continuity plan availability*), allowing the analysis of the expressions representing the formal computation prescriptions of the corresponding KPIs. Hence, the MxL type checker determines all information model elements the expression refers to via an identifier, and lists them as "Outgoing MxL References" below the function's definition.

4.3.2. Page template for EAM KPI descriptions

In addition to all the types of the EAM KPI Catalog's integrated information model depicted in Figure 2.3, the Living KPIs prototype provides the type *EAM KPI Description*, whereas this type's instances represent the catalog's KPIs.

The attributes of the *EAM KPI Description* type are both the general structure elements (c.f. Subsection 2.1.1) and organization-specific structure elements (c.f. Subsection 2.1.2). Moreover, this type provides an additional attribute *Function* referring to the custom function defining the KPI's formal computation prescription.

However, as stated in Subsection 2.1.2, each KPIs also possesses a mapping table linking the catalog's information model elements to organization-specific concepts. Since this table's structure strongly depends on the KPI's definition, and Tricia does not provide a complex property type allowing the definition of a table structure, the mapping has to be defined unstructured, i.e., as rich-text content. Therefore, the mapping table has to be defined for each KPI separately, while the page template defines its position on the page.

To ensure a consistent and uniform layout and design of the *EAM KPI Description*'s instances, a page template is defined 4.8, whereas its layout is deduced from the traditional EAM KPI Catalog's layout and design. This template is applied on each of the type's instances. Hence, by simply instantiating the KPI description's properties as well as its mapping table, its layout is as depicted in Figure 4.9.

Due to the definition of the layout as a type-based page template, any changes of the EAM KPI structure (e.g., organization-specific adaptations) are easily propagatable to all instances.

4.3.3. EAM KPI Catalog data

In order to deploy the Living KPIs, the mechanism introduced in Section 4.2 is used create all the catalog's data on the initialization of Tricia.

For this purpose, a new so-called *Tricia App* with name "LivingKPIs" was created in order to build upon the basic functionality of Tricia. However, the only extension of this *Tricia App* is the definition of Tricia's initial data sets as described in Section 4.2. Therefore, in the *initial-data* folder of the *Tricia App*'s *config* directory a space folder named "EAM KPI Catalog" is defined, whereas its directory structure is depicted in Figure 4.10. The elements of the space folder are:

space.yml Defines the appearance of the space's homepage by showing a table of all EAM KPI Descriptions in the homepage's rich-text content, whereas the content's HTML markup is defined in the file "homepage.htm".

types_catalog.yml This is the definition of the EAM KPI Catalog's integrated data model.

mxl.kpis.yml In this descriptive file, all the catalog's KPIs are defined as MxL 2.0 custom functions

kpidescription This folder contains the definition of the types *EAM KPI Description* (as described in the previous subsection), *Goal*, and *Layer* (in "types_kpidescription.yml"), as well as the definition of all KPIs (in "pages.kpidescription.yml"), EAM goals

Name of property ,Description'		
Value of property ,Description'		

Name of property ,EA information model'	Name of property ,Goal'
Value of property ,EA information model'	Value of property ,Goal'

Organization-specific mapping of the predefined information model elements	Name of property ,Calculation'
Value of property ,Description'	Value of property ,Calculation'

Organization-specific instantiation Sort ▼	Name of property ,Function'
	Value of property ,Function'

KPI property	Property value
Name of property ,Frequency of Measurement'	Value of property ,Frequency of Measurement'
Name of property ,Interpretation'	Value of property ,Interpretation'
Name of property ,KPI consumer'	Value of property ,KPI consumer'
Name of property ,Owner'	Value of property ,Owner'
Name of property ,Target Value'	Value of property ,Target Value'
Name of property ,Planned values'	Value of property ,Planned Values'
Name of property ,Tolerance values'	Value of property ,Tolerance Values'
Name of property ,Escalation rules'	Value of property ,Escalation rules'

Name of property ,Code'
Value of property ,Code'

Name of property ,Source'
Value of property ,Source'

Name of property ,Category'
Value of property ,Category'

Name of property ,Layer'
Value of property ,Layer'

Figure 4.8.: The prototypical page template for the type *EAM KPI Description*, whose layout and design is deduced from the layout of the traditional EAM KPI Catalog.

and EA layers (in "pages_goalsandlayers.yml") of the EAM KPI Catalog. Moreover, the subfolder "templates" contains the page template for the *EAM KPI Description* type, while the other sub-folders are containing the mapping tables as well as the information models for each of the catalog's KPIs, whose upload is defined by "files_kpimodels.yml".

testdata For testing purposes, this folder contains exemplary instances for all the types of the catalog's integrated information model.

Therefore, by starting the Living KPIs prototype, Tricia is initialized with all the EAM KPI Catalog's data and structure.

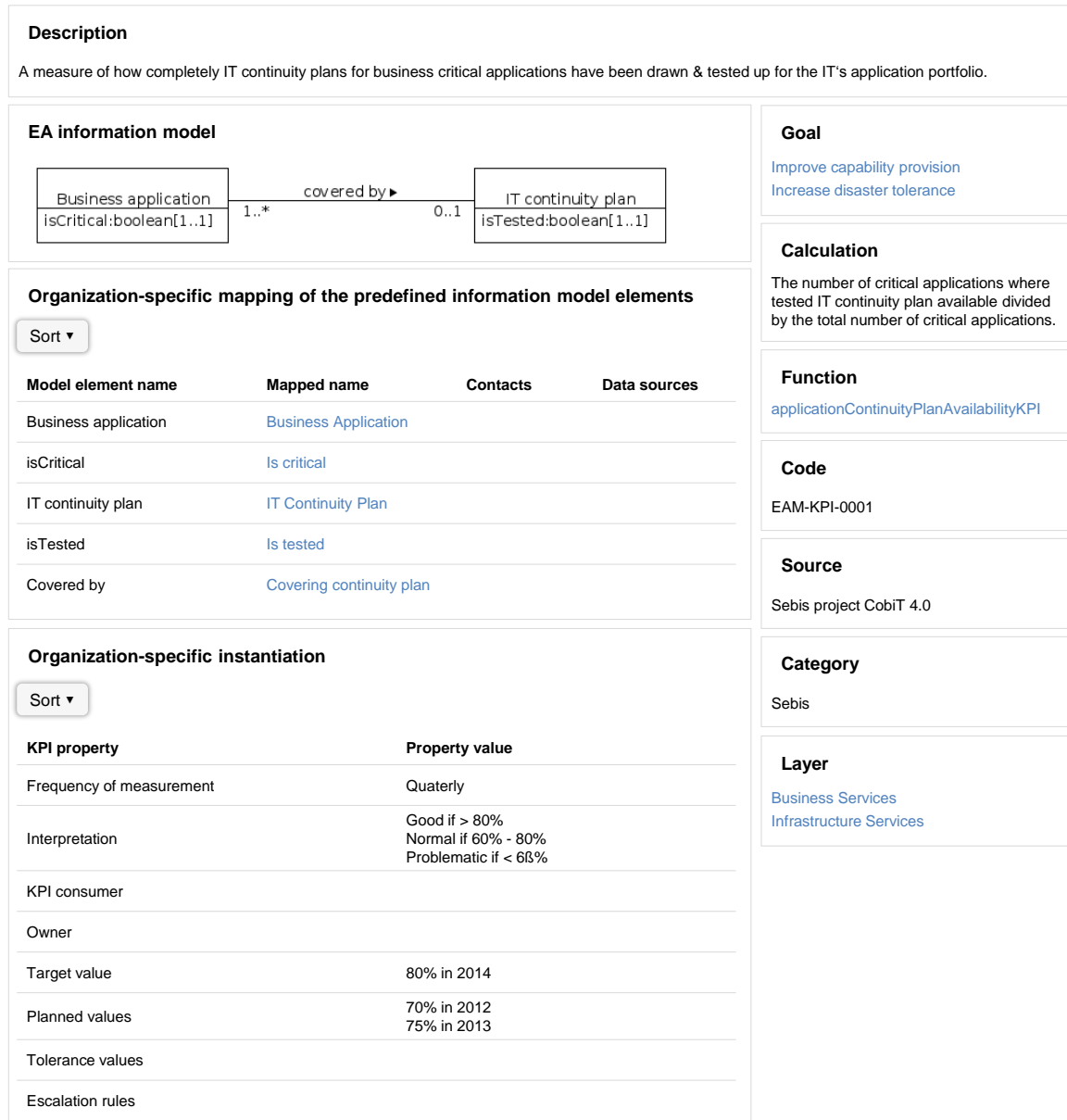


Figure 4.9.: The first KPI of the EAM KPI Catalog, implemented in the Living KPIs. The layout is based on the prototypical template in Figure 4.8, yielding to an appearance similar to the one of the traditional catalog (c.f. Figure 2.1).

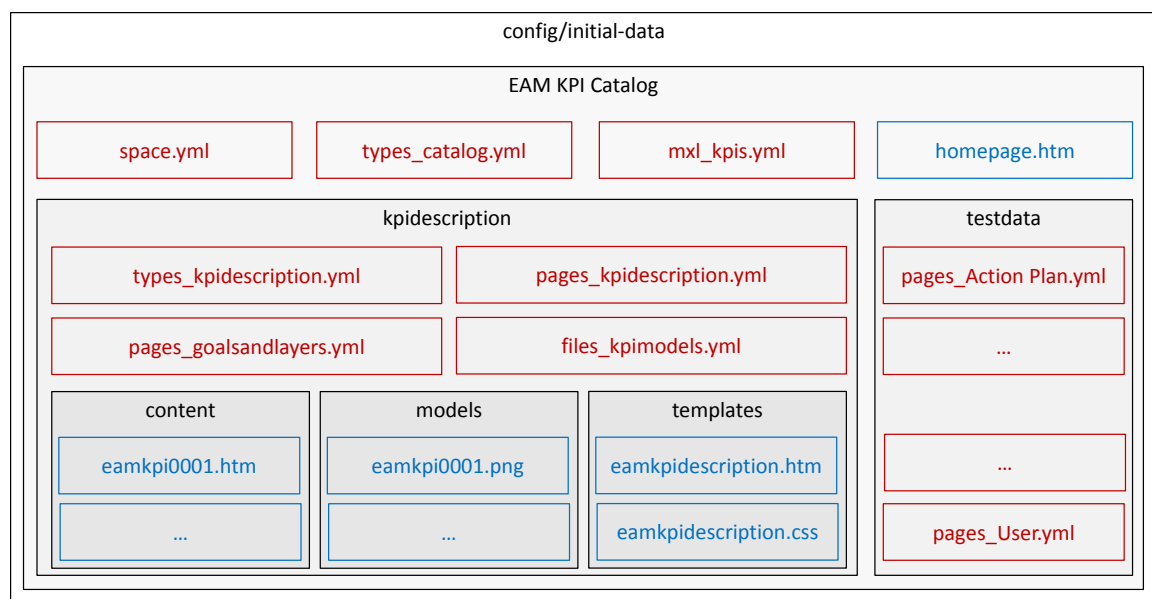


Figure 4.10.: The folder structure of the `config/initial-data` directory defining the Living KPI's initial data.

Part III.

Results

5. Summary & Conclusion

Chapters 3 and 4 are extensively describing the contribution of this thesis, consisting of the re-engineering of MxL 1.0's successor – MxL 2.0 – as well as the implementation of both a type-based template engine and a deployment mechanism in Tricia.

This chapter, summarizes (Section 5.1) and concludes (Section 5.2) the work done by this thesis.

5.1. Summary

As motivated in Chapter 1, KPIs are an important tool to monitor, measure, and evaluate certain performance-related EA characteristics and to measure the degree of achievement of given EAM goals. Due to the lack of practice-proven KPIs, Matthes et al. [13] gathered 52 KPIs from literature and industry partners for the development of the EAM KPI Catalog. Based on this catalog, Monahov et al. [15] designed the language MxL 1.0 for the formal computation prescription of the KPIs, enabling a tool-supported evaluation of the defined KPIs.

However, as stated in Section 1.3, the goal of the thesis is the prototypical implementation of an integrated and adaptable EAM platform based on the EAM KPI Catalog. However, Chapter 2 reveals some shortcomings of existing foundations, namely MxL 1.0's missing support for compile-time analysis, and the lack of both a type-based template engine and a proper deployment mechanism in Tricia. Moreover, MxL 1.0 is very tightly coupled to Tricia, ruling out the integration of MxL 1.0 in other tools.

To address MxL's issues, Chapter 3 covers the development of MxL 2.0 and its implementation in Tricia. In contrast to MxL 1.0, MxL 2.0 is type safe, i.e., a type checker analysis an expression at compile-time and checks, if all the expression's identifiers are referring to existing assets (c.f. Subsection 3.1.5). By interpreting an EA as a dynamic system, MxL 2.0 expression allow the formalization and analysis of dependencies of system components. Moreover, MxL 2.0 was decoupled from Tricia. It provides a connector component as interface between MxL 2.0 and the implementing system, so that MxL 2.0 is easily integratable in any tool.

On of the subjects of Chapter 4 is the extension of Tricia by a type-based template engine. While Tricia in its original state already provided a template-engine, it was not possible to define type-specific layouts, but just one common page layout. Hence, the implemented type-based template engine allows the definition of a page template for each type, whereas this layout is applied onto each of the type's instances. Furthermore, Section 4.2 covers the implementation of a mechanism to deploy the EAM KPI Catalog's data and structure in an intuitive and descriptive way. This allows the definition of the data, which has to be created on Tricia's initialization, in a human-readable notation, so that the Living KPI prototype's data is flexible and easily adaptable to eventual changes of the EAM KPI Catalog.

5. Summary & Conclusion

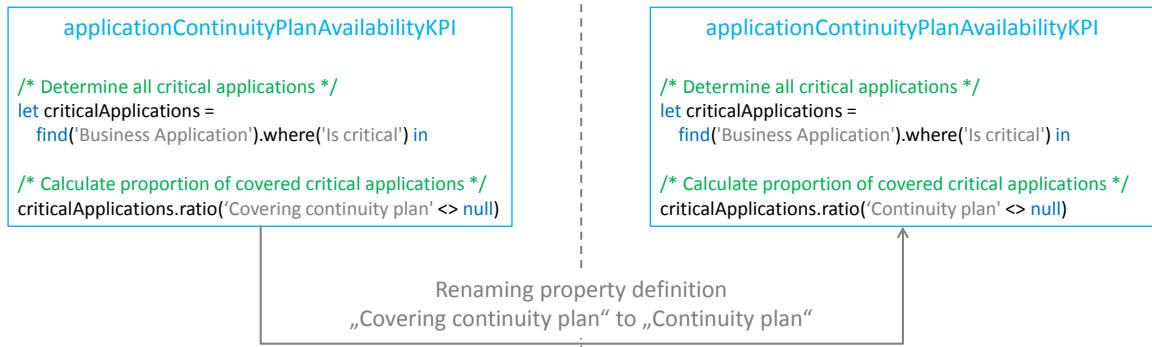


Figure 5.1.: Automated update of all MxL 2.0 expressions containing identifiers referring to an asset, which is renamed. In this example, the custom function *applicationContinuityPlanAvailabilityKPI* contains the identifier 'Covering continuity plan' referring to a corresponding property definition. By renaming the property definition, the custom function's expression is automatically updated

Finally, Section 4.3 describes the implementation of the Living KPIs, enabled by this thesis' contribution, i.e., by using MxL 2.0 to formally define the EAM KPI Catalog's KPIs, defining a page template for the EAM KPI descriptions, and defining all the catalog's data and structure as initial data, which has to be loaded on the initialization of the Living KPIs.

5.2. Conclusion

As motivated in Section 1.3, the goal of the thesis is an integrated and easily adaptable EAM platform based on the EAM KPI Catalog. The approach of the thesis is the prototypical implementation of such a platform in Tricia and by using the domain-specific language MxL 2.0, whose compile-time analysis capability enables the retention of the platform's consistency on changes of the information model.

Not only the EAM KPI Catalog's data and structure is initially available in the Living KPIs, but also a MxL 2.0 custom function as the formal computation prescriptions of each KPI. Most of them are static functions, i.e., they are listed in the "Static Functions" view of the *EAM KPI Catalog* workspace, whereas other functions (e.g., *projectsEmployeeAndContractorMixKPI*, representing the 6th KPI of the catalog) have an owner type (e.g., *Project*), and hence are listed in the "Functions" view of the corresponding type.

As previously mentioned, the adaptability of the platform is provided by MxL 2.0, i.e., the identifiers of a MxL 2.0 expression are linked to the assets they are referring to. Hence, if any asset (e.g., type, attribute, function) is renamed or deleted, all MxL providers (functions, derived attributes, and embedded expressions) are notified, so that the expression containing the referring identifier can either be updated or deleted.

For example, as shown in Figure 4.7, the custom function *applicationContinuityPlanAvailabilityKPI* has an outgoing reference to the property definition *Covering continuity plan* of type *Business Application*. By clicking onto this property definition and observing its incoming MxL references, it can be observed that there is another function also referring to this property definition. If the property definition is renamed, the expressions of all

MxL providers referring to this property definition are updated (e.g., the identifier in *applicationContinuityPlanAvailabilityKPI*'s method stub is updated to the property definition's new name, c.f. Figure 5.1).

Similarly, assets can also be deleted, whereas there are two options to handle incoming MxL references to a deleting asset. The first option is the triggering of a *Cascades Delete*, i.e., all MxL providers referring to the deleting asset should also be deleted yielding also to the deletion of all MxL providers, which transitively refer to the deleting asset. The second option is to omit the deletion of MxL providers referring to the deleting asset, whereas these MxL providers are becoming inconsistent.

Therefore, by renaming and deleting the Living KPI platform's initial information model elements, it can be adapted to organization-specific needs, without violating the integrity of the Living KPIs. However, by the IDs of the assets, an adapted Living KPIs prototype is always resettable to its initial state on the one hand, and updates on the EAM KPI Catalog's structure are easily propagatable to adapted versions of the Living KPIs on the other hand.

Therefore, the Living KPIs are fulfilling the requirements examined in Section 1.3, since being an integrated and adaptable EAM platform based on the EAM KPI Catalog.

6. Outlook & Future research

Although the current state of the Living Catalog already is a flexible and adaptable EAM platform, there are still some open issues, whereas most of them are related to MxL and the processing of its expressions.

This chapter will point out these issues, which can be subject in future research. Moreover, the following sections are proposing possible solutions to tackle the mentioned issues.

6.1. Authorization in MxL 2.0

For multi-user application, authorization is a prevalent security requirement for specifying the access rights to resources [38].

For example, Tricia implements an authorization mechanism allowing to specify, which user or group can *read* (permits read-only access), *write* (permits write access; implies *read*), or *administrate* (permits change of access rules, implies *write*) a page, document, or space. The authorization model of Tricia is depicted in Figure 6.1.

6.1.1. Problem / Open issue

While the access to Tricia's information objects is controllable, the access to MxL functions is not. Hence, each user is allowed to call each MxL function without any restriction, i.e., it is not possible to define functions, which are just invocable by a certain group of users (e.g., a KPI only evaluable by a certain manager).

However, since the MxL's access to the information model is built upon the authorization model of the information objects, the authorization rules defined on information objects are also applied when accessing them via a MxL expression. The issue regarding an executor's identity is also considered in Section 6.2.

6.1.2. Proposed solution

The lack of an authorization mechanism for MxL functions can be tackled by applying the authorization model for information objects on MxL functions, whereas the three permission levels *read*, *write*, and *administrate* would have a slightly different meaning for functions:

read A function's reader is allowed to invoke the function, but also to access the function's definition. However, the reader is now allowed to change the function's name, description, parameters, or method stub.

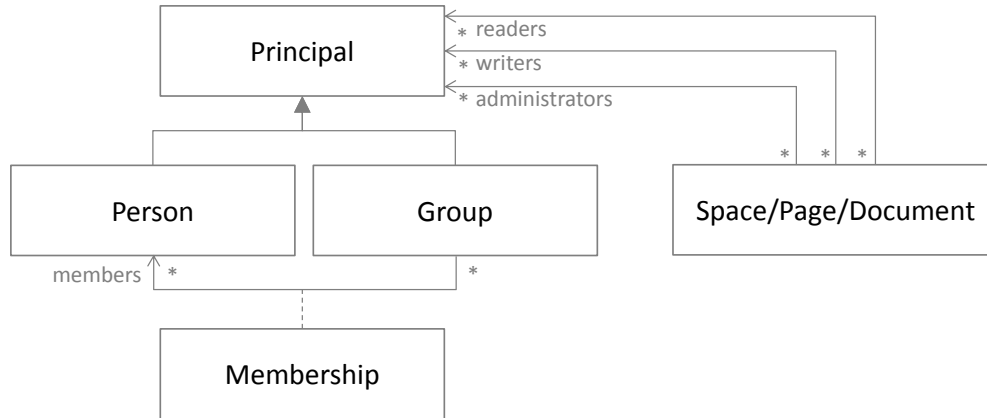


Figure 6.1.: The conceptual authorization model of Tricia.

write A function’s writer is allowed to invoke and read the function, and also to change the function’s name, description, parameters, and method stub. In fact, in Tricia’s current state, each user is implicitly writer of each function.

administrate A function’s administrator is a writer of the function, but additionally manages the function’s access rights.

While all these access rights apply on existing functions, the authorization rule allowing users to create new functions, has to be defined on another object. For example, a space may contain a role *MxL creators* referring to all users allowed to create both static functions in this space and to create functions for types of this space. Alternatively, the *MxL creators* may be defined for each type definitions to control the creation of MxL functions for each type separately.

6.2. Evaluating identity

As already mentioned in Section 6.1, MxL builds upon the authorization mechanism of Tricia, so that the authorization rules defined on information objects are also applied when accessing them via a MxL expressions. Furthermore, a MxL provider (functions, derived attributes, and objects containing embedded expressions) is always executed under the executor’s identity.

6.2.1. Problem / Open issue

If a MxL expression defines the query of an underlying information model, the executing user’s identity is used to access the information objects. However, this implies that a MxL Provider defining a query may return different results for different users.

While this might be useful for some scenarios (e.g., to define user-specific views), the definition and evaluation of KPIs requires another approach, since they have to return the same value for each user.

Furthermore, a use case, in which users should not have access to individual data objects

of a certain data set, while consolidations and aggregations of the data set have to be accessible, is not implementable with the current approach of evaluating MxL expression.

6.2.2. Proposed solution

While the execution of a MxL provider's expression under the executor's identity is useful for the definition of user-specific queries, there is a need for other approaches (e.g., evaluation of KPIs).

Conceivable approaches for the selection of an identity, under which a MxL provider is executed, are:

Executor of the function/derived attribute This is the current approach, using the executor's identity to perform the expression's actions.

Definer of the function/derived attribute The execution of a function or derived attribute under its creator's identity is a prevalent approach to execute a process (e.g., a workflow) under a common identity.

Specified user identity The creator of a function or derived attribute can specify an user, under whose identity the MxL provider is executed.

System identity There may be a certain system identity, under which MxL providers are executed. For example, this identity might be the web application's process identity.

All these approaches are practice-proven and used in several platforms, e.g., in Microsoft SharePoint 2010¹ workflows [39].

Since there are use cases for the current as well as the mentioned alternative approaches, an optimal solution would be to allow the selection of the current approach and at least one of the other ones. This would enable the definition of both user-specific and user-independent MxL providers.

6.3. Evaluation strategy

In this context, an evaluation strategy refers to the time a MxL expression is evaluated. Currently, each expression is evaluated on demand, i.e., each time a function or derived attribute is invoked, its expression is reevaluated.

6.3.1. Problem / Open issue

While the current approach is unproblematic for trivial expressions, extensive queries affecting lots of information objects as well as complex processing of the data may lead to exorbitant evaluation durations. This is especially painful, if the result of the expression's evaluation has not changed compared to the previous one.

While this problem will occur especially when querying large data sets, it is not solely tied to query expressions, but to low performance expressions in general. The optimization of queries will be discussed later on in Section 6.6.

¹<http://office.microsoft.com/en-us/sharepoint>

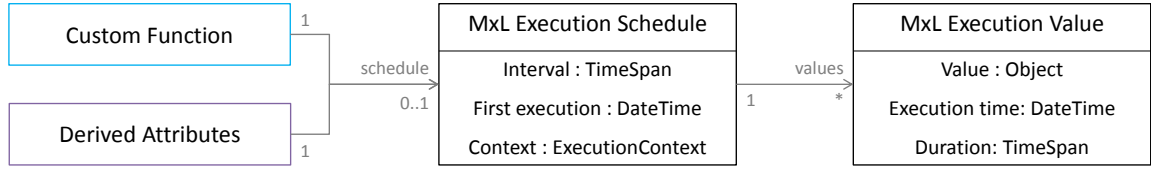


Figure 6.2.: A proposed architecture for the scheduling evaluation strategy for custom functions and derived attributes.

6.3.2. Proposed solution

While an optimization of an expression might enhance its evaluation performance, this is not feasible in general. Hence, the only way to enhance an arbitrary expression's evaluation performance is the selection of a proper evaluation strategy, whereas the following lists contains four possible approaches:

On demand This is the current approach for all MxL expressions, i.e., each time a function or derived attribute is invoked, its expression is evaluated.

Cached The most obvious evaluation strategy for optimizing an expression's evaluation performance is caching (temporary storing) an expression's result for future requests. For example, on the definition of a custom function, the creator could determine the function's cache lifetime. Hence, if the function is invoked, its result is cached for the specified duration and not reevaluated until the cache expires.

Scheduled In contrast to the caching approach, this evaluation strategy ensures the regular evaluation of an expression based on a configurable schedule, while in the caching evaluation strategy an expired cache yields to an on-demand evaluation. Furthermore, storing the values obtained by each scheduled evaluation allows easy access to the expression's history, which will be subject in Section 6.4.

A conceptional architecture for the implementation of the scheduling might consist of an execution schedule, which is linked to either a custom function or derived attribute, whereas the schedule could be configured by the creator of the MxL provider (c.f. Figure 6.2). Moreover, an execution value represents a value evaluated according to the related schedule. Hence, if a function has to be evaluated, it determines and returns the latest execution value of the related schedule instead of reevaluating the function's method stub.

However, the execution schedule has to consider the execution context of the function, i.e., its execution identity (as mentioned in Section 6.2), the object, on which the function is invoked (for non-static functions), and the function's parameters.

On change While in the on-demand evaluation strategy, the incentive for an expression's reevaluation comes from its executor (e.g., an user), in the on-change the reevaluation is caused by the information model the expression is based on (as illustrated in Figure 6.3). Hence, if a data set is updated, each query expression referring to these data set is reevaluated.

While all previously mentioned evaluation strategies are rather simple implementable,

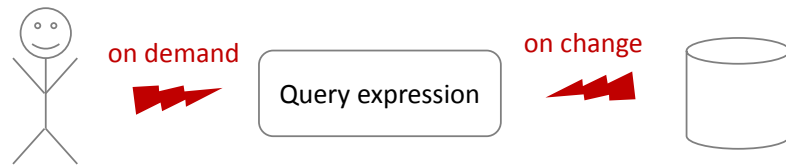


Figure 6.3.: Illustration of the evaluation strategies *On demand* and *On change*.

this one requires an exhaustive analysis of MxL expressions and their relation to the information model. However, since MxL 2.0 supports a compile-time analysis, the on-change evaluation strategy can be implementable in Tricia.

6.4. History of evaluation results

One of the most important activities in EAM is the analysis of the EA's evolution [2]. For this purpose, it might be helpful to look at a KPI's evolution and, for example, make predictions regarding the KPI's future trend.

Since versioning is an important concept in information systems in general, Tricia already manages versions for its pages and documents, which includes functions for comparing two versions and restoring previous versions.

6.4.1. Problem / Open issue

While the evolution of the information model's data itself is potentially accessible through Tricia's versioning feature, MxL 2.0 is not able to access the information model's version history.

However, accessing the information model's version history would allow the definition of functions over the time representing a KPI's evolution over a certain time period. Moreover, a visualization of a KPI's evolution as a time series diagram would illustrate the KPI's trend in a human-readable manner (visualizations are also subject in Section 6.5).

6.4.2. Proposed solution

Basically there are two ways of gathering a MxL expression result's evolution over time, namely either

- by managing the history of the expression's evaluated values, or
- by accessing the information model's history.

Assuming the availability of the previously mentioned scheduling evaluation strategy (c.f. Section 6.3), a hybrid approach would be a possible solution for the mentioned issue, as described in the following.

In a hybrid approach, scheduled functions and derived attributes are storing their evaluated values anyway, wherefore they are managing the evaluation value's history. For

Custom MxL Function**STATIC::applicationContinuityPlanAvailabilityKPIByDate**

Name	applicationContinuityPlanAvailabilityKPIByTime
Description	Evaluates the KPI 'Application continuity Plan availability' at the given date
Parameters	evalDate : Date
Return Type	Number

Method Stub	<pre>/* Use of the get-at construct */ get applicationContinuityPlanAvailabilityKPI() at evalDate</pre>
-------------	---

Outgoing MxL References**Custom Functions**

STATIC::applicationContinuityPlanAvailabilityKPI

Figure 6.4.: An exemplary function for the evaluation of the KPI from Figure 4.7 at a given date.

other expressions, MxL 2.0 has to be extended by a construct to access an information object's history, i.e., to obtain an information object's state at a given time. For example, this construct might be defined as follows:

get <any expression> **at** <any time>

By this construct, each time the expression accesses the information model, MxL 2.0 does not obtain the current values, but retrieve the information model's history to obtain the value at the given time. Hence, the *get-at* construct allows the definition of functions over time, e.g., Figure 6.4 defines a function evaluating the KPI function defined in Figure 4.7 at a given time.

However, by providing the history of an expression evaluation value, a new issue arises, which has to be taken into consideration: What to do on a change of the MxL 2.0 expression itself? The two solutions are to either keep the values evaluated by the old expression, or to evaluate the new expression with the old data.

6.5. Visualization of evaluation and type checking results

The visualization is the a human-friendly representation of data (e.g., KPIs), reinforcing the human's cognition and supporting the user's understanding and interpretation of the information [40].

In the context of EAM, KPIs are often quantitative values ranging from 0 % to 100 %, whereas this range is divided into a problematic, normal, and good sub range (for example, c.f. KPI interpretation in Figure 2.1). Hence, a prevalent visualization for KPIs is the



Figure 6.5.: An exemplary visualization of the KPI from Figure 2.1.

traffic light, which is able to visualize the three states of a KPI. An exemplary definition of a traffic light visualization was already subject in Subsection 3.3.4.

Moreover, since the MxL 2.0 interpreter consists of a type checker, all the MxL assets (types, attributes, functions, ...) an expression is referring to are obtainable, enabling the determination of all assets the expression depends on. Therefore, the representation of this environment as a graph with MxL asset's as its nodes and the dependencies as its edges yields to a complex network, whose analysis enables powerful features like the on-change evaluation strategy mentioned in Section 6.3.

6.5.1. Problem / Open issue

However, while a simple traffic light visualization is implementable by the conditional rendering of HTML markup (because there are just three different states), more complex visualizations require another approach. For example, the visualization of a KPI as shown in Figure 6.5 contains slightly more information (the concrete value of the KPI), so that this visualization is practically no longer definable by the conditional rendering of HTML markup, because in this way – even by assuming just natural numbers for the KPI – its definition would require exactly 100 *if-then-else* constructs, since there are 101 different values.

Moreover, as mentioned in Section 6.4, the analysis of an EA's evolution is one of the main activities in EAM. For this purpose, a KPI's value and its trend can be visualized as a time series, which furthermore allows the prediction of future values. However, time series are not implementable by simple visualizations based on HTML markup.

The same is also true for the visualization of dependency graphs as networks.

6.5.2. Proposed solution

There are already many existing JavaScript libraries for the rendering of a variety of visualizations (time series, bar charts, pie charts), e.g., Highcharts², Raphael³, or JavaScript InfoVis⁴.

Hence, in order to visualize a MxL expression's value and its evolution, an interface between the visualization library and MxL is necessary, so that a MxL expression feeds the visualization with proper data sets.

For example, to visualize the KPI *Application continuity plan availability* of Figure 4.7 as a time series, the chart's x-axis has to be initialized with a sequence of dates, while the corresponding y-values are computed by a proper MxL function, which takes a single x-value

²<http://www.highcharts.com>

³<http://dmitrybaranovskiy.github.io/raphael/>

⁴<http://philogb.github.io/jit>

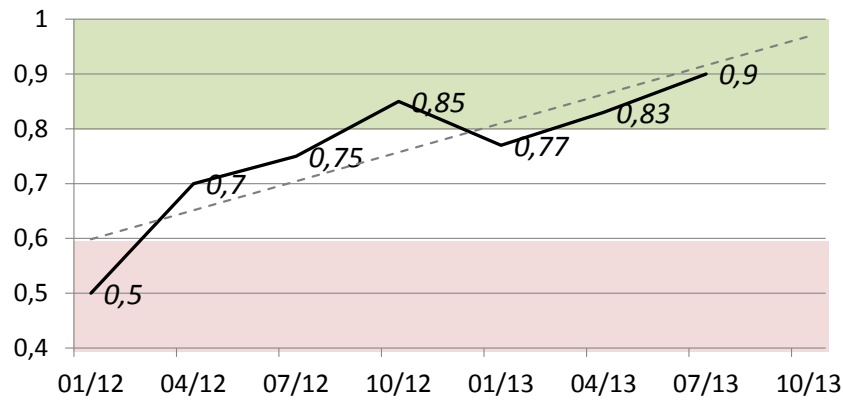


Figure 6.6.: An exemplary visualization of the KPI from Figure 2.1 as time series. This visualization also contains a trend line predicting the KPI's value for the next quarter.

as parameter and returns the KPI's value for the given time (such a function's definition is shown in Figure 6.4). In this way, a visualization as shown in Figure 6.6 could be definable. For the definition of network graphs, Tricia has to analyze its MxL assets and the dependencies between them. A proper JavaScript visualization library can be fed with this data in order to render a network representing the dependencies between MxL functions, attributes, and other MxL assets (similar to the visualization in Figure 3.18).

6.6. Query processing

As already mentioned several times, the definition of queries on an underlying information model is one of MxL's main purposes. In this context, querying means ,i.a., filtering, projecting, sorting, and grouping a sequence of information model elements based on lambdas defining proper predicates or map functions.

6.6.1. Problem / Open issue

MxL 2.0 does not parse its queries into a language suitable for querying the underlying information model's repository (SQL for querying relational data bases), since this parsing is highly dependent on the underlying repository's type.

For example, the following query determines all employees with a salary higher then 20:

```
find (Employee)
  .where(e => e.Salary > 20)
```

For this query, MxL loads all employees and subsequently iterates through all of them to check the given predicate. Therefore, the currently implemented query processing is highly inefficient, since it is completely done in Java.

6.6.2. Proposed solution

If the underlying repository would implement the information model's types as proper tables (e.g., "Employees") with a proper column for each property (e.g., "Salary"), this query would be easily parsable to the following SQL statement:

```
SELECT * FROM Employee AS e WHERE e.Salary > 20
```

However, while trivial predicates (e.g., *Salary* > 20) are easily parsable to a proper SQL equivalent, this is not true in general (e.g., if the predicate contains basic function calls).

As mentioned in Subsection 2.3.1, MxL was inspired ,i.a., by Microsoft's LINQ, which is a first-class query concept enabling an unified access to a multitude of data sources [21], i.e., similar to MxL, LINQ is abstracting the querying of an underlying repository. Hence, LINQ also has to deal with the issue of parsing complex lambdas.

In order to solve this issue, LINQ implements a hybrid approach: All operators, which are supported by the repository's query language, are parsed, whereas the others are implemented in LINQ's implementation language (e.g., C#, VB.NET). A similar approach would be conceivable for MxL's query processing.

While this hybrid query processing approach would work for many systems implementing MxL 2.0, Tricia's Hybrid wiki concept causes another issue: In Tricia, an information object's schema is changeable at runtime, which excludes the implementation of the information model's types as static tables in a relational database. Hence, all the page's attribute as defined by an assigned type definition are serialized to a flexible JSON object and stored in a single repository field, so that it is not queryable by SQL. Hence, the strong dependency of the query processing on the implementing system's persistence layer makes it very hard to implement a general query processing mechanism.

However, addressing distributed and efficient data and query processing, the programming model *MapReduce* enables automatic parallelization and distribution of large-scale computations [41]. As its name suggests, MapReduce consists of a map function (applied onto each element of a source data set emitting key-value pairs) and a reduce function (processes all key-value pairs with the same key), whereas each map and reduce function is independent from each other and therefore parallelly executable (c.f. Figure 6.7).

Based on this MapReduce concept, Sauer et al. [42] introduce a simple generalization of this model, allowing the reuse of established techniques for the evaluation of queries. The implementation of this suggested model for MxL 2.0's query processing would address the mentioned issue with the Hybrid wiki concept on the one hand, and enable a distributed processing of queries on the other hand.

6.7. Fully supported Living KPIs

As stated in Section 2.1, our group not only developed a uniform and configurable structure for the definition and documentation of EAM KPIs [12] as well as a catalog of 52 KPIs based on this structure [13], but also a design method for defining EAM KPIs [14] (c.f. Figure 2.4).

Since the Living KPIs prototype meets the requirements listed in Section 1.3, it is able to support all design steps – from the selection of EAM goals to the configuration of instantiated KPIs.

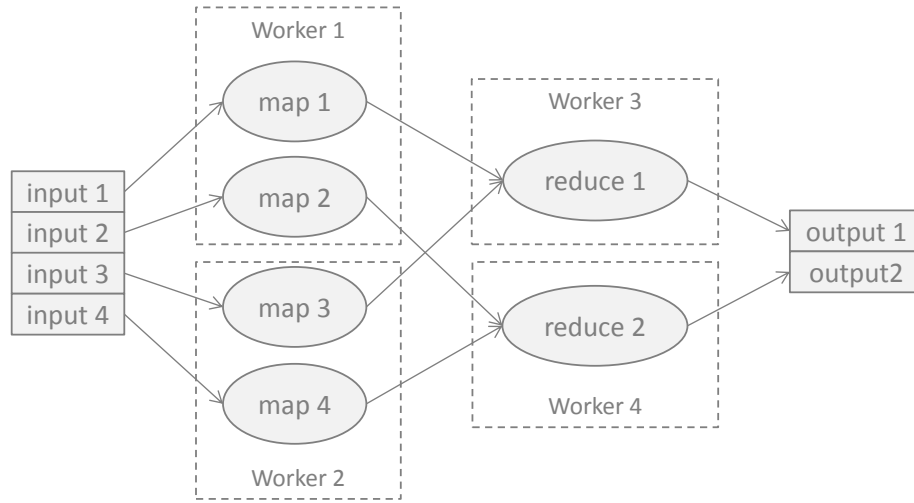


Figure 6.7.: An exemplary and simplified job execution in a MapReduce cluster [41, 42], assuming *map 1* and *map 3*, respectively *map 2* and *map 4* are emitting key-value pairs with the same key.

6.7.1. Problem / Open issue

As stated in Subsection 2.1.2, the EAM KPI structure consists, i.a., of organization-specific structure elements, e.g., the KPIs interpretation (specifying which values of the KPI are good, normal, or bad), tolerance values, target values, and planned values. However, all these values are “lifeless values” of the Living KPIs prototype, i.e., the interpretation of these values (e.g., are target values met, or is the current KPI value good) is not supported by the Living KPIs prototype and has to be done by the users. Hence, while the definition and adaption of EAM KPIs is supported by the Living KPIs, their interpretation is not.

6.7.2. Proposed solution

In order to implement a Living KPIs prototype supporting the definition and configuration EAM KPIs as well as their interpretation, the EAM KPI descriptions representing the EAM KPI structure cannot be implemented as an ordinary TypeDefinition, but as a built-in Tricia entity allowing the implementation of a specific behavior of the KPIs, e.g.:

- Automated adaption of the MxL expression representing the formal computation prescription when changing the KPI’s mapping table
- Automated interpretation of the KPI’s current value (e.g., as traffic light)
- Automated execution of defined processes in case of the KPI’s escalation
- Tool-supported analysis of the KPI’s evolution considering its planned, tolerance, and target values.

- Automated evaluation of the KPI according to its measurement frequency (c.f. scheduled evaluation strategy in Section 6.3).
- Automated dashboard provision for owners and/or consumers of the KPI

Bibliography

- [1] Sabine Buckl, Thomas Dierl, Florian Matthes, and Christian M. Schweda. Building Blocks for Enterprise Architecture Management Solutions. *The 2nd Practice-driven Research on Enterprise Transformation*, 2010.
- [2] Frederik Ahlemann, Eric Stettiner, Marcus Messerschmidt, and Christine Legner. *Strategic Enterprise Architecture Management*. Springer-Verlag, 2012.
- [3] Florian Matthes, Sabine Buckl, Jana Leitel, and Christian Schweda. Enterprise Architecture Management Tool Survey 2008. *ISIS Business Integration Special, Nomina Informations- und Marketing Services*, 2008.
- [4] Robert A. Handler and Chris Wilson. Magic Quadrant for Enterprise Architecture Tools. <http://imagesrv.gartner.com/media-products/pdf/reprints/ibm/external/volume4/article28.pdf>, 2011.
- [5] Sabine Buckl, Alexander M. Ernst, Josef Lankes, and Florian Matthes. Enterprise Architecture Management Pattern Catalog (Version 1.0, February 2008). Technical report, Chair for Informatics 19 (sebis), Technische Universität München, Munich, Germany, 2008.
- [6] Sabine M. Buckl. *Developing Organization-Specific Enterprise Architecture Management Functions Using a Method Base*. PhD thesis, Fakultät für Informatik, Technische Universität München, Germany, München, 2011.
- [7] Christian M. Schweda. *Development of Organization-Specific Enterprise Architecture Modeling Languages Using Building Blocks*. PhD thesis, Fakultät für Informatik, Technische Universität München, Germany, 2011.
- [8] Sabine Buckl, Florian Matthes, Christian Neubert, and Christian M. Schweda. A Wiki-based Approach to Enterprise Architecture Documentation and Analysis. *7th European Conference on Information Systems*, 2009.
- [9] Florian Matthes and Christian Neubert. Wiki4EAM - Using Hybrid Wikis for Enterprise Architecture Management. *7th International Symposium on Wikis and Open Collaboration (WikiSym)*, 2011.
- [10] Josef K. Lankes. *Metrics for Application Landscapes*. PhD thesis, Fakultät für Informatik, Technische Universität München, Germany, München, 2008.
- [11] C. Lucke, S. Krell, and U. Lechner. Critical Issues in Enterprise Architecting - A Literature Review. *Proceedings of the Sixteenth Americas Conference on Information Systems*, 2010.

- [12] Florian Matthes, Ivan Monahov, Alexander W. Schneider, and Christian Schulz. Towards a unified and configurable structure for EA management KPIs. *Trends in Enterprise Architecture Research and Practice-Driven Research on Enterprise Transformation*, 2012.
- [13] Florian Matthes, Ivan Monahov, Alexander Schneider, and Christopher Schulz. EAM KPI Catalog v1.0, 2012.
- [14] Florian Matthes, Ivan Monahov, Alexander Schneider, and Christopher Schulz. A Template-Based Design Method to Define Organization-Specific KPIs for the Domain of Enterprise Architecture Management. *DASMA Software Metrik Kongress*, 2012.
- [15] Ivan Monahov, Thomas Reschenhofer, and Florian Matthes. Design and prototypical implementation of a language empowering business users to define Key Performance Indicators for Enterprise Architecture Management. *Trends in Enterprise Architecture Research Workshop*, 2013.
- [16] Andre Wittenburg. *Softwarekartographie: Modelle und Methoden zur systematischen Visualisierung von Anwendungslandschaften*. PhD thesis, Fakultät für Informatik, Technische Universität München, Germany, 2007.
- [17] Unified Modeling Language (UML), v2.4.1. <http://www.omg.org/spec/UML/2.4.1>, 08 2011.
- [18] Thomas Büchner, Florian Matthes, and Christian Neubert. Data Model Driven Implementation of Web Cooperation Systems with Tricia. *3rd International Conference on Objects and Databases*, 2010.
- [19] Florian Matthes, Christian Neubert, and Alexander Steinhoff. Hybrid Wikis: Empowering users to collaboratively structure information. *6th International Conference on Software and Data Technologies*, 2011.
- [20] Object Constraint Language, v2.3.1. <http://www.omg.org/spec/OCL/2.3.1/>, 01 2012.
- [21] Paolo Pialorsi and Marco Russo. *Programming Microsoft LINQ in Microsoft .NET Framework 4*. Microsoft Press, 2010.
- [22] Peter Van Roy and Seif Haridi. *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.
- [23] Michael Scott. *Concepts of Programming Languages*. Morgan Kaufmann Publishers, 2001.
- [24] Anders Heijlsberg and Mads Torgersen. Standard Query Operators Overview. <http://msdn.microsoft.com/en-us/library/bb397896.aspx>, 2013.
- [25] Matheus Hauder, Sascha Roth, Florian Matthes, Armin Lau, and Heiko Matheis. Supporting collaborative product development through automated interpretation of artifacts. *3rd International Symposium on Business Modeling and Software Design*, 2013.

- [26] Heiko Matheis. SmartNet Navigator and application guidelines. *Seventh Framework Programme*, 2013. SmartNets - The Transformation from Collaborative Knowledge Exploration Networks into Cross Sectoral and Service Oriented Integrated Value Systems.
- [27] JFlex - The Fast Scanner Generator For Java. <http://jflex.de/>, 01 2009.
- [28] Roger S. Scowen. Extended BNF – A generic base standard. *Software Engineering Standards Symposium*, 1993.
- [29] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers. Principles, Techniques and Tools*. Addison-Wesley Longman, 2006.
- [30] Beaver - a LALR Parser Generator. <http://beaver.sourceforge.net/>, 12 2012.
- [31] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [32] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.
- [33] elasticsearch - flexible and powerful open source, distributed real-time search and analytics engine for the cloud. <http://www.elasticsearch.org/>, 06 2013.
- [34] tinymce - Javascript WYSIWYG Editor. <http://www.tinymce.com/>, 08 2013.
- [35] Gilad Bracha and William Cook. Mixin-based Inheritance. *Proceedings of the European conference on object-oriented programming on Object-oriented programming systems, languages, and applications*, 1990.
- [36] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. YAML Ain't Markup Language (YAML) Version 1.1. <http://yaml.org/spec/1.1/>, 2005.
- [37] snakeyaml - YAML parser and emitter for Java. <http://code.google.com/p/snakeyaml/>, 01 2005.
- [38] Steffen Bartsch. Authorization enforcement usability case study. *Proceedings of the Third international conference on Engineering secure software and systems*, 2011.
- [39] Paul Andrew. Collaborative Workflow improvements in SharePoint 2010. *MSDN Magazine*, 11 2009.
- [40] Nahum Gershon and Ward Page. What storytelling can do for information visualization. *Commun. ACM*, 2001.
- [41] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 01 2008.
- [42] Caetano Sauer and Theo Haerder. Compilation of Query Languages into MapReduce. *Datenbank Spektrum*, 13:5–15, 2013.