University of Magdeburg

School of Computer Science

Master's Thesis

Design and Implementation of an Efficient Approach for Custom-fields and Formulas with SAP HANA

Author:

Molham Kindakli

July 20, 2015

Advisors:

Dr. Steffen Goebel
SAP SE

Prof. Dr. rer. nat. habil. Gunter Saake
M.Sc. David Broneske
Department of Data and Knowledge Engineering

Kindakli, Molham:
Design and Implementation of an Efficient Approach for Custom-fields and Formulas with SAP HANA
Master's Thesis, University of Magdeburg, 2015.

Acknowledgement

This thesis would never have been completed successfully without the help of following people.

First and foremost, I would like to thank my adviser Dr. Steffen Göbel, for his patient guidance during selecting of research topic as well as generous contribution of knowledge and valuable comments during the writing of the thesis.

I would also like to thank a second adviser Prof. Gunter Saake, who always promptly answered on all questions and helped with administration issues during the work.

I would like to give a special thank to M.Sc. David Broneske for his valuable remarks and suggestions during the work.

Furthermore, I must also express gratitude to my family, who continuously supported me and were very patient for my limited time during this work.

Contents

List of Figures

List of Tables

List of Code Listings

Glossary

AFL	Application Function Library
ANTLR	ANother Tool for Language Recognition
AST	Abstract Syntax tree
BNF	Backus–Naur Form
DOM	Document Object Model
ERB	Embedded Ruby
HTML	HyperText Markup Language
JSON	JavaScript Object Notation
MDA	Model Driven Architecture
MDD	Model Driven Development
MDE	Model Driven Engineering
MDX	Multidimensional Expressions
NFR	Non-functional Requirement
PLC	Product Life-cycle Costing
PLCUDF	Product Life-cycle Costing User-defined Formula
SQL	Structured Query Language

1. Introduction

1.1 Motivation

Product costing is an important field for any company, especially in the area of manufacturing. It helps the company estimate its product cost and manage the costing split among different production phases and various plants.

SAP is currently developing a new product for creating and managing product cost calculations based on SAP HANA database. The costs shall be calculated from the early stages of the product lifecycle. These calculations provide the customer with indicators about pricing to compete with similar products in the market.

This product should provide a general-purpose calculation to serve the diversity of customers. Therefore, the database schema cannot cover all the varieties of customers' needs. Whereas, each customer has custom costing calculations for a particular product. Moreover, the customers need and request to expand the calculation functionalities and define their own price calculations based on the provided fields or additional required fields.

The mentioned problem description implies the need for an efficient way to store the additional fields and an implementation of an effective method to perform the customer's calculations. The implementation strategy is based on the dynamic creation of database artefacts during the application runtime.

1.2 Goals and tasks

Goal of this Thesis

The goal of this thesis is to design an approach for the custom-fields and custom-formulas and then integrate an implementation with SAP HANA.
Furthermore, it is aimed to parse the user formulas and evaluate them on an optimized way within SAP HANA. Eventually a proof the concept is shown by implementing prototypes for both sections.

The tasks

With this thesis, we aim to provide the following:

- An investigation of the related work for generating database artefacts including stored procedures using model-driven approaches.
- A requirement analysis for code generation with JavaScript based on the custom-fields generation use case.
- A prototypical design for dynamic SQLScript generation and related database artefacts creation.
- A requirement analysis for custom formulas calculation based on an existing grammar for the formula language and the potential extensions.
- A prototypical design of an application function as interpreter for custom formulas in C++.

1.3 Product Life-cycle Costing (PLC)

Product development evolves and is subject to change, meanwhile the cost estimation over time is influenced by various factors such as the material prices and currency exchange rates.

Product life-cycle costing (PLC) is a new product powered by SAP HANA which aims to determine costs at an early stage of the product life-cycle to drive profit margins and reduce risks. These costs can be calculated manually; however it is challenging to keep the calculation history and often the time effort for the calculation is prohibitive. Additionally, the resulting data has a high potential for inconsistency.

PLC provides many features to calculate costs precisely and increase the success factor. It is designed to keep track of all the calculations and help the management to make decisions and develop future strategies. For instance, PLC supports pricing strategies in a way the customers can compare material prices with alternatives and provide competitive cost.

1.4 Example scenario

For better illustration, we use a running example throughout the thesis. In Table 1.1, we show a simple cost calculation for an item similar to PLC calculation. The cost of the item is calculated by summing up all the sub-item costs.

The aim of the thesis is to add additional columns dynamically based on customer specifications. For example, adding *energy consumption or carbon dioxide emissions* columns. Another task is to provide formulas as column values in addition to the literal values. Those formulas are based on both existing fields and custom defined fields. The formula is defined for one column and applies on each row according to custom conditions set by the customer.

For instance, if we have an additional column that represents the *energy consumption*, the energy equation depends on the *Manufacturing hours*.

$$Energy = Manufacturing\_hours * 3.5$$

Item			ID	Price	Quantity	Manufacturing _hours	Total Cost
Mobile			8437		1		257.5
	Screen		7501	20	1	25	30
	Speaker		7128	3	2	3	6
	Board		9734		1	23	212.5
		CPU	9045	100	1	62.5	100
		RAM	7934	25	1	4	50
		GPU	9200	15	1	4	15
	Camera		5233	2	1	7	2.5
	Screw		8794	0.5	15	1	6.5

Table 1.1: Calculation example

The materials in the calculation table have some meta-data other than price and quantity. There are also some header fields that are defined for all calculations related to a customer. These additional properties can be included in the formulas as well as in the conditions. In Table 1.2, we present a sample of these properties related to previous calculation.

1.5 Related work

There were two previous works before this thesis discussing the custom-defined formulas.

Item ID	**Material type**	**Plant ID**	**Cost element ID**	**...**
7501	A1	1000	652	
7934	A2	1000	634	
5233	B7	2000	541	
...				

Table 1.2: Material properties

In the thesis of [Kü14], the custom-formulas were expressed by a grammar using the ANTLR v4 parser generator. The prototype was developed independently of PLC using C#. To calculate the custom-formulas, a grammar corresponding to the input has been converted to a database query. The database query is generated on the client side, transmitted to and executed on the server. This work faces several basic problems. First, it is working independently of PLC. Second, the queries are generated on the client side and have to be considered as threats to database security.

In the same context, the thesis of [Rus15] used a JavaScript client-side parser PEG.js and a special kind of views as a generation target. These views are called Calculation Views. In [Rus15], the calculation views were generated in XML format. Those views are converted to SQLScript internally and executed by SAP HANA. This work was expected to have performance problems but due to SAP HANA optimization on the SQLScript execution, it has sufficient performance. However, there were some limitations at this approach:

- It cannot provide a sufficient, flexible solution because it is limited to SQLScipt functionalities. For example, it is only possible to define additional built-in functions or user-defined functions in case there is a corresponding SQLScript function.
- Another restriction is that the execution is done row by row and there is no possibility to store or access values on a different row.
- The error reporting during the execution is also limited to SAP HANA errors. There is no way to define custom error messages.
- Finally, in case an error occurs the execution for all view fails and the trace does not report in which row the error occurs.

1.6 Structure

Structure of the Thesis

Apart from this chapter, our thesis comprises five further chapters as follows:

In Chapter 2, we give fundamental basics about Model-driven engineering and templating systems, and provide technical overview of the used technologies.
In Chapter 3, we consider different choices of templating systems and propose our approach to dynamically generate custom-fields and the related database artefacts.
In Chapter 4, we propose our approach to implement an interpreter to evaluate custom-formulas and conduct an example through the chapter to explain the different stages.
In Chapter 5, we provide test cases related to custom-formula evaluation beside, performance evaluation for the interpreter to compute parsing time and execution time.
In Chapter 6, we summarize the outcome of the thesis and our contributions, and we present future work.

2. Background

In this chapter, we introduce background knowledge and the main foundations on which our approach relies. We describe the software engineering concepts related to model-driven engineering and template engines at the first part of the chapter in Section 2.1.

In the second part in Section 2.2, we present the related technical details of SAP HANA database. We start by briefly describing the database architecture and give more detailed description of the database parts related directly to our thesis.

2.1 Fundamental basics

Model-driven Engineering (MDE) is a software development approach which focuses on creating models, or abstractions. Model-Driven Architecture (MDA) is a standard implementation of the MDE by Object Management Group (OMG) which defines three levels of modeling abstraction. In this context, the Model-Driven Development (MDD) is a new software development pattern for creating software using the concepts of models and model transformations. The application of the Model-Driven Development could improve the quality of software development and increase the flexibility in adaptation to different technologies. It also shortens the ramp-up time of products. [Lon10]

2.1.1 Model-driven engineering

MDE is a software development approach that is used to raise the abstraction level of system modeling and increase the development automation to alleviate complexity. This kind of abstraction raises the productivity by increasing the compatibility between systems and simplifying the design process. This abstraction leverages simpler models with problem based focusing.

MDE supports automated transformation to convert the system model into another representation such as source code. Model transformation allows moving from an abstract meta-model to a model representing the implementation of the system. To ensure the quality of the model, MDE uses model validation, model checking, and model-based testing. [CH06]

Model-driven development

MDD is a standardized methodology for development approaches. MDD focuses on modeling rather than developing programs which leads to concepts closer to the problem domain and not bounded to an implementation technology [Sel03].
In MDD, code is generated automatically from abstract models. MDD also supports model-to-model transformations [BIJ06].

Model-driven architecture

MDA was proposed by Object Management Group (OMG) applying MDE as a standard and working efficiently for tools and models. MDA uses UML standard in development model driven software. MDA increases development productivity by maximizing compatibility while transforming between the models. In Figure 2.1 [1], we illustrate the relation between the different concepts.

MDA defines the guideline specifications in terms of models and their transformations. OMG has defined Query View Transformation (QVT) which is a standard language for model transformation.

MDA separates between the system functionality and its implementation. It has three levels of abstraction based on the model that is considered the main structure unit. This provides flexibility and reusability during the implementation targeting different platforms.

Model transformations

Model transformations allow converting models from one representation into another [SK03], so that one model conforms to a meta-model. The model is an abstract representation of the structure, functionality and behavior of a system. Whereas the meta-model is a model characterizing the structure, semantics and constraints of other models [SV05]. It is written in a well-defined language that defines the abstract syntax of modeling languages. M2T (model-to-text) transformation is one type of model transformation that converts abstract models into textual representations of particular programming languages [CH03]. M2T transformations are template-based approaches using predefined textual templates, which contain slots that point to particular model entities and are substituted by an appropriate generator engine with corresponding content of input models [Loc10].

M2T examples:

[1]This figure is based on http://modeling-languages.com/clarifying-concepts-mbe-vs-mde-vs-mdd-vs-mda/

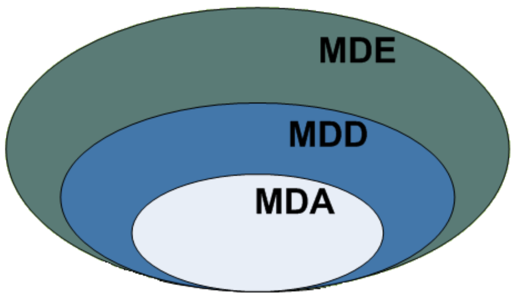


Figure 2.1: The relation between MDE, MDD, and MDA

- **Acceleo** is an open source code generator for Eclipse used to generate textual languages.

- **Velocity** is a Java-based template engine, introduces a powerful template language with reference to objects in Java code. It includes placeholders filled with content during the rendering process.

- **FreeMarker** is a Java-based template engine, supports the construction of web pages with dynamic content.

- **RISE** is a software suite for MDD, generates the code and then integrates the generated solution with a development environment.

- QVT, MOFScript, openArchitectureWare, JET, EMF ...

2.1.2 Template engines

The template-based approach is an M2T transformation that uses a transformation engine for one single domain that can produce complex structures of the model corresponding to this specific domain [Loc10]. The template engines are used to combine structured templates with input data to get the result document. The templates are usually text documents written by templating language holding placeholders that are replaced by actual input arguments using the template processing system or simply the template engine [ABvdB07]. The templating system consists of three elements:

- The template

- The data model

- The processor

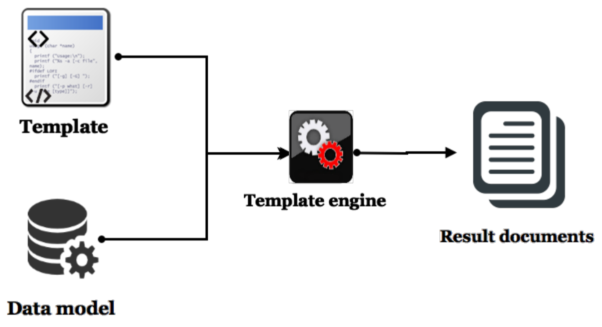


Figure 2.2: Processing flow of a template engine.

Usage

Different types of output can be produced for different purposes such as: source code generation, automatic emails, and web applications. The most common template engines are web template systems that generate web pages. Another type is code generation tools which use standard modeling languages such as UML to generate the source code of a specific language. In general, the template engines use an evaluator to generate a text document from a data model. In Figure 2.2 we show the evaluation process.

> A template-based approach is used wherein a template specifies a text template with placeholders for data to be extracted from models. [OMG08]

Template engines are used mainly to generate dynamic documents [Par04], which in turn simplify development and improve system flexibility. They can also help with reusing components during the development and reduce maintenance costs. With the goals of efficiency, implementation transparency, security, and standards compliance in mind. The main goal of using template engines is to organize the code and make it more readable, extensible and to separate the presentation from the business logic [TS09].

Comparison

There are many template engine solutions sharing common properties. However, we can differentiate between them using several criteria. Each template engine is based on a model, either textual or graphical. The structure of the template and what will be produced are the main criteria for choosing a template engine. Addressing system security is very important to prevent security holes. Especially when using web template systems which have been vulnerabele to SQL injection in the past [LX14]. Eventually, different solutions can be distinguished by their ability to support template testing and validation in order to prevent and track errors [Ber07].

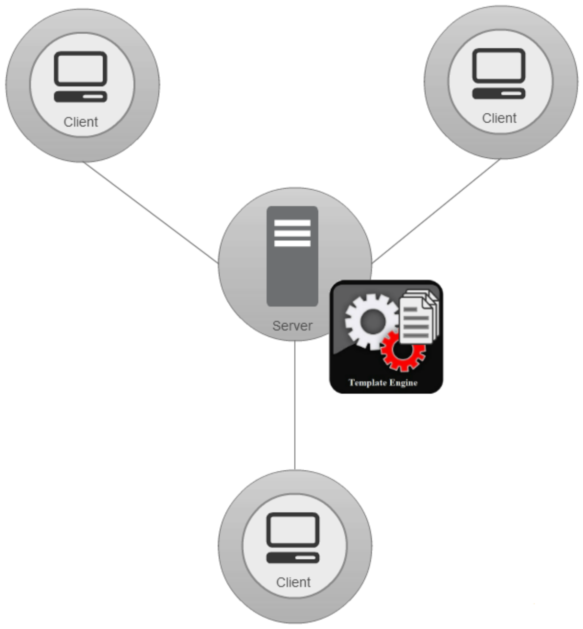


Figure 2.3: Server-side template system

Kinds of web template systems

On client-server architecture we can identify different kinds of template systems by where the template is combined with the data model.

1. **Server-side systems**
   The template substitution occurs on the server side. The template system generates the result documents on the server and sends them to the clients Figure 2.3. The same procedure is used in script languages such as: Active Server Pages (ASP), JavaServer Pages (JSP), and PHP. Many other t§emplate systems are generally used as server-side template systems: Mustache, XSLT, FreeMarker, and Smarty.

2. **Edge-side systems**
   The template substitution occurs on an intermediary between the server and the client Figure 2.4. The main purpose of this separation is to reduce the network traffic and the server processing by having many Edge-side servers and caching the result content in them. Edge Side Includes (ESI) is a special markup language for edge level dynamic web content assembly that is used on the Edge-side server [Tec15].

3. **Client-side systems**
   The template substitution occurs on the client side Figure 2.5. The server transfers the template and the data model to the client then using client-side scripting

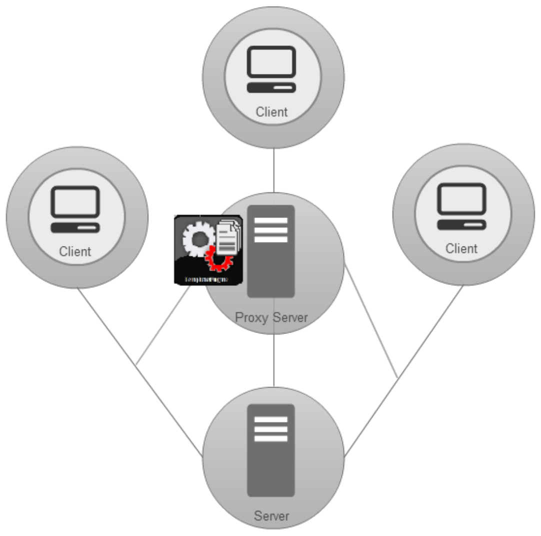


Figure 2.4: Edge-side template systems

language such as Javascript. The result documents can be generated and displayed directly. Closure Templates, EJS, and Pure. are examples of client-side templating systems.

4. **Static page generators**
   This kind of templating system creates static pages. The template language is used only once while creating the pages. Then they are published on the server as static content Figure 2.6. This can be used on content management systems (CMS) such as Wordpress as a feature to speed up the system.

2.2 Technical background

The following sections describe the technical environment used within the thesis. First of all, this includes the SAP HANA database. The subsequent sections provide a closer look at basic parts, the Extended Application Services and the Application Function Library.

2.2.1 SAP HANA database

SAP HANA is an in-memory data platform for relational database management systems that supports multiple cores as well as multiple CPUs and large main memories to provide parallel processing capabilities and thereby improves the performance [Wal12]. SAP HANA database supports two data storage layouts, the column-based store for OLAP support and the row-based store for OLTP operations.

SAP HANA includes various engine types. The main SAP HANA database management component is the *index server*, which contains the actual data storage and many

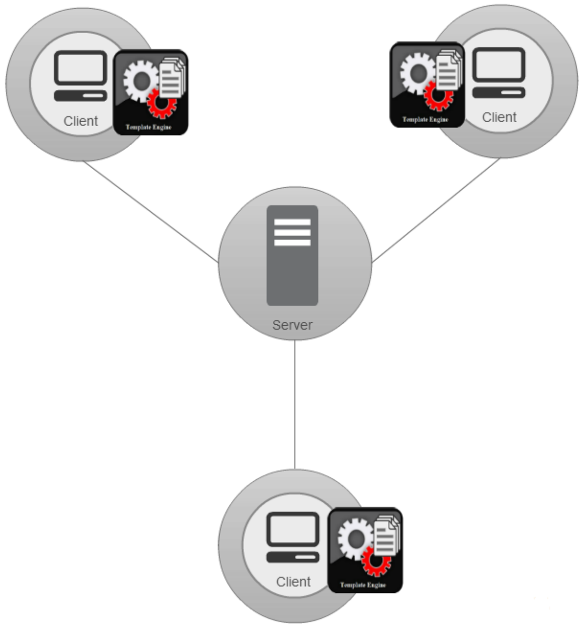


Figure 2.5: Client-side systems

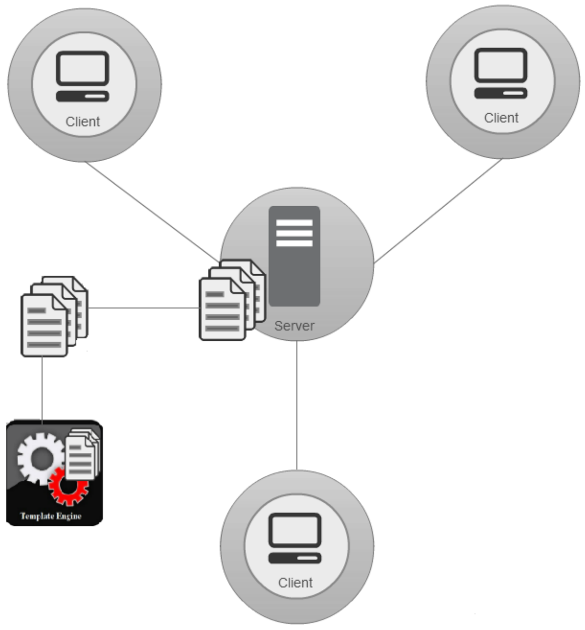


Figure 2.6: Static page generators

engines for processing the data such as the processing engine, the calculation engine, and the execution engine [FM12].

- The processing engine is responsible for managing and converting the data storage between column-based and row-based layout.
- The calculation engine transforms the query from different interfaces such as standard SQL, SQLScript and MDX into execution plan to be optimized and executed.
- The execution engine uses the processing engine to execute the queries represented by execution plans and send the result back.

In Figure 2.7 we show a simplified architecture of SAP HANA database divided into different layers. Further details about SAP HANA are beyond the scope this work, for detailed information, it is advised to check [MV13, PL15] and [Wal12]. The used scripting language for SAP HANA is SQLScript as well as the standard SQL. SAP HANA also supports a framework for the installation of specialized functional libraries which can be called directly from within SQLScript. These application function libraries (AFL) are integrated with index server's engines.

SQLScript

SQLScript is SAP specialized scripting language used in SAP HANA database which is based on a collection of extensions to SQL. The extensions are [sql11]:

- Data extension allows the definition of table types without corresponding tables.
- Functional extension allows the definition of functions which can be used to express and encapsulate complex data flows.
- Procedural extension provides imperative constructs executed in the context of the database process.

SQLScript is designed to move the logic from the application layer into the database to provide optimization possibilities and avoid copying the data to the application server and for leveraging complex parallel execution strategies of the database.

2.2.2 Extended application services (XS)

Extended application services (XS) engine is both a lightweight web server and the application server of SAP HANA platform. It handles the control flow logic and provides a set of services that support web-based applications. This includes OData support, server-side JavaScript execution, and full access to SQL and SQLScript. In Figure 2.8 we show various services supported by the XS engine and the communication between the client and SAP HANA via the XS engine.
Accordingly the XS engine is an application development platform for building and deploying applications. It manages the HTTP requests from the client to transmit the data.

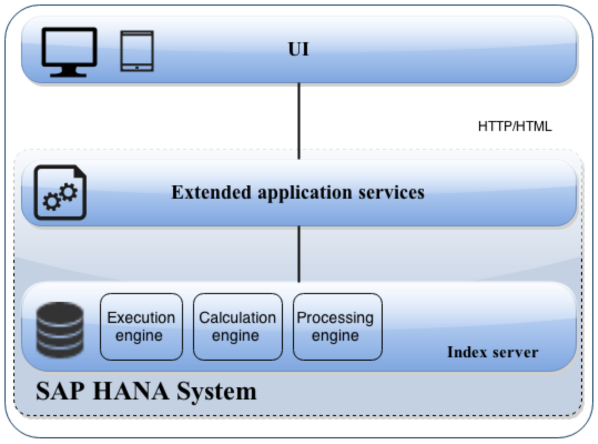


Figure 2.7: SAP HANA simplified architecture

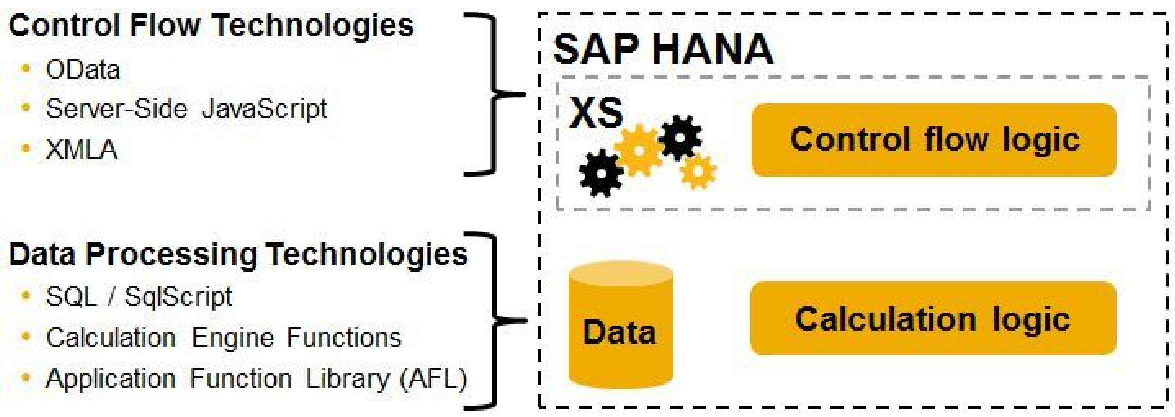

Control Flow Technologies
OData
Server-Side JavaScript
XMLA
Data Processing Technologies
SQL / SqlScript
Calculation Engine Functions
Application Function Library (AFL)
SAP HANA
XS
Control flow logic
Data
Calculation logic

Figure 2.8: SAP HANA technologies[Sil15]

2.2.3 Application Function Library

Application function library (AFL) is creating extension libraries for the kernel of SAP HANA database using C++. These libraries can be installed into a SAP HANA instance. Subsequently, the functions provided in the library are available to the database users, similar to the SAP HANA SQLScript procedure.

These libraries can be integrated directly in the project development, speeding up projects by avoiding writing custom complex algorithms from scratch. AFL also offers very fast performance, as AFL functions are running in the core of SAP HANA database.

3. Custom-fields code generation

In the previous chapter we introduced the necessary background knowledge to understand this thesis. In this chapter we present an overview of the issue that we want to solve in Section 3.1. In Section 3.2 we analyze the implementation target of our approach. Then, we propose our approach to generate SQLScript code using a template processor. In Section 3.3 we list the system requirements and categorize the templating system solutions in Section 3.4. In Section 3.5 we enumerate the applicable templating solutions and give a comprehensive compression. Finally we present in Section 3.6 a proof of concept representing the integration of a template processor within the SAP HANA environment.

3.1 Overview

The PLC has a database schema containing a limited number of tables and columns. Since it is targeting different varieties of customers, it is not possible to include everything required. Our proposed approach is to generate custom-fields code dynamically which may leads to create or update other database artefacts such as tables, columns, views, or procedures. Thus, an efficient way to store the model for additional columns and tables as well as building a variable data scheme is required.

3.2 Selecting the implementation target

There are many possibilities to achieve extensibility and generate the custom-field columns in relational databases. However, we are looking for an approach that is applicable during the application run-time. Moreover the custom-fields code generation has an impact on other database artefacts that should be generated or updated accordingly.

SAP HANA provides a database access and SQLScript commands execution through XS Engine. XS Engine uses server-side JavaScript and allows the native development capabilities. It gives the possibility to build and execute SAP HANA XS applications on the server side using JavaScript. Those applications are tightly integrated with the database and run directly on SAP HANA to perform the generated statements.

As a result, we consider using the XS engine as our implementation target and focus our investigations on the web templating systems that can be used within the XS engine.

3.3 Requirement analysis

The following sections aim at analyzing the requirements of choosing the templating system and building a solution based on it. We divide the obtained requirements into *functional* and *non-functional* requirements. *A functional requirement* describes a desired function of the templating system that considers a specific input into the system, performs one or more operations in response to the input and creates an output from the system. In contrast, *a non-functional requirement* describes the characteristics of these functions. The last subsection summarizes the requirement analysis by providing a list of the obtained requirements.

3.3.1 Functional requirements

As initially stated, generating database artefacts such as columns, tables, views and procedures dynamically in SAP HANA during the application run-time is the main requirement **(FR-1)**.

Another requirement is to store the template at the database **(FR-2)**. The templating system requires a model that can be stored in the database as meta-data. In furtherance of forming the SQLScript commands, the relevant template information should be stored as well.

The third requirement is to integrate the approach with SAP HANA system and the PLC. In order to provide a flexible solution with dynamically generated artefacts and compatible with variant customers **(FR-3)**.

3.3.2 Non-functional requirements

Non-functional requirements define the system attributes and place restrictions and constraints on the system and the development process. Based on IEEE-Std 29148:2011 [Sta12] non-functional requirements might be split into quality requirements and human factor requirements.

Quality requirements should be tailored to the system being developed. Measuring the quality requirements should be included as well. Such as: flexibility, portability, reusability, reliability, maintainability, and security.

Human Factors Requirements Define the required properties for the interaction with the system users. Such as: safety, performance, effectiveness, efficiency, reliability and maintainability.

To compare and evaluate different web templating systems, the definition of evaluation criteria, as well as an evaluation measurement are necessary. In the following, we will illustrate the NFRs that are elicited and documented based on SAP product qualities and interviews with the PLC product manager [AACF12]. Each constraint leads to one or more adequate requirements, the Table 3.1 contains NFRs definitions along with their priorities.

It is very important to have a secure solution (**NFR-1**) to eliminate any unauthorized access and protect the system from accidental damage. Therefore the templating system should prevent SQL injections and other malicious damage. Moreover the access permission is granted from the XS engine while generating the adequate artefacts.

The templating system should be able to reuse parts of the template to generate other artefacts (**NFR-2**) and this is for two reasons: reducing the repetition, and improving the system maintainability by making it less risky to update the code.

Each template contains placeholders in addition to the logical statements such as loops and conditions, hence well-organized templating language makes it easy to figure out the template and to update it later, which infers to (**NFR-3**).

Another constraint is using the templating system and integrate it with SAP HANA without spending a lot of time in learning the templating language, taking this constraint into account by choosing a solution with short ramp-up time and productivity (**NFR-4**).

Furthermore, the rendering process to generate the dynamic artefacts code should be done fast so the produced code is executed in HANA (**NFR-5**).

Basic validations and error tracking are required to increase the efficiency of the system (**NFR-6,7,8**).

Meanwhile, SAP is providing a commercial product that must be supported continuously, any external system integrated with SAP HANA must have well-organized documentation and supportive community to communicate with the system developers in case there is any bug and to keep the system up to date for the prospective development.(**NFR-9,10,11**).

The execution in SAP HANA server demands an additional restriction that the templating system must work as standalone solution and does not require any external dependencies while running (**NFR-12**).

It is very important for SAP to use public-licensed solution that allows commercial usage (**NFR-13**). One example of those open source public licenses is the MIT license which includes the following section: ”the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software” [MIT]

3.3.3 Requirements summary

Based on the requirements, we are looking for a web templating system that works on the server-side and renders JavaScript template to generate SQLScript code. The

ID	Name	Description	Priority
NFR-1	Security	”Specify the requirements to protect the software from accidental or malicious access, use modification, destruction, or disclosure”[Sta12]	High
NFR-2	Reusability/DRY	The support for reusing code and reducing repetition	Low
NFR-3	Readability	How easily can the structure of a template be distinguished and understood what it will produce.	Medium
NFR-4	Ramp-up time	The shape of the learning curve using the templating system language and how fast building things can be done later on.	Medium
NFR-5	Performance	The execution time while rendering a template	Low
NFR-6	Ease of Debugging	The possibility to track down errors while the templating system is running	Medium
NFR-7	Validation	The validations that can be done at compile-time to prevent errors at runtime	Medium
NFR-8	Testable	The ability to write unit and integration tests	Medium
NFR-9	Documentation	The available description about the templating system	High
NFR-10	Community	The active community using the templating system	Medium
NFR-11	Maturity	The time period since the last stable version of the templating system is produced	Medium
NFR-12	Library agnostic	The dependencies on other libraries	Medium
NFR-13	License	The copyright obligation on using the templating system or parts of it	High

Table 3.1: Non-functional requirements of the tempating system

generated statements will be executed to produce the custom-fields and the related database artefacts.

Additionally, the templating systems must be integrated with the PLC but work independently. Furthermore, we should ensure simplicity and avoid security vulnerabilities.

3.4 Template engines classifications

From now on, we will refer to templating system as template engines. Template engines classifications can be divided into two main categories: syntax and logic. Throughout this section, we explain in detail how the classification is performed.
As stated in Section 2.1.2, template engines use templating languages to express the syntax which in turn defines the structure of the template.

Our approach works in application server layer of SAP HANA; this layer uses JavaScript to build the applications. We can confine our classification of the different web template engines with those using JavaScript as templating language.

3.4.1 Template logic

The degree of the permitted logic supported by the templating language varies among the template engines. Some templating languages do not support logic while other templating engines allow logic to control the structure of the generated document. In contrast to pure JavaScript templating engines that use JavaScript to generate the documents, these kinds of templaing engines are compared below [1]:

- **Logic-less template engines** do not allow inline JavaScript code in the template. In this type the template engine is responsible for the view and it is separated from control. However, the template engine can control the logic within the template by using special directives to express loops and conditions.
  - Mustache.js
  - Hogan.js
  - Handlebars
  - Dust.js
- **Embedded JavaScript template engines** allow to embed JavaScript code directly within the template.
  - Underscore.js
  - Jade
  - doT
  - EJS

[1]based on http://psteeleidem.com/the-JavaScript-templating-landscape

3.4.2 Template syntax

From another perspective, the syntax defines the structure of the template. The template engines can be broken down into three types based on how the syntax is formed, including the following categories [2]:

- **Text-based** A text-based templating language has a custom parser to recognize template directives to generate text documents. Examples of text-based templating languages are:
  - Dust
  - Mustache
  - Handlebars
- **Markup-based** which recognizes templating directives inside the templates. The directive types can vary individually. HTML-based templating language uses the HTML elements as directives. Those directives are translated into HTML code and generating proper web pages. Whitespace-based template engines organize the generated documents on the whitespace, for example:
  - Marko
  - Haml
  - Jade
- **DOM-based** A DOM-based templating language depends on DOM tree which represents the logical document structure to recognize templating directives. It produces the DOM tree that is rendered in the browser so it is basically working on the client-side or on virtual DOM on the server-side. Examples of DOM-based templating languages are:
  - AngularJS
  - Knockout
  - Weld

3.5 Template engine selection

After we listed our requirements and explained the template engine classifications, we will enumerate the most common server-side JavaScript template engines, analyze their features, and pick the best template engine that meets our requirements. Then, we will expose our approach using the picked template engine.

[2] based on http://psteeleidem.com/the-JavaScript-templating-landscape

3.5.1 Listing the potential template engines

There are many web template engines using JavaScript as templating language and sharing common features. The template engine selection is based mainly on Linkedin research where they picked a JavaScript templating solution and moved from Java Servlet to JavaScript template engine [Lin].

We came up with a list of 10 web template engines. In the following, we mention and cite an overview of the template engines to be examined:

Mustache.js is a logicless, text-based template engine which defines a standard templating language used in many other template engines. It does not support any logic statement and can be used to generate any text file. [Leh15]

Jade.js is a whitespace-based template engine that is used for writing HTML. It allows JavaScript code to be embedded in the template. But, it is not suitable for non-HTML output. [Lin15]

Handlebars.js is a logicless and text-based template engine, it is based on mustache syntax but it is extended to offer other directives supporting conditional statements.[Kat15]

Hogan.js is a logicless and text-based template engine that is also based on mustache syntax. It supports compiling method to save a compiled template in order to fast rendering and having fast performance.[Inc15]

Underscore.js is a utility library for JavaScript that provides many built-in functions. It also provides simple text-based template engine with ERB syntax to embed JavaScript code. [Ash15]

EJS is a file-based template engine that also uses ERB syntax. The JavaScript code is embedded within special directives and evaluated in-place. [Con15]

Marko.js is an HTML-based template engine. Its syntax and its logic are embedded within HTML directives. It is not suitable for non-HTML templates. [PSI15]

Dust.js is a logicless and text-based template engine, which has the basic conditional operators. It supports composable templates from partial includes and dynamic template blocks.[Cor15]

Markup.js is a logicless and text-based template engine. It is based on mustache syntax. It supports conditional statements with many flexible options to manage the control over the template.[Mar15]

doT.js is a text-based template engine that supports including partial templates. It also supports delimiters customization and conditional operators. [Dok15]

3.5.2 Criteria matrix

After taking the requirements to generate the SQLScript code in consideration, we have the following main criteria summarized in Table 3.2 to choose a template engine from the previous candidates. This table is based on [Lin] research and has been updated to fit our requirements. In Chapter A, we have more detail about how Linkedin filled the criteria table.

	Mustache	Handlebars	Dust	Underscore	EJS	doT	Markup	Hogan
DRY	4	4,5	4,5	3	3	3	3	4
Performance	2	4	5	5	3	4	3	3
Ramp-up time	4	4	4	4	3	3	3	3
Ramp-up productivity	4	4	4	4	3	3	4	3
Community	4	4	3,5	5	2	2	1	3
Library agnostic	4,5	4,5	4,5	4	4	2	2	4,5
Testable	3	3	3	4	3	3	3	3
Ease of debugging	3	3	3	3	3	3	2	3
Maturity	4,5	3,5	3	4	2	1	3	3
Documentation	4	4,5	4	3	2	1	3	4
Code documentation	3,5	3,5	3,5	3	3	3	3	3,5
License	MIT	MIT	MIT	MIT	MIT	MIT	MIT	Apache v2.0
Total	41	43	41	42	31	28	30	38

Table 3.2: the evaluation criteria

For reasons of clarity, Figure 3.1 is established based on Table 3.2. We focus on the most important requirements.

After going through the evaluation process, we had four candidates with the highest ratings: Mustache.js, Handlebars.js, Underscore.js, and Dust.js. We prioritize the requirements and list the detailed specifications for these candidates:

Mustache.js

+ Popular choice with a large, active community.
+ Clean syntax leads to templates that are easy to build, read, and maintain.
- A little too logicless. Just for doing the basic tasks. It is difficult to apply our work with the current logic.

Handlebars.js

+ The same syntax as **Mustache**, so the templates are easy to build, read, and maintain.

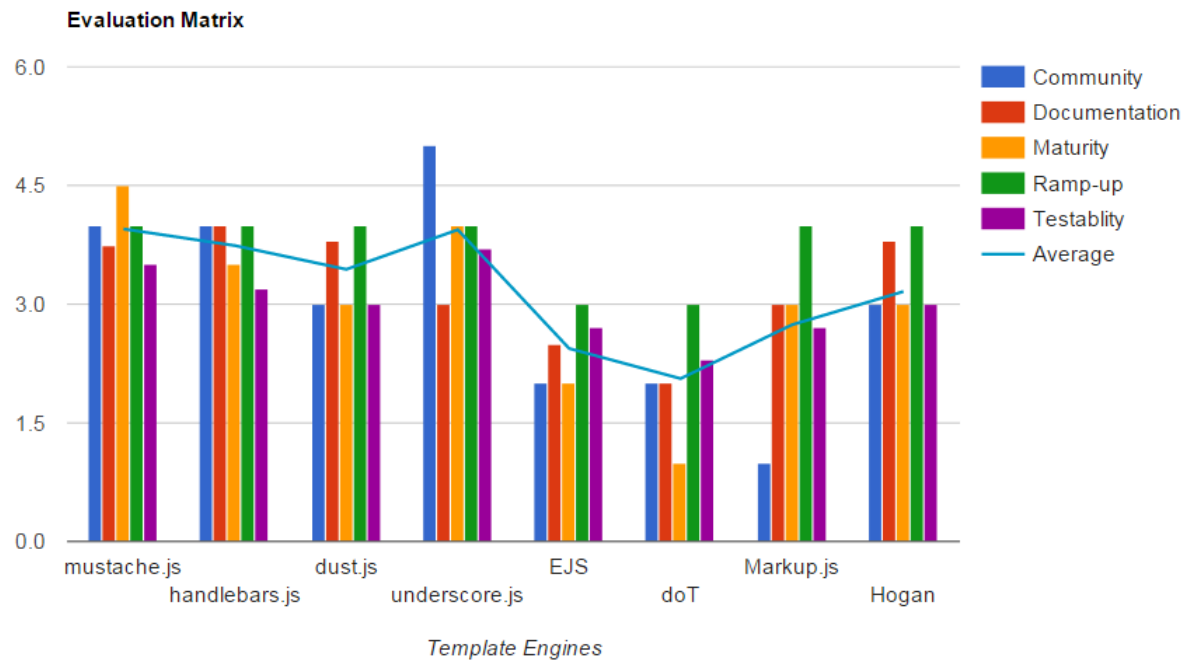


Figure 3.1: The evaluation criteria

+ The templates are compiled rather than interpreted templates which make it faster.
+ Better support for going deep through the data model.
+ Allow to define custom directives called *helpers*.
+ Allow template partials for code reusing.

Underscore.js

+ Very popular choice with a large, active community.
+ Provides a lot of built-in functions.
- The syntax is more complex than other templating languages.
- Required a source code modification to work with SAP HANA.

Dust.js

+ The same syntax as **Mustache**, so the templates are easy to build, read, and maintain.
+ The templates are compiled rather than interpreted templates which make it faster.
+ Better support for going deep through the data model.
+ Allow to define custom directives called *helpers*.
+ Inline parameters support.
- Maintainer of *Github* repository is not responsive.
- Required HTML root object otherwise the engine code should be updated to work properly with SAP HANA.

We choose *Handlebar.js* as our template engine from the previous list because of the following:

- It does not require any change to work with SAP HANA XS engine.
- There is no dependencies on DOM or HTML directives while parsing the template.
- It is able to generate the SQLScript code as a plain text.
- The software license allows commercial use.
- It has good maturity and high community support.

3.6 Contribution

Custom-field generation is the main purpose of our approach. In this section, we show how to use the web template engine to generate the custom-fields generation code or any other related database artefacts. In particular, we show an implementation of a use case, using the chosen template engine, integrated with SAP HANA.

To create those artefacts, we have the following components shown below:

- **The library manger** imports the JavaScript library into the XS and provides a handler to use its functionalities. The library manger is also responsible for logging states while executing a template. Moreover using the library manger, we can define a new helper that provides additional functionalities.
- **The database manger** stores the different templates and data models and manages the dependencies between the templates in addition to the execution of the SQLScript commands and the creation of the artefacts.
- **The process manager** is responsible for rendering and compiling the templates. It verifies and reports the errors as well.

In Figure 3.2 we illustrate our approach in a sequence diagram. We show the system components and the processes order. The process manager initializes the custom-fields code generation and proceeds with the different stages till the SQLScript code is generated. The database manager executes the code statements, creates or updates the associated artefacts and stores them at the database.

We will give a code example to illustrate our approach on how the template engine can be integrated with SAP HANA and fulfill the functional requirement by generating a dynamic SQLScript code. The following sample creates a dynamic database view based on a custom data that is stored in a meta-data table.

The database manager stores the data model in the database. During the execution the process manager requests the data model. The data model should be converted into JSON by the process manager in order to use it from the template engine.
In Listing 3.1, we have a JSON object that holds the data for creating the view such as:

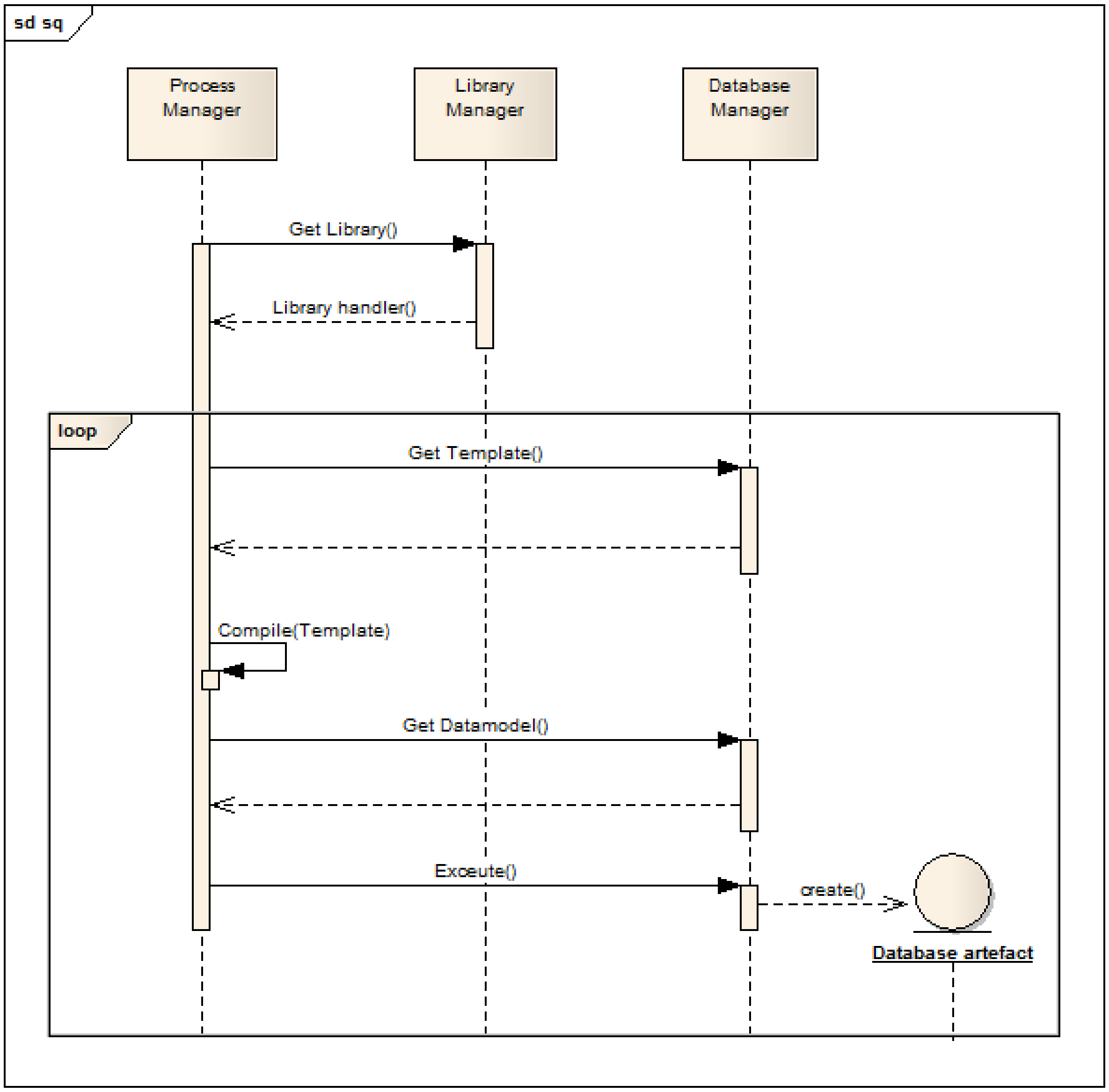


Figure 3.2: Sequence diagram of custom-field code generation

- The generated view name at line 3.
- The data tables and the columns that will be included in the result which are represented as array of objects at line 4.
- And the SQL conditions for creating the view at line 12 and 13.

Listing 3.1: The input context variable

```
 1 var context = {
 2  "view" : {
 3  "id" : "v_temp",
 4  "tables" : [ {
 5    "name" : "\"t_item\" i",
 6    "columns" : [ "name", "parent_id" ]
 7  }, {
 8    "name" : "\"t_item_extra\" e",
 9    "columns" : [ "extra_col1" , "extra_col2" ]
10  } ],
11
12  "join_condition" : ["i.\"id\" = e.\"id\"" ]
13  ,"additional_condition" : ["i.\"parent_id\" = 39465 " , "extra_col1 < 231"
       " ]
14  }
15 };
```

The template is stored in the database and defined as text template containing place holders. The process manager requests it from the database manager. In Listing 3.2, we show the template code of creating a view. The syntax is based on the SQLScript syntax rules to create view. The lines 4-12 will produce the *select* part of the create statement. The line 13 will produce the *from* for table selection. And finally the line 14-17 will produce the conditions.

Listing 3.2: Template code for generating dynamic database view

```
 1 var template =
 2 " CREATE view
 3    {{view.id}} AS select
 4    {{#view.tables}}
 5    {{#if @last}}
 6       {{#columns}} {{#if @last}} \"{{.}}\"
 7                {{else}} \"{{.}}\" , {{/if}}
 8       {{/columns}}
 9    {{else}}
10       {{#columns}} \"{{.}}\", {{/columns}}
11    {{/if}}
12    {{/view.tables}}
```

```
from {{#view.tables}} {{#if @index}}, {{/if}} {{name}} {{/view.tables}}
where {{#view.join_condition}}
    {{#if @index}} AND {{/if}} {{.}}
    {{/view.join_condition}}
  {{#view.additional_condition}} AND {{.}} {{/view.additional_condition}}";
```

In Listing 3.3 we show the JavaScript code after importing the *handlebars* as external library into SAP HANA XS engine. First, the process manager compiles the template which means to parse and translate it. And then the process manager renders the template by applying the actual data and generating the result. The rendering is done by replacing each placeholder with the actual provided data from the data model and executing the templating logic if there is any. Eventually, the process manager sends the generated statements to the database manager, which in turn executes those statements and generate the artefacts.

Listing 3.3: Tempalte engine execution

```
var compiled_template = handlebars.Handlebars.compile(template);
var sql = compiled_template(context);
var conn = $.db.getConnection();
var ps = conn.prepareStatement(sql);
   ps.execute();
   ps.close();
```

In the Listing 3.4 we display the result after the template engine rendered the template. The result is a SQLScript command which can be executed in SAP HANA to generate the desired *view* that will be stored permanently in the database. The first line is containing the name of the generated *view* and the name of the columns at this *view*. The second line is containing the table names. And the third line has the conditions.

Listing 3.4: The generated SQLScript code

```
CREATE view v_temp AS select "name", "parent_id", "extra_col", "extra_col2"
from "t_item" i , "t_item_ext" e
where i."id" = e."id" AND i."parent_id" = 39465 AND extra_col1 < 231
```

This *view* is created dynamically by the template engine based on various column names. The used columns might be custom-fields that are generated at the same way. The database manager stores those the dependencies and the process manager execute the template on a provided order to keep the database consistency.

4. Designing and implementing a calculation interpreter of custom-defined formulas

In this chapter, we propose an approach to evaluate the custom-defined formulas using a special Application Function Library (AFL). We present the system requirements in Section 4.2. The formulas are written using the specification of PLCUDF language which is described in Section 4.3. In Section 4.4, we discuss selecting the AFL as our generation target. In Section 4.5, we present different approaches for the evaluation and in Section 4.6, we present our contribution.

4.1 Overview

The PLC allows the user to calculate the product costing from different costs of product's materials. Those costs are either literal values that can be computed directly or custom defined formulas that need to be evaluated to obtain the result values. For having a standard language that looks familiar for PLC customers, the formula language is defined as Excel-like formula.

For illustration, we will use the introduction example described in secSection 1.4. We assume that a customer has the following equations for computing the energy consumption and carbon dioxide emissions.

$$Energy: \begin{cases} E1 = Manufacturing\_hours * 3.5 \\ E2 = Manufacturing\_hours * 3.5 + plant\_distance/2.7 \end{cases}$$

$$\text{CO2}: \text{ C} = \sqrt{Energy} * 0.05$$

ID	Pre-conditions			
	Plant_ID	Material_ID	Cost_element_ID	Working_center_ID
E1	1000	2342	652	
E2	2000		634	22
C		1290	541	

Table 4.1: Preconditions of the formulas

Many equations can be defined for the same field. However, those equations are computed only when the values from Table 4.1 equal the corresponding values of the evaluated row. We check the matches between Table 4.1 from one side and Table 1.1, Table 1.2 on the other side. Those values form the calculation preconditions.

The preconditions are conjunctions of item's properties where the empty value means no precondition is applied.

Our contribution is to implement an interpreter evaluator, that takes the formulas as input strings, compiles them on the fly, and computes the results. For this, we have to build an interpreter in the AFL that checks the correctness of grammar, and then generates the appropriate results.

4.2 Requirement analysis

The following sections aim at analyzing and specifying the requirements of the formula interpreter. A functional requirement describes a desired function of the interpreter. In contrast, a non-functional requirement describes the characteristics of these functions.

4.2.1 Functional requirements

The main requirement is to evaluate user-defined formulas in SAP HANA and return the results to the user **(FR-1)**. The formula must match the PLCUDF language which uses Excel-style formulas. Furthermore, the execution order of different formulas should be determined based on the relations between those formulas and the dependencies among them **(FR-2)**. Moreover, the existence of any circular reference should be checked, where the formula depends upon the result of the same formula. In the same vein, we should validate the preconditions for each formula **(FR-3)**, to identify the proper formula before starting the evaluation since the formula is not valid for all items.

4.2.2 Non-functional requirements

The execution speed is a measure for the time duration of a query on data sets. The aim is that the execution speed is as fast as possible **(NFR-1)**. To achieve good performance, the formula should be parsed and evaluated in the core of SAP HANA.

ID	Name	Priority
NFR-1	Performance	High
NFR-2	Flexibility	Medium
NFR-3	Validation	High
NFR-4	Extensibility	Medium
NFR-5	Integrity	High
NFR-6	Library agnostic	High
NFR-7	License	High

Table 4.2: Non-functional Requirement of the formula interpreter

Furthermore, the PLC calculation has a hierarchical structure. To calculate any item cost, we sum up all the cost of sub-items. Therefore, the formula language should support a flexible navigation through different hierarchical levels **(NFR-2)**.

The parsing algorithm indicates how quickly the formula can be processed. Furthermore, the algorithm determines the structure of the grammar and may enforce restrictions. For instance, not all algorithms can process a left-recursion **(NFR-1,2)**.

Implementing the solution in the core on SAP HANA imposes a high demand on the validation process as it may affect the SPA HANA database **(NFR-3)**. The approach should be flawless and detect the different errors such as lexical, syntactical and semantics errors. Moreover, it should provide clear messages describing the errors.

A certain level of grammar extensibility is required **(NFR-4)**. One important extension, for example, is adding custom-defined functions.

The integration **(NFR-5)** refers to how easy it is to integrate the approach with PLC, how the call is done and how the returned data is displayed.

Moreover, the library should be agnostic **(NFR-6)** meaning that the dependency on external libraries or tools for the parser generation is minimized. A distinction is made when an external library is needed before runtime and while implementing the solution, or at the runtime to evaluate the formulas.

To integrate the solution with PLC, it is necessary that any external library has a suitable license for commercial uses **(NFR-7)**. The license describes the rights granted to the user using the library. The most common and applicable licenses are MIT, Apache and BSD. In Table 4.2, we list the non-functional requirements with the priority for each.

4.3 Formula language

The formula language in PLC shall resemble the Excel-style formula syntax, since Excel is wide-spread and also used by many PLC customers for their current calculation. However, in PLC the formula is applied to a column and is not defined per cell as in Excel. Therefore, the number of the evaluations may be the same as the row counts.

The prototype of the formula language, Product Life-cycle Costing User-defined Formula (PLCUDF), was discussed in more details in a previous thesis [Kü14]. The grammars in Listing A.2 were expressed in ANTLR using Extended BNF.
The formulas might contain standard or custom fields besides literal values. To distinguish between the item fields and the header fields, two different symbols were used. The dollar symbol ($) refers to item fields, while the hash symbol (#) refers to header fields. The difference between item fields and header fields was explained in Section 1.4.

The formula language supports the arithmetic operators and the most common functions. We can divide the functions into the following categories:

- Mathematical functions such as SQRT, LOG, & ROUND.
- Logical functions such as AND, OR, & IF .
- Aggregate functions such as MAX, COUNT, & SUM.

Those are the basic built-in functions and have similar behavior as in Excel. Additionally, the formula language shall be extended by adding other custom functions.

Using the PLCUDF, the equations in the Section 4.1 are represented as follows:

$$E1 = \$Manufacturing\_hours * 3.5$$

$$E2 = \$Manufacturing\_hours * 3.5 + \$Plant\_distance/2.7$$

$$C = SQRT(\$Energy) * 0.05$$

4.4 Using AFL for evaluation

The formula can be evaluated in SAP HANA using different implementation options. The thesis [Rus15] discusses parsing the formula and generating a special SAP HANA database artefact called Calculation View to make the evaluation and return the result. Our approach is to implement an interpreter for custom formulas using AFL as our target platform for evaluation. AFL was developed specifically so that HANA can execute complex algorithms by maximizing database processing rather than bringing all the data to the application server. The execution is done inside the core of HANA database which brings crucial performance improvements. We list the main benefits for implementing an AFL:

- By pushing the logic down to the data level, the performance is improved. We can take advantage of the powerful CPU features to parallelize the execution.
- Additionally, there are no limitations on writing the language grammars to provide an extensible features such as adding additional data types or defining custom functions.

4.5 Selection of the development approach

There are numerous ways to parse mathematical expression and evaluate it. We will discuss the different solutions, list the advantages and disadvantages of each, in order to build our approach.

1. **ANTLR** is a parser generator that uses LL(*) parsing algorithm. It takes an Extended Backus–Naur Form (BNF) grammar as an input to generate the AST for a given input stream like expressions.
   ANTLR generates the lexer, parser and the syntax tree. It translates the grammar into executable code.
   The tool is very useful in such cases, however the current version of ANTLR 4.5 does not support C++ yet. The previous versions support parsing the expression but do not support generating the AST.

2. **Flex/Bison** are commonly known and mature tools that are used to write parsers for any arbitrary file format from BNF grammar. The parsing algorithm used by Bison is LALR(1).
   These tools are distributed under GPL license. Flex converts the input stream into sequences of tokens, while Bison replaces the sequence that conforms to the syntax with the corresponding grammar in order to build the AST.

3. Another way of implementation is to utilize **an open source math expression parser and evaluator** written in C++, like the following expression parsers:

   - **Speqmath expression parser:** is an expression parser written in C++. It supports using the operators and the basic functions with the possibility to define own variables.
     It is distributed under Apache 2.0 License.
   - **Partow:** is mathematical expression library that supports using operators and many built-in functions with custom variable support. It is distributed under the Common Public License which has restriction against commercial distribution.
   - **MuParser:** is an extensible math expression parser library written in C++. It supports user-defined operators and functions in addition to the predefined features.
     It includes built-in error handling and allows distribution under MIT license.
   - **MTParser:** is an extensible math expression parser written in C++ with many predefined operators and functions and supports user-defined functions and operators.
     It is distributed under The Code Project Open License (CPOL), free to use for non-commercial purpose.

> There are many other similar solutions that share common features. Nevertheless, the main disadvantage of these solutions is the effort to modify the source code to meet our language specifications and adapt the calculation process to work column-wise.

Based on the foregoing, the first solution lacks the required support for our target platform. And using an existing solution requires many modifications. So, our implementation will be using Flex/Bison tools. Writing our own grammar gives the solution flexibility and extensibility to achieve high performance.

4.6 Contribution

Our presented contribution in this thesis consists of various phases, each phase takes the input from the previous stage. We explain the processing steps for each phase, the inputs and the outputs. In Figure 4.1[1], we present a workflow of these different steps to build the interpreter. After building the interpreter, the solution integrates with SAP HANA using AFL as our evaluation target.

4.6.1 Lexical analysis

After the initialization, we start with the actual execution for each formula. The lexical analyzer takes the formula as an input stream and converts it into a list of tokens. The tokenizer checks for valid tokens and passes them on demand to the syntax analyzer [Pys88].
The PLCUDF language includes many tokens such as: identifiers, literals, operators and functions. In Table 4.3, we show the different token types that can be distinguished.

Token type	Token value
Literals	string, real, float, boolean, time, date
Arithmetic symbols	+,-,*,/,%
Comparison symbols	==, !=, <, <=, >, >=
Keywords	functions such as: ceiling, count, lookup and so forth.
Special symbols	), (, !, ˆ

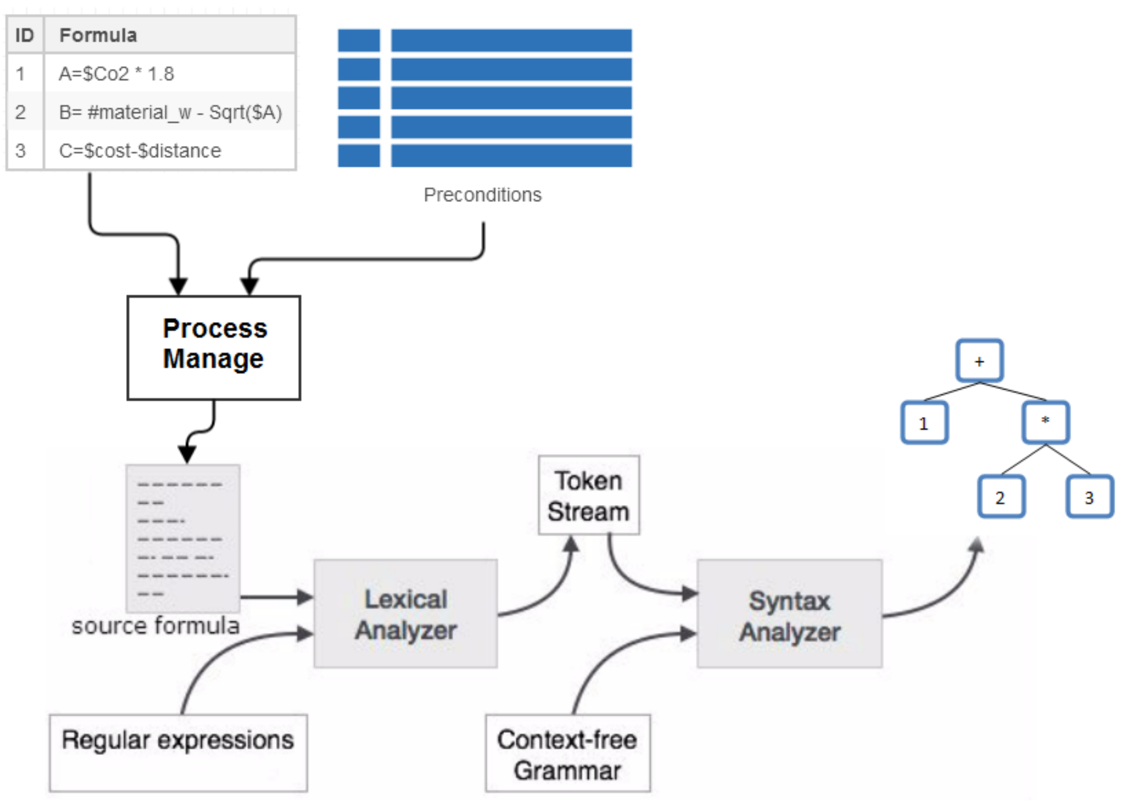
Table 4.3: PLCUDF token types

Token specifications

There are two steps in order to characterize the tokens. First, present the token patterns as regular expressions. Second, define those tokens with their values.

In Listing 4.1, we illustrate a sample of the lexical grammar. This sample consists of two parts. The first part is the regular expression to define an integer value while the

[1]This figure is based on http://www.tutorialspoint.com/compiler_design/compiler_design_syntax_analysis.htm



ID	Formula
1	A=$Co2 * 1.8
2	B= #material_w - Sqrt($A)
3	C=$cost-$distance

Figure 4.1: The workflow scheme

Token type	**Token name**	**Token value**
literals	real_literal	3.5 2.7 0.05
Arithmetic operators	MATH	+ * /
Functions	FUNC	SQRT
Identifiers	ID	$Manufacturing_hours
		$Plant_distance
		$Energy

Table 4.4: Sample of tokens

second part (at line 8) stores the token value and passes both the token name and its value to the parser.

Listing 4.1: Lexical definition for integer literal

```
1  sign                 [+\-]
2  dec_digit            [0-9]
3  dec_literal          {dec_digit}+
4  integer_literal      {sign}?{dec_literal}
5
6  %%
7
8  {integer_literal}   {
9                           yylval.int_value = atoi(yytext);
10                          return INT_LITERAL;
11                     }
```

The full lexical grammar is described extensively in the appendix.

The lexical analyzer detects the token, then builds the lexeme as key-value pair `<token name, token value>` and passes it to the parser.

If we apply this on the energy consumption and carbon dioxide emissions formulas described previously, the tokens in Table 4.4 can be extracted.

4.6.2 Syntax analysis

The syntax analysis is the parsing phase. It takes a list of tokens generated by the lexical analyzer and checks the syntax. The syntax is based on BNF grammar to generate the abstract syntax tree(AST).
At this phase, we are generating a grammar that conforms to the PLCUDF language. BNF is a popular notation for context-free languages [AU72, Gru12]. We will describe the parsing algorithms and show our grammar that is used to validate the formula syntax.

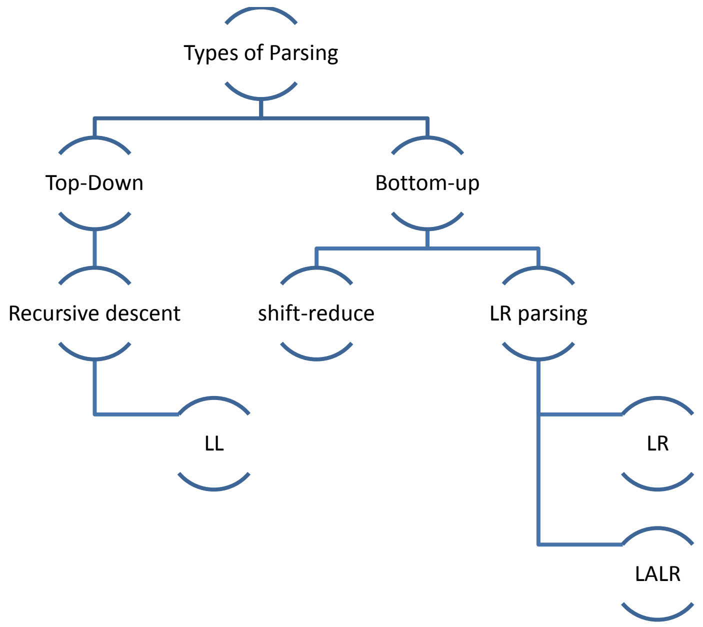


Figure 4.2: Different types of parsing algorithm

4.6.2.1 Types of parsing

The parsing is the first step at the syntactic analysis that analyses a string of symbols conforming to the rules of a formal grammar. The parsing algorithms are divided into two types: top-down parsing and bottom-up parsing [AU72].

Bottom-up

Bottom-up parsers [2] start with the input and try to construct the parse tree up to the start symbol to reach the root node. They start from the basic rule and try to expand it [AU72].
This type of parser performs two actions: *shift* and *reduce*. At the *shift* action, the parser is pushing the next input symbol onto the top of the stack. The shifted symbol is treated as a single node of the parse tree. In contrast, with the *reduce* action, the parser pops the top of the stack when a complete grammar rule is found and replaces it with the right-hand side of the grammar.

[2]Also known as shift-reduce parsers

Operator	Description	Associative
!	Logical negation	non-associative
ˆ	Power operator	right-associative
*, /, %	Multiplicative operators	left-associative
+, -	Additive operators	left-associative
==, !=, <>, <, >, <=, >=	Relational operators	left-associative

Table 4.5: Precedence order

Here we list some of the parser types that use bottom-up parsing include:
LR parser is a non-recursive, shift-reduce parser. It is a left-to-right, rightmost derivation parser. An LR parser is essentially a shift-reduce parsing algorithm driven by parsing a stack.
LALR is look-ahead LR parser. A LALR parser is less powerful than the LR parser, However, this parser generators cannot handle all LR grammars easily.

Top-down Top-down parsers are constructing the parse tree from the root node and gradually moving down to the leaf nodes. They try to find left-most derivations of an input stream. Top-down parsers cannot recognize all context-free languages, there is some grammars producing conflicts. Those conflicts should be solved in order to make the parser works correctly [ALSU13].
Some of the parsers that use top-down parsing include:
Recursive descent parser uses recursive procedures to process the input. It is using backtracking to detect the valid grammar for the input. The backtrack calling is not certain, so it is expensive and time consuming. **LL parser** is a left-to-right, leftmost derivation parser that does not suffer from backtracking. By using look-ahead pointers, the parser knows the right grammar and avoids conflicts.

We summarize in Figure 4.2, the distribution of the most common parsers.

Bison is a parser generator that converts an annotated context-free languages into an LALR parser for that grammar. The parsing algorithm uses one token look-ahead which can be expressed as LALR(1). The generated parser is implemented as a C++ program with a parsing function that can be called from application programs.

4.6.2.2 Grammar

A Bison grammar's file has many sections, the declaration section contains declarations for the terminal and non-terminal symbols, and specify precedence to solve the conflicts between the grammars and set the priorities. In Table 4.5, we show the precedence order for the different operators used in the grammar.

We use BNF notation to write the Bison grammar. The initial point to start parsing from the rule *grammar*. Our grammar has Excel-style, implying each statement starts with the symbol (=) followed by the actual formula. The formula is allowed for an arbitrary number of operators between different types of operands. It may contain literal values, function results or references to database fields. The grammar specification section is shown in Listing 4.2.

Listing 4.2: Context-free languages

```
formula:
  | formula '=' exp END

exp:
    exp CMP exp
   | exp '+' exp
   | exp '-' exp
   | exp '*' exp
   | exp '/' exp
   | exp '^' exp
   | '!' exp
   | '(' exp ')'
   | '-' exp %prec UMINUS
   | literal
   | FUNCTION '(' param_list ')'
   | at_identifier

literal:
     INT_LITERAL
   | REAL_LITERAL
   | STRING_LITERAL
   | BOOL_LITERAL
   | DATE_LITERAL
   | TIME_LITERAL

param_list:
   exp
 | exp ',' param_list
```

Each grammar rule is accompanied with an action. The action is executed each time an instance of that rule is recognized. Through these rules, the AST is built and the semantic checks are done. For simplicity we do not present the actions in Listing 4.2.

After applying these grammars on the formula for the energy consumption (E2), we get the parse tree represented in Figure 4.3.

4.6.2.3 Abstract Syntax Tree

The abstract syntax tree (AST) is the representation of the syntax structure of the source formula that is built recursively during parsing the grammar. It is used in pro-

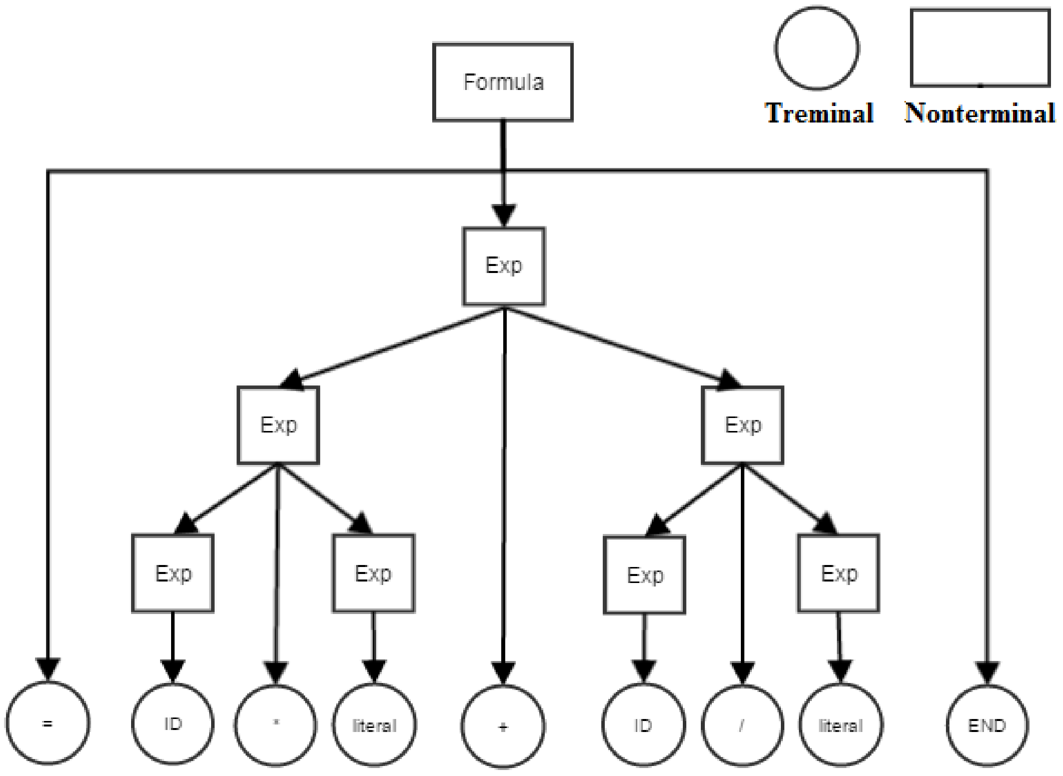


Figure 4.3: Parse tree example

gram analysis and also for the semantic checking. ASTs are generally created bottom-up [Jon03].

The AST has different node types. Those nodes represent grammar rules. In Figure 4.4, we show the class diagram for the *AST* class and the *Nodes*.
The *AST* class has a pointer for the root node of the parse tree. The tree is built while parsing the grammar for an input formula by adding the suitable node to the tree. The *AST* class has also access to the symbol table which in turn contains copies of all the database columns that are used as references in the formulas. The symbol table is filled at the initialization phase.

The *AST* is a binary tree. All the nodes forming the tree are derived from the base class *Node*. Each node has two child nodes: the left node and the right node.
The sub-node classes have an enumeration data member, which refers to its function or the actual holding type. For example, the *MathNode* can be a subtraction node, adding node, multiplication node, division node, or power node.
The formula evaluation is done by invoking the method *evaluate*. This method is defined as a virtual method at the class *Node* and overloaded at the sub-classes. The *evaluate* method does a post order tree traversal. The method is called recursively on both left node and right node.
The evaluation results are combined through the tree traversal up to the root node. During the evaluation the semantic check is done too, for instance, checking the number of parameters passed to a built-in function or for type checking. The type casting among different literal values is another task during the evaluation.

We get the syntax tree represented in Figure 4.5 for the energy consumption formula (E2). This tree is constructed from the previous parse tree and has three different node types. The first type is the **math node** for adding, multiplying and dividing. The second type is the **literal node**, that holds the numeric values. And the last type is the **access node** that refers to other fields.

4.6.3 Error handling

The interpreter should be able to handle different types of errors and report them. The errors may be encountered at various stages [Lou97].

The lexical errors occur at the first stage when the lexer fails to recognize a token, That can be in the case of either a misspelled token or an undefined one. The syntactic errors occur during the parsing when a list of tokens does not conform to any rule and the parser fails to reduce a grammar. Typical errors might be unbalanced parentheses, or a missing operator. The semantic errors occur during the building of the AST and evaluating the formula. Typical errors might be a reference to an undefined variable, an incorrect number of function arguments, an incompatible type, or dividing by zero.

The error-recovery represents the action that should be done when an error arises. The error-recovery strategies that can be implemented are to ignore the error if it does not

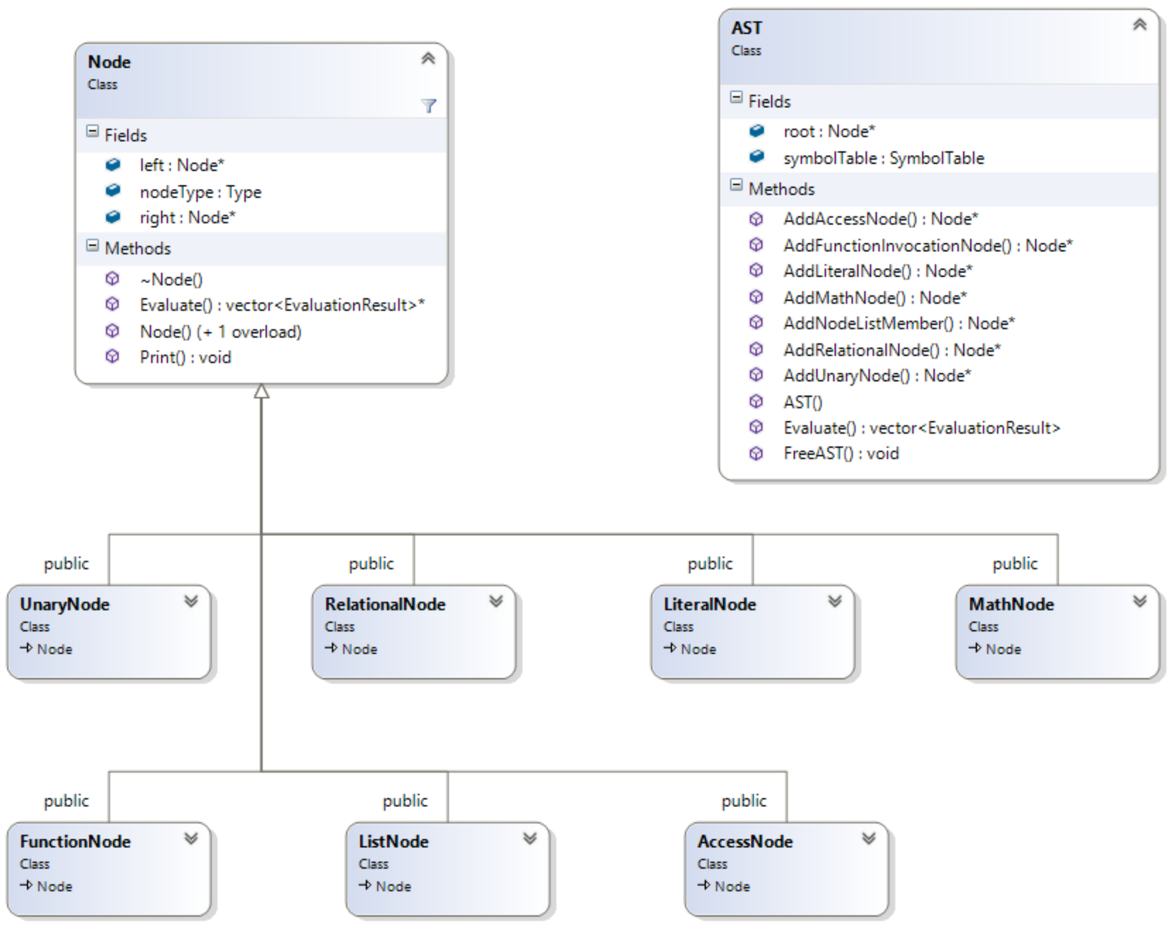


Figure 4.4: Class diagram of the abstract syntax tree nodes

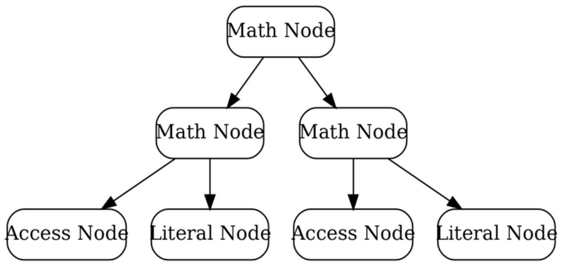


Figure 4.5: Abstract syntax tree example

affect the formula. If the error cannot be ignored, the formula will not be processed and the interpreter reports the error to the user.

4.6.4 Optimization and features

The parsing process consumes more time than the formula evaluation. However, in our case the evaluation is more time consuming while formulas are specified on a column not on individual item level. So, we focus on optimizing the formula evaluation process.

The optimization goal is to make the evaluation as fast as we can and reduce the access to the database as much as possible. The following optimization techniques are feasible with our approach:

- Parallelize the execution
  By taking advantage of the architecture of the SAP HANA server CPU, multiple computations may happen in parallel.There are two possible types of parallelization: The AST parallelization by dividing the execution of branches on many threads. And the other type is column-wise parallelization by dividing the column into ranges and distribute them on threads.
- Expression optimization
  Replace the parts of the expression that represent constant values with a literal node expressing the result.
- Avoid redundancy
  Reuse branches that are already computed and cache them for later use instead of re-computing them every time.
- Dead part elimination
  Remove the instructions that will not execute like multiplying by zero.

4.6.5 Execution

HANA has many possibilities to evaluate the generated AST of the custom-defined formulas. Each of these possibilities uses different technologies and various potential generation targets. There are mainly three approaches distinguished in SAP HANA. The first approach is to generate SQL queries to do the calculation, this generation target was used by [Kü14]. The second approach is to use generated views mapped to the formula to be calculated, this approach was used by [Rus15]. The third approach is to write a function using a programming language supported by SAP HANA. As described in Section 4.4, We choose AFL as our generation target. AFL allows creating extension libraries for the SAP HANA using the C++ language.

To implement AFL extension library, we define the member functions and specify the input and output table assigned to those functions. The definition must conform to AFL standards to be valid. We refer to defined member function as the Process Manager.

The data type definition of the input table is *SharedTableViewer*, while the data type definition of the output table is *SharedTable*, which are special AFL classes to copy the inputs from the database and the copy back the results to data tables. In Listing 4.3, we show the *EVALUATE_EXPRESSION* function signature, containing both input and output tables.

Listing 4.3: AFL function signature

```
AFLMETHODIMP CParserAFL::EVALUATE_EXPRESSION(
      const hana::SharedTableViewer expression,
        const hana::SharedTableViewer preconditionTable,
        const hana::SharedTableViewer itemTable,
        const hana::SharedTableViewer customItemTable,
        const hana::SharedTableViewer metaTable,
        hana::SharedTableViewer result,
        hana::SharedTableViewer errorList) const;
```

The execution follows these steps:

- The defined formulas are extracted and passed to the interpreter to generate the AST. In the same context, the symbol table is filled based on the content of the formulas. Furthermore, we define the sequential order of the formula execution. The order is defined according to a topological sort of the dependencies between the formulas.
- Then, we optimize the generated ASTs and evaluate them. Since all the required fields for the evaluation are available, we do not evaluate the formula row by row, but rather we calculate the results for all the rows at once regardless of the preconditions. This way of evaluation can benefit of the CPU caching among the recursive calling of the different tree nodes and is executed as SIMD (Single instruction, multiple data) as we have the same formula evaluated on all the rows.
- The final step is to check the conditions while writing the results back in the output tables.

5. Evaluation

5.1 Overview

In this chapter, we show different test cases to illustrate the the interpreter features and provide a discussion about evaluation of the implementations.

5.2 Test cases validation

This section is providing many test cases for selected parts of the implementation. In this regard, each test case specifies the execution conditions, the input data and the expected results. The aim of this section is to verify the provided implementation and its underlying concepts but also to carve out their current limitations. Thereby, some of the following test cases also map to particular items of the requirement specification of Section 4.2.

5.2.1 Implicit casting

The aim of this test case is to check the conversions between the compatible types. The type casting provides more flexibility to the formula syntax. During building the AST phase, we support an implicit type casting. Therefore, this is part of the semantic checks. We do the casting whenever it is possible and in case of error a semantic error is thrown.

Input I
The following formula, concatenating two values, is passed to the interpreter. The *code_center* has a string type.

$$Company\_code = Concat(\$code\_center, 100)$$

Expected Results
The *function tree node* has two sub-nodes. It should concatenate these sub-nodes and return a literal results of the type string.

Actual Results
The *function tree node* calls the evaluate method for left sub-node and for the right sub-node. To do the concatenation, it finds out that the sub-nodes have different data types. However, the type casting can be applied. So the numeric data type is converted to a string data type and node returns the concatenation between two strings. The actual result and the expected result do match.

Input II
The following formula is passed to the interpreter. It returns the square root of the *center_name* field. This field has a string type.

$$Company\_code = Sqrt(\$center\_name)$$

Expected Results
The formula should return the square root results of a numeric type or an error in case the interpreter could not apply the type casting.

Actual Results
The interpreter finds out that the mathematical function requires a numeric type argument. However, the *center_name* field does not have a compatible type. The interpreter tries to convert the string into a numeric value. In this case, the conversion will fail and an error will be thrown. So, the actual result and the expected result do match.

5.2.2 Constant branches

The constant branch is a part of the AST generated from a formula or part of a formula. It is equivalent to a literal value so there is no need to compute the value each time we evaluate the formula. During the phase of building the AST, the constant branch is marked and replaced by a literal value.

Input
The following formula is passed to the interpreter, it calculates the distance based on the given equation.

$$Distance = \$plant\_loc + (4 * 2)$$

Expected Results
The interpreter finds out that the formula has a constant branch. The AST is updated and the branch is replaced by a literal node.

Actual Results
The right branch of the AST is marked as constant. The AST is updated and a new literal node is created instead of the branch. In Figure 5.1, we show how the AST is updated. Therefore, the result of this test case meets the stated expectation.

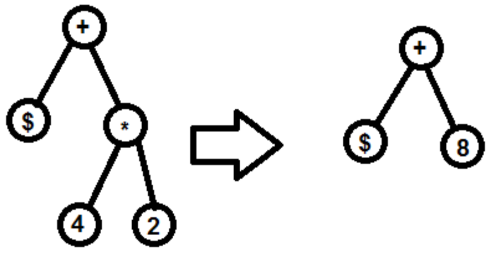


Figure 5.1: AST update, constant branch

5.2.3 Arithmetic exceptions

The interpreter faces unknown input that can cause arithmetic exceptions such as dividing by zero or taking the square root of a negative number. Those cases are affecting the execution and may have impact on the database system.

Input
The following formula is passed to the interpreter. It calculates the profit bases on the *fixed_cost* field.

$$Profit = 1.75/\$fixed\_cost$$

Expected Results
Some of the materials do not have fixed cost, so the fields value is zero. The interpreter should detect division by zero and report an error.

Actual Results
The interpreter catches the exception and adds a division by zero error to the output including the formula identifier and the material number. The interpreter does not stop the execution for the other materials and the other formulas. The actual result and the expected result match.

5.3 Performance evaluation

In the previous section, we presented different test cases showing the features of our approach. In this section we discuss the performance and evaluate the parsing time and the execution time using different formula. First, we present the evaluation setup, then the parsing and execution evaluation.

5.3.1 Setup

The testing supposed to work in HANA server to get the real performance numbers, however it was not easy at the remaining time to test it in HANA due to additional

Formula	Row counts		
	100 rows	1000 rows	100000 rows
Formula1	7	30	1600
Formula2	20	110	7600

Table 5.1: Execution time

required steps from AFL SDK. Instead, we test the evaluation at normal PC. this can give good indicator about the performance .

We used two type of formulas to check the impact on the parsing time. The first one is a simple while the second is complex formula using complicated functions and many references to item fields.

$$Formula1 = \$price * 1.05$$

$$Formula2 = if(\$plantID = 1000, max(\$fixed\_price, \$variable\_price),$$
$$sum(\$fixed\_price, \#gross\_cost, \#work\_price))$$

The data set size is 100 rows, 1000 rows, and then 100000.

5.3.2 Parsing evaluation

As mention in the previous chapter, at the parsing phase the interpreter reads the formula and generate the AST. the parsing time is not so critical for our solution. Each formula is parsed once while evaluated many times. The first formula takes around 40 microseconds, while the second formula takes around 150 microseconds.

5.3.3 Execution evaluation

The execution time is significant. unlike the parsing that occur once, the execution will be occur for all the columns for each formula. It differs according to the AST size and the column numbers. In Table 5.1, We show the execution time in milliseconds for the previous formulas using sample data sets.

6. Future Work

The contribution of our thesis is aimed at adding flexibility to PLC on custom-fields and custom formula domains. We focus on giving a prototypical implementations. Both domains show promising results, in that our approach misses additional optimization studies.

Moreover, the current SDK version of AFL does not support arbitrary SQL command execution. This has an impact on some functions such as the Lookup function, which requires the Lookup table. The only case that it can work in our approach, when the table is included as an input.

Several security topics remain open for the future work, especially at the custom formula evaluation. AFL execution presents a critical security issue.

During the limit of time for the thesis, we put the design for some features which was not implemented. We suggest to extend the implementation of the hierarchy navigation through the levels of the items and providing different aggregation capabilities.

A. Appendix

This appendix shows the lexical grammar that is used in Flex.

Listing A.1: lexiacal grammar

```
white_space          [ \r]+
identifier          [A-Za-z_][A-Za-z_0-9]*

at_identifier       [#|$]{identifier}

eol                 [\n\0]



dec_digit             [0-9]
dec_literal           {dec_digit}+
integer_literal       {dec_literal}

identifier_error    {dec_digit}+{identifier}


sign                  [+\-]
exponent_part         [eE]{sign}?{dec_digit}+
whole_real1           {dec_digit}+{exponent_part}
whole_real2           {dec_digit}+
part_real             {dec_digit}*\.{dec_digit}+{exponent_part}?
real_literal          {whole_real1}|{whole_real2}|{part_real}


simple_esc_seq        \\[\'\"\\0abfnrtv]
error_esc_seq       \\[^\'\"\\0abfnrtv]
```

```
single_string_char   [^\\\"\n]
reg_string_char      {single_string_char}|{simple_esc_seq}
regular_string       \"{reg_string_char}*\"
err_string_not_closed \"{reg_string_char}*
error_string       \"({reg_string_char}*{error_esc_seq})+{reg_string_char}*\"
string_literal       {regular_string}


date          (0[1-9]|[12][0-9]|3[01])[- /.](0[1-9]|1[012])[-
   /.](19|20)[0-9][0-9]

time24format    ([01]?[0-9]|2[0-3]):[0-5][0-9]
time12format    (1[012]|[1-9]):[0-5][0-9](am|pm)

time          {time24format}|{time12format}

%%

                    /***** White Space *****/
{white_space}       { colCount += yyleng; /* ignore */ }

                    /***** identifier *****/
{at_identifier}      { colCount += yyleng; yylval.sv = _strdup(yytext);
   return at_identifier;}



                    /***** Literals *****/
{time}             { colCount += yyleng; yylval.sv = yytext; return TIME;}
{date}             { colCount += yyleng; yylval.sv = yytext; return DATE;}
{integer_literal}   { colCount += yyleng; yylval.fv = atoi(yytext); return
   real_literal; }
{real_literal}      { colCount += yyleng; yylval.fv = atof(yytext); return
   real_literal; }
{string_literal}  { colCount += yyleng; yylval.sv = _strdup(yytext); return
   string_literal; }
"false"            { colCount += yyleng; yylval.bv = false; return BOOL; }
"true"             { colCount += yyleng; yylval.bv = true; return BOOL; }

{error_string}     { ast->AddLexError(yytext,rowCount,colCount,"String
   Error");
                        colCount += yyleng; yylval.sv = _strdup(yytext);
                        return string_literal;}
{err_string_not_closed} { ast->AddLexError(yytext,rowCount,colCount,"String
   Error");
                        colCount += yyleng;yylval.sv = _strdup(yytext);
                        return string_literal; }


\t            { colCount = (ceil((colCount+1)/8.0f))*8 ; /* ignore */}
{eol}            { colCount=0; return EOL;}

           /***** Multi Chars Oprs *****/

">"    {colCount += yyleng; yylval.iv = 1; return CMP; }
"<"    {colCount += yyleng; yylval.iv = 2; return CMP; }
"<>"   {colCount += yyleng; yylval.iv = 3; return CMP; }
"!="   {colCount += yyleng; yylval.iv = 3; return CMP; }
"=="   {colCount += yyleng; yylval.iv = 4; return CMP; }
">="   {colCount += yyleng; yylval.iv = 5; return CMP; }
"<="   {colCount += yyleng; yylval.iv = 6; return CMP; }

        /**** Single Allowed Chars ****/
"("  { colCount += yyleng; return yytext[0]; }
")"  { colCount += yyleng; return yytext[0]; }
"+"  { colCount += yyleng; return yytext[0]; }
"-"  { colCount += yyleng; return yytext[0]; }
"*"  { colCount += yyleng; return yytext[0]; }
"/"  { colCount += yyleng; return yytext[0]; }
"%"  { colCount += yyleng; return yytext[0]; }
"!"  { colCount += yyleng; return yytext[0]; }
"^"  { colCount += yyleng; return yytext[0]; }
"="  { colCount += yyleng; return yytext[0]; }



                  /***** Keywords *****/
"sqrt"  { colCount += yyleng; yylval.iv = B_sqrt;  return FUNC; }
"round"    { colCount += yyleng; yylval.iv = B_round;  return FUNC; }
"roundup"  { colCount += yyleng; yylval.iv = B_roundup; return FUNC; }
"rounddown" { colCount += yyleng; yylval.iv = B_rounddown; return FUNC; }
"ceiling"  { colCount += yyleng; yylval.iv = B_ceiling; return FUNC; }
"floor"    { colCount += yyleng; yylval.iv = B_floor;  return FUNC; }
"sign"  { colCount += yyleng; yylval.iv = B_sign;  return FUNC; }
"abs"      { colCount += yyleng; yylval.iv = B_abs;  return FUNC; }
"exp"      { colCount += yyleng; yylval.iv = B_exp;  return FUNC; }
"string"   { colCount += yyleng; yylval.iv = B_string;  return FUNC; }


"ln"       { colCount += yyleng; yylval.iv = B_ln;    return FUNC; }
"log"      { colCount += yyleng; yylval.iv = B_log;  return FUNC; }
"mod"      { colCount += yyleng; yylval.iv = B_mod;  return FUNC; }
"concat"   { colCount += yyleng; yylval.iv = B_concat;  return FUNC; }
```

```
"count"    { colCount += yyleng; yylval.iv = B_count;  return FUNC; }
"min"      { colCount += yyleng; yylval.iv = B_min;  return FUNC; }
"max"      { colCount += yyleng; yylval.iv = B_max;  return FUNC; }

"if"    { colCount += yyleng; yylval.iv = B_if;    return FUNC; }
"and"      { colCount += yyleng; yylval.iv = B_and;  return FUNC; }
"or"       { colCount += yyleng; yylval.iv = B_or;    return FUNC; }
"not"      { colCount += yyleng; yylval.iv = B_not;  return FUNC; }




                /***** COMMA *****/
","    { colCount += yyleng; return yytext[0];}

                    /***** Error handle *****/
{identifier_error} {ast->AddLexError(yytext,rowCount,colCount,"invalid
    identifier name"); colCount += yyleng;
                //yylval.sv = TrimStart(yytext); return at_identifier;
                }

{identifier} { ast->AddLexError(yytext,rowCount,colCount,"invalid keyword");
    colCount += yyleng;}

.         {  ast->AddLexError(yytext,rowCount,colCount,"invalid character");
    colCount += yyleng;}
%%
```

The Linkedin criteria list [Lin]

This appendix shows the criteria used by Linkedin to choose the tempalte engine. The following list of features was given to different teams to look for in the assigned templating solution, and fill out a score, from one (poor) to five (excellent), for each item:

1. DRY: how DRY is the templating technology? is there support for code-reuse and partials?
2. i18n: is there support for translations and multiple languages?
3. Hot reload: are changes visible immediately or is there a compile/deploy cycle?
4. Performance: how long does it take to render in the browser and server?
5. Ramp-up time: how is the learning curve?

6. Ramped-up productivity: once you've ramped-up, how fast can you build things?
7. Server/client support: can the same template be rendered both client-side and server-side?
8. Community: is there an active community using this project? Can you google issues?
9. Library agnostic: are there dependencies on other JS libraries, such as jQuery or Mootools?
10. Testable: how hard is it to write unit and integration tests?
11. Debuggable: is it possible to step through the code while it's running to track down errors?
12. Editor support: is there an editor with auto-complete, syntax highlighting, error checking, etc?
13. Maturity: is this a relatively stable project or still experimenting and churning?
14. Documentation: how is the documentation?
15. Code documentation: do the templates encourage/require documentation/comments?

This appendix shows the PLCUDF ANTLR grammar used by [Kü14].

Listing A.2: PLCUDF grammar for ANTLRv4 [Kü14]

```
//Grammar definition for ANTLRv4, PLC User-defined-Formulas
//Version 30.06.2014

grammar PLCUDF;

//Parser Rules
start : '=' expr ; // Excel-Style
expr : ' ( ' expr ') ' #ExprParentheses
| ( '+'|'-') expr #ExprPrefix
| <assoc=right> expr '^' expr #ExprMyPower
| expr ('*' | '/') expr #ExprMulDiv
| expr '%' #ExprPercent
| expr ('+'|'-') expr #ExprPlusMinus
| expr ('='|'<>'|'<'|'<='|'>'|'>=') expr #ExprLogic
| fkt #ExprFunction
| HEADER #ExprHeader
| COLUMN #ExprID
| NUMBER #ExprNum
| BOOL #ExprBool
```

```
;
fkt : IDENTIFIER '(' list? ')'; //for all functions
list : elem (' ; ' elem)* ;
elem : expr #ElemSingle
| COLUMN ':' COLUMN #ElemRange
| TABLE #ElemTable
;

//Lexer Rules
BOOL : TRUE | FALSE ;
TRUE : [Tt] [Rr] [Uu] [Ee]; //case-sensitive-free
FALSE : [Ff] [Aa] [Ll] [Ss] [Ee];
HEADER : '#' IDENTIFIER ;
TABLE : '!' IDENTIFIER ;
COLUMN : '$' IDENTIFIER ;
IDENTIFIER: LETTERS(DIGITS|LETTERS|SPECIALS)*;
NUMBER : DIGITS+   //e.g. 1234
| DIGITS+ (','|'.' ) DIGITS*  //e.g. 1, or 1,234 or 1. or 1.234
| ' , ' DIGITS+  //e.g. ,1
| ' . ' DIGITS+  //e.g. .1
| ((DIGITS)? DIGITS)? DIGITS ('.' DIGITS DIGITS DIGITS)+ ( ',' DIGITS* )*
   //e.g. 1.234.567,345345 or 1.234.1233, or 1.234.123
| ((DIGITS)? DIGITS)? DIGITS (',' DIGITS DIGITS DIGITS)+ ( '.' DIGITS* )*
   //e.g. 1,345,345.245034 or 1,234,1233. or 1,123,123
;
fragment LETTERS : [a-zA-Z] ;
fragment DIGITS : [0-9] ;
fragment SPECIALS: '#'|'_' ;
WS : [ \t\r\n]+ -> skip ;
```

Bibliography

[AACF12] David Ameller, Claudia Ayala, Jordi Cabot, and Xavier Franch. How do software architects consider non-functional requirements: An exploratory study. In *2012 20th IEEE International Requirements Engineering Conference, RE 2012 - Proceedings*, pages 41–50, 2012. (cited on Page 19)

[ABvdB07] Jeroen Arnoldus, Jeanot Bijpost, and Mark van den Brand. Repleo. In *Proceedings of the 6th international conference on Generative programming and component engineering - GPCE '07*, page 25, New York, New York, USA, October 2007. ACM Press. (cited on Page 9)

[ALSU13] A.V. Aho, M.S. Lam, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools*. Always learning. Pearson, 2013. (cited on Page 40)

[Ash15] Jeremy Ashkenas. *Underscore.js - Template Engine*, 2013 (Accessed June 15, 2015). http://documentcloud.github.io/underscore/. (cited on Page 23)

[AU72] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1972. (cited on Page 38 and 39)

[Ber07] Antonia Bertolino. Software Testing Research: Achievements, Challenges, Dreams. In *Future of Software Engineering (FOSE '07)*, pages 85–103. IEEE, May 2007. (cited on Page 10)

[BIJ06] Alan W Brown, Sridhar Iyengar, and Simon Johnston. A rational approach to model-driven development. *IBM Systems Journal*, 45(3):463–480, 2006. (cited on Page 8)

[CH03] Krzysztof Czarnecki and Simon Helsen. Classification of model transformation approaches. In *Proceedings of the 2nd OOPSLA Workshop on Generative Techniques in the Context of the Model Driven Architecture*, volume 45, pages 1–17. USA, 2003. (cited on Page 8)

[CH06] K Czarnecki and S Helsen. Feature-based Survey of Model Transformation Approaches. *IBM Syst. J.*, 45(3):621–645, 2006. (cited on Page 8)

[Con15] Jupiter Consulting. *EJS - Template Engine*, 2010 (Accessed June 15, 2015). http://www.embeddedjs.com/. (cited on Page 23)

[Cor15] LinkedIn Corporation. *Dust.js - Template Engine*, 2015 (Accessed June 15, 2015). http://www.dustjs.com/. (cited on Page 23)

[Dok15] Laura Doktorova. *doT.js - Template Engine*, 2014 (Accessed June 15, 2015). http://olado.github.io/doT/. (cited on Page 23)

[FM12] F Färber and Norman May. The SAP HANA Database–An Architecture Overview. *IEEE Data ...*, pages 1–6, 2012. (cited on Page 14)

[Gru12] van Reeuwijk K.-Bal H.E. Jacobs C.J.H. Langendoen K Grune, D. *Modern compiler design. Springer Science & Business Media*. Springer-Verlag New York, 2012. (cited on Page 38)

[Inc15] Twitter Inc. *Hogan.js - Template Engine*, 2011 (Accessed June 15, 2015). http://twitter.github.io/hogan.js/. (cited on Page 23)

[Jon03] Joel Jones. Abstract syntax tree implementation idioms. In *Proceedings of the 10th conference on pattern languages of programs (plop2003)*, 2003. (cited on Page 43)

[Kat15] Yehuda Katz. *Handlebars - Template Engine*, (Accessed June 15, 2015). http://handlebarsjs.com/. (cited on Page 23)

[Kü14] Alrik Künne. Entwicklung und Erprobung einer DSL für kundenspezifische Kalkulationen in SAP HANA. Diploma thesis, HTW Dresden, 2014. (cited on Page xi, 6, 34, 45, and 57)

[Leh15] Jan Lehnardt. *Logic-less mustache templates with JavaScript*, (Accessed June 15, 2015). https://github.com/janl/mustache.js/. (cited on Page 23)

[Lin] Lindedin, The client-side templating throwdown. (cited on Page 23, 24, and 56)

[Lin15] Forbes Lindesay. *Jade - Template Engine*, 2015 (Accessed June 15, 2015). http://jade-lang.com/. (cited on Page 23)

[Loc10] Henrik Lochmann. *HybridMDSD: Multi-Domain Engineering with Model-Driven Software Development using Ontological Foundations*. PhD thesis, Technischen Universität Dresden, 2010. (cited on Page 8 and 9)

[Lon10] Grzegorz Loniewski. OpenUP/MDRE: A Model-Driven Requirements Engineering Approach for Health-Care Systems. Master's thesis, Universitat Politècnica de València, 2010. (cited on Page 7)

[Lou97] K. Loudon. *Compiler Construction*. Boston, MA, 1997. (cited on Page 43)

[LX14] Xiaowei Li and Yuan Xue. A survey on server-side approaches to securing web applications. *ACM Computing Surveys*, 46(4):1–29, March 2014. (cited on Page 10)

[Mar15] Adam Mark. *Markup.js - Template Engine*, 2013 (Accessed June 15, 2015). https://github.com/adammark/Markup.js/. (cited on Page 23)

[MIT] The MIT License. http://opensource.org/licenses/MIT. Accessed: 2015-05-01. (cited on Page 19)

[MV13] C. Mankala and G.M. V. *SAP HANA Cookbook*. **. Packt Publishing, 2013. (cited on Page 14)

[OMG08] OMG. MOF Model to Text Transformation Language, v1.0, 2008. (cited on Page 10)

[Par04] Terence John Parr. Enforcing strict model-view separation in template engines. In *Proceedings of the 13th conference on World Wide Web - WWW '04*, page 224, New York, New York, USA, May 2004. ACM Press. (cited on Page 10)

[PL15] H. Plattner and B. Leukert. *The In-Memory Revolution: How SAP HANA Enables Business of the Future*. Springer International Publishing, 2015. (cited on Page 14)

[PSI15] Phillip Gates-Idem Patrick Steele-Idem. *Marko - Template Engine*, (Accessed June 15, 2015). https://github.com/raptorjs/marko. (cited on Page 23)

[Pys88] Arthur B. Pyster. *Compiler Design and Construction*. Van Nostrand Reinhold, 1988. (cited on Page 36)

[Rus15] Rico Ruszewski. Analyse, Entwurf und prototypische Implementierung eines Übertragungsund Speicherungsformats für kundenspezifische Formeln in HANA XS. Master's thesis, HTW Dresden, 2015. (cited on Page 6, 34, and 45)

[Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003. (cited on Page 8)

[Sil15] Ronald Silberstein. *Introducing...SAP HANA Extended Application Services (XS)*, 2013 (Accessed June 1, 2015). https://blogs.saphana.com/2013/04/25/introducing-sap-hana-extended-application-services-xs/. (cited on Page vii and 15)

[SK03] Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Softw.*, 20(5):42–45, September 2003. (cited on Page 8) [sql11] *SAP In-Memory Database – SQLScript Guide*, volume 1.0. SAP SE, 2011. (cited on Page 14)

[Sta12] International Standard. INTERNATIONAL STANDARD ISO / IEC / IEEE Systems and software engineering — agile environment. 2012, 2012. (cited on Page 18 and 20)

[SV05] T. Stahl and M. Völter. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt-Verlag, 2005. (cited on Page 8)

[Tec15] Akamai Technologies. *ESI Language Specification 1.0*, 2011 (Accessed June 1, 2015). http://http://www.w3.org/TR/esi-lang. (cited on Page 11)

[TS09] Michiaki Tatsubori and Toyotaro Suzumura. HTML templates that fly. In *Proceedings of the 18th international conference on World wide web - WWW '09*, page 951, New York, New York, USA, April 2009. ACM Press. (cited on Page 10)

[Wal12] M. Walker. *SAP HANA Starter*. Packt Publishing, 2012. (cited on Page 12 and 14)

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den *[. . .]*