BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

IDE FOR SCADA DEVELOPMENT AT CERN

IDE PRO SCADA VÝVOJ V CERN

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR Bc. MATĚJ MAREČEK
AUTOR PRÁCE

SUPERVISOR Ing. ZBYNĚK KŘIVKA, Ph.D.
VEDOUCÍ PRÁCE

BRNO 2016

Master Thesis Specification/18697/2015/xmarec12

Brno University of Technology - Faculty of Information Technology

Department of Information Systems Academic year 2015/2016

Master Thesis Specification

For: **Mareček Matěj, Bc.**
Branch of study: Management and Information Technologies
Title: **IDE for SCADA Development at CERN**
Category: Compiler Construction

Instructions for project work:

1. Study the theory of programming languages, code parsing, and analysis.
2. Get familiar with SCADA systems (SIMATIC WinCC Open Architecture) and widely used IDEs (integrated development environments).
3. Discuss with SCADA developers at CERN (European Organization for Nuclear Research) their needs related to SCADA development and based on the knowledge and experience gained in steps 1 and 2, design a software tool that will suit the needs.
4. Implement the designed solution/software tool.
5. Gather a feedback form CERN SCADA developers and analyze it in order to summarize weaknesses and strengths of the implemented tool.
6. Based on the feedback analysis, evaluate the results and discuss the future development of SCADA systems and tools used for SCADA development.

Basic references:

- Clayberg, E., Rubel, D.: *Eclipse Plug-ins*. Addison-Wesley, Third Edition, 2009, 878 s.
- Vogel, L.: *Eclipse 4 Application Development: The complete guide to Eclipse 4 RCP development* (*Volume* 1). Lars Vogel, 2012, 432 s.
- Bettini, L.: *Implementing Domain-Specific Languages with Xtext and Xtend*. Packt Publishing, 2013, 324 s.
- Nielson, F., Nielson, H.R., Hankin, C.: *Principles of Program Analysis*. Springer-Verlag, 2005, 452 s.

Requirements for the semestral defense:
Items 1 to 3.

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Master Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Křivka Zbyněk, Ing., Ph.D.**, DIFS FIT BUT
Beginning of work: November 1, 2015
Date of delivery: May 25, 2016

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

Dušan Kolář
Associate Professor and Head of Department

Abstract

The goal of this master's thesis is to design and implement an IDE (Integrated Development Environment) that makes development for SIMATIC WinCC Open Architecture more effective and secure. This thesis is based on a research made by Eindhoven University of Technology and it meets needs of CERN EN ICE SCD section.
The developed IDE is built on top of the Eclipse Platform and it uses Xtext for code parsing, scoping, linking and static code analysis. The IDE also supports a new programming language that allows programmers to easily define templates for WinCC OA configuration files. The interpreter of this new language is able to parse a template and a configuration file and decide whether the configuration file matches the template.
The practical result of this thesis is an IDE that supports WinCC OA developers at CERN and performs periodical analysis of CERN code written in Control script Language.

Abstrakt

Cílem této magisterské práce je navrhnout a implementovat IDE (integrované vývojové prostředí), které zvýší efektivitu a bezpečnost vývoje pro SIMATIC WinCC Open Architecture. Tato práce je založena na výzkumu provedeném týmem z Technické univerzity v Eindhovenu a splňuje požadavky pocházející ze SCD sekce v CERN (Evropské organizace pro jaderný výzkum).
Vyvinuté IDE je postaveno na platformě Eclipse, přičemž pro syntaktickou analýzu, linkování a sémantickou analýzu kódu používá Xtext framework. IDE nabízí také podporu pro nově vytvořený programovací jazyk, který umožňuje programátorům jednoduše nadefinovat šablonu pro konfigurační soubory používané WinCC OA. Interpret tohoto nového jazyka je schopen provést syntaktickou analýzu šablony a konfiguračního souboru a rozhodnout, zdali konfigurační soubor odpovídá šabloně.
Praktickým výstupem této práce je integrované vývojové prostředí, které podporuje vývoj WinCC OA aplikací v CERN a periodicky provádí analýzu kódu těchto aplikací napsaného v jazyce Control script.

Keywords

SCADA systems, WinCC OA, PVSS, CERN, programming languages, Java, Xtext, Eclipse, grammars, abstract syntax tree (AST), linking, static code analysis, code interpretation.

Klíčová slova

SCADA systémy, WinCC OA, PVSS, CERN, programovací jazyky, Java, Xtext, Eclipse, gramatiky, abstraktní syntaktický strom (AST), linkování, statická analýza kódu, intepretace kódu.

Reference

MAREČEK, Matěj. *IDE for SCADA Development at CERN.* Brno, 2016. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Křivka Zbyněk.

IDE for SCADA Development at CERN

Declaration

Hereby, I declare that I have created this thesis independently and under the supervision of Zbyněk Křivka (VUT FIT), Axel Voitier and Manuel Gonzalez Berges (CERN).

Another thesis-related information was provided by Łukasz Góralczyk, Paul Burkimsher, Stefan Klikovits, Jean-Charles Tournier, Josef Hofer, Fernando Varela Rodriguez, Piotr Golonka, Rafal Lukasz Kulaga a Giulio Morpurgo.

I have mentioned all literature sources and publications used in this work.

.....
Matěj Mareček
May 17, 2016

Acknowledgements

First of all, I would like to thank my academic supervisor Zbyněk Křivka for his valuable input. By sharing his experience and giving me advice, he helped me a lot with shaping this thesis into the current form.

I would also like to express my deep gratitude to the whole EN-ICE-SCD team at CERN, especially my supervisors; Axel Voitier and Manuel Gonzalez Berges. They gave me the chance to work at CERN, to work with awesome people and gain a really unique experience.

And finally, a special thanks to my friend Łukasz Góralczyk. He was the one, who helped me the most. His guidance, suggestions and willingness to help me with both work and non-work related problems was unbelievable.

© Matěj Mareček, 2016.

This thesis was created as a school work at the Brno University of Technology, Faculty of Information Technology. The thesis is protected by copyright law and its use without author's explicit consent is illegal, except for cases defined by law.

Contents

1 Introduction **4**

2 Thesis Origin **5**
2.1 CERN History . . . 6
2.1.1 EN-ICE Group . . . 7
2.1.2 WinCC OA . . . 8
2.2 Original IDE Requirements . . . 9
2.3 Additional IDE Requirements . . . 11
2.4 Aims of the IDE . . . 13

3 Theoretical Base **15**
3.1 Integrated Development Environments . . . 15
3.1.1 Plug-ins and IDE Platforms . . . 15
3.1.2 Comparison of IDEs . . . 16
3.1.3 Selection of the Platform for WinCC OA IDE . . . 18
3.2 Programming Languages . . . 19
3.2.1 Config Template Language . . . 20
3.2.2 Control script Language . . . 20
3.3 Processing of Programming Languages . . . 22
3.3.1 Grammar Definition . . . 24
3.3.2 Semantic Analysis . . . 32
3.3.3 Code Execution . . . 36

4 Design of the IDE **41**
4.1 Overall Decomposition . . . 41
4.2 The IDE Architecture . . . 42
4.2.1 The Workspace . . . 43
4.2.2 The Workbench . . . 44
4.3 Functional Decomposition of the IDE . . . 46
4.4 The IDE Software Stack . . . 47
4.5 The IDE Plug-ins Design . . . 49
4.5.1 CTRL Plug-in Design . . . 49
4.5.2 Config Plug-in Design . . . 50
4.5.3 Config Template Plug-in Design . . . 51

5 Implementation **54**
5.1 Used Programming Languages . . . 54
5.1.1 Java . . . 54
5.1.2 Xtend . . . 55
5.1.3 C++ . . . 56
5.1.4 JavaScript . . . 57
5.1.5 XML & JSON . . . 57
5.2 Dependency Injection . . . 57
5.2.1 Dependency Injection – Developer's Notes . . . 58
5.3 CTRL Plug-in . . . 59
5.3.1 Parsing . . . 59
5.3.2 Linking and Scoping . . . 61
5.3.3 Linking and Scoping in CTRL Plug-in . . . 63
5.3.4 Performance Optimizations . . . 66
5.3.5 Memory Optimizations . . . 67
5.3.6 CTRL Static Code Analysis . . . 69
5.3.7 Panel Files Support . . . 82
5.4 Config Plug-in . . . 83
5.4.1 Simplified Config Grammar . . . 84
5.4.2 Data-Driven Approach . . . 85
5.5 Config Template Plug-in . . . 87
5.5.1 Config Template Grammar . . . 88
5.5.2 Config Template Engine . . . 90
5.6 Working with Xtext and Xtend . . . 94
5.6.1 Xtext Experience . . . 94
5.6.2 Xtend Experience . . . 95
5.7 Results . . . 95
5.7.1 Personal Observations . . . 95
5.8 Detected Problems . . . 96
5.8.1 Technology Limitations . . . 96
5.8.2 Missing Information . . . 97

6 Conclusion **98**
6.1 Future Work . . . 99

Bibliography **101**

Appendices **104**
List of Appendices . . . 105

A Demonstration Videos **106**

B Supplementary Information about Programming Languages **107**
B.1 Formal languages (Chomsky hierarchy) . . . 107
B.1.1 Regular Languages . . . 108
B.1.2 Context-free Languages . . . 109
B.1.3 Context-sensitive Languages . . . 110
B.1.4 Recursively Enumerable Languages . . . 110
B.2 Classification of Real-World Programming Languages . . . 110

B.2.1 Classification of CTRL Language 112

C Manual **114**

D Notes **115**

Chapter 1

Introduction

We live in the *Information Age*. Every year, mankind gathers, generates and stores more and more data. In order to process so much data as we have nowadays, we have to create virtual systems (programs) of enormous complexity and with a good performance. To describe programs, we usually use a formalized description, often in a form of some programming language.

And the very programming languages are the reason why we use IDEs (Integrated Development Environments). Tools that allow programmers to better understand code base they have to work with, write new code faster and detect potential bugs.

This master's thesis aims to design and implement an IDE that makes development for SIMATIC WinCC Open Architecture more effective and it reduces the amount of bugs in the code. The IDE was developed at CERN (European Organization for Nuclear Research) EN ICE SCD section to meet its needs of SCADA development.

The contents of this thesis clarifies the origin of the developed IDE, requirements of CERN and the connection to Eindhoven University of Technology that started the research.

The following chapter covers the theoretical base. It explains what an IDE is, what are the most commonly used IDEs and what are the main differences among them. More importantly, the theory of programming languages and its application is described later in that chapter. It shows the whole process of a compiler/interpreter development; formal description of programming languages, code parsing, semantic analysis and code execution.

The next, more practical part of the thesis is focused on the design of the IDE and the reasoning behind the design decisions and the technology choices that were taken.

The topic that follows highlights some interesting or somehow problematic parts of the implementation of the IDE. The problems and solutions are mostly related to Eclipse, Xtext (Xtext Grammar Language), performance issues and Java programming.

The results are evaluated and along with future development of the IDE discussed at the end of this thesis.

Chapter 2

Thesis Origin

This project started approximately in the autumn of 2013. EN ICE SCD (Engineering Department / Industrial Controls & Engineering / SCADA[1] Systems) section at CERN was exploring some possibilities how to increase productivity and safety when writing code for the SCADA system they were using (SIMATIC WinCC Open Architecture[2]).

The SCD section cooperated with Eindhoven University of Technology (TUE) in order to analyze CERN needs, suggest a solution and implement a prototype that could prove that the suggested solution can satisfy the needs and is useful for real SCADA development.

I personally got involved in this project at the beginning of 2014 when I was accepted as a *Technical Student* at the European Organization for Nuclear Research (CERN). The *Technical Student Programme* allows students to learn on-the-job, get some more practical experience before graduating their university or complete their final projects. The duration of this program is 4 to 12 months. Additional extension of another 2 months can be given in certain cases, so the total maximum is 14 months.

My work at CERN began in July of 2014 and continued till the end of August of 2015. During this 14 months, I have spent most of the time working on the IDE.

The first two weeks were mostly about learning the architecture of the IDE and related technologies and experimenting with them (WinCC OA, Eclipse, Xtext, ...). After this phase, the real work began.

It did not take long and I reached the limits of the internal architecture of the IDE and used technologies. The early implementation was really more like a proof of concept than a software that can be deployed to a production and used by developers at CERN. For me, the main reasons for this initial "failure" were that the original requirements were not complete and the IDE was not properly tested.

Both reasons make sense from the point of view that CERN is an enormous research organization with specific needs. The tools used at CERN have to be safe, scalable, easy to use and performant and the technologies are often pushed to their limits. So it is only logical that the first iteration in the development of the IDE was not a total success and new requirements emerged after some time.

[1] SCADA stands for "Supervisory Control and Data Acquisition".

[2] WinCC OA (previously known as PVSS) is the most commonly used SCADA system at CERN.

2.1 CERN History

CERN is the European Organization for Nuclear Research. The acronym CERN was derived from the original French name *"Conseil Européen pour la recherche nucléaire"*.

The idea of reviving fundamental research after World War II came to its live in 1952. In this year, under the auspices of UNESCO, the 12 European governments established a council for building a laboratory. The result of this council was among others a CERN convention that defined the mandate of CERN.

> The Organization shall provide for collaboration among European States in nuclear research of a pure scientific and fundamental character, and in research essentially related thereto. The Organization shall have no concern with work for military requirements and the results of its experimental and theoretical work shall be published or otherwise made generally available.

In 1954, construction of the laboratory begins. It is positioned north-west of Geneva, on Franco-Swiss border. This, from geopolitical perspective, relatively neutral position and the purely scientific nature of the research, turns out to be very important in the near future. During the Cold War, CERN is one of few places in the world, where scientists from the Soviet Union and the Western countries cooperate and work together.

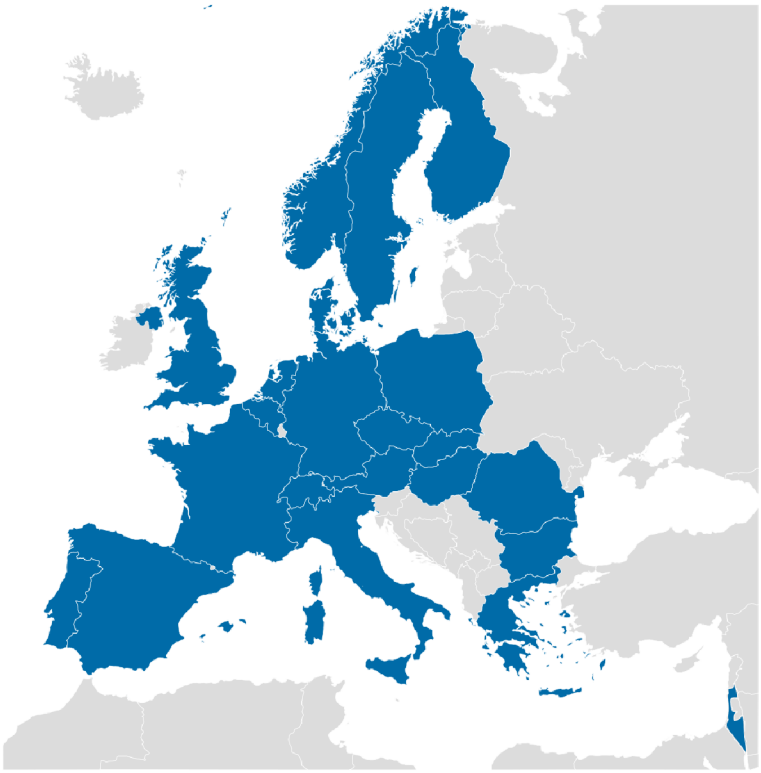

Figure 2.1: CERN member states.

[2]CERN member states. Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001-, 2015-07-31 [cit. 2016-01-05]. Available at: https://commons.wikimedia.org/wiki/File:CERN_member_states_.svg

In the 1980's, the LEP (The Large Electron-Positron Collider) and later its successor LHC (The Large Hadron Collider, started working in 2008) are built. These particle accelerators are one of the biggest and technologically most advanced machines ever created. Thanks to this machines, the new era of global particle physics begins.

Today, CERN is the world's biggest particle physics laboratory. It has 22 member states (see Figure 2.1) and over 10 000 researches of more than 110 nationalities exchange ideas and work together.

2.1.1 EN-ICE Group

The EN-ICE is one of the groups at CERN. The name stands for "Industrial Controls & Engineering" and it is a part of the Engineering Department. It also happens to be the group I have worked in as a technical student for more than one year.

This group develops solutions and provides support for control systems of various sizes (from plant-like installations down to small size laboratory systems), built on top of industry-standard equipment (see Figure 2.2), most of them are configured around CERN-defined and CERN-developed frameworks (see Figure 2.3). These control systems are often used in the technical infrastructure, research and accelerator sectors.

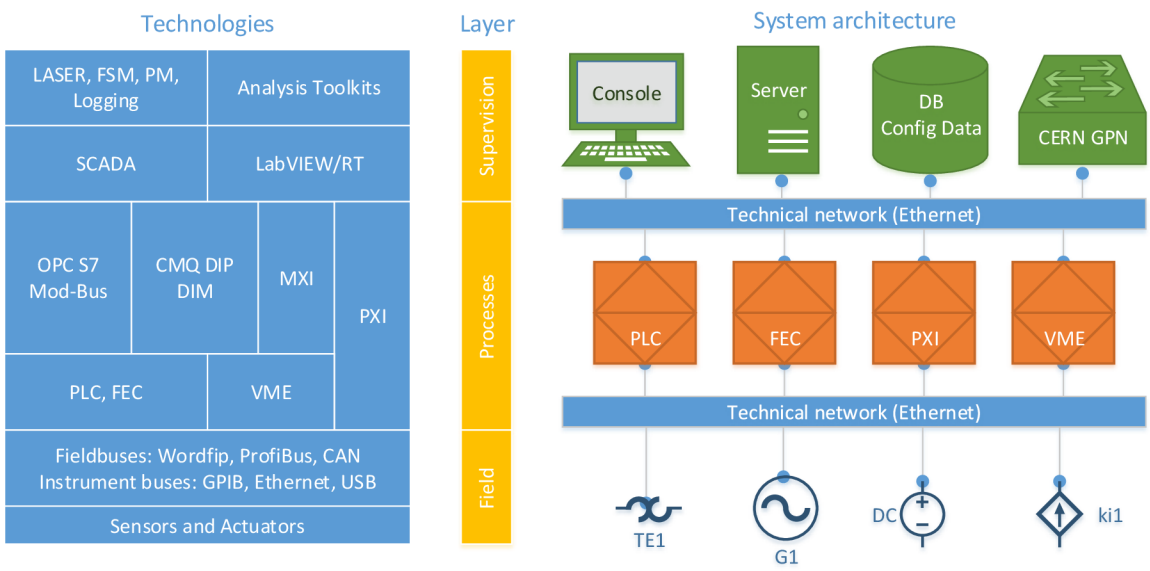

Figure 2.2: Components of control system at CERN.

Inside the EN-ICE group, there is the SCD (SCADA Systems) section. Projects, frameworks and services provided by this section are dedicated to the SCADA part of control systems at CERN. The SCD section uses SIMATIC WinCC Open Architecture (hereinafter referred to as WinCC OA) as their primary SCADA tool. The tool is a commercial product developed by ETM (currently owned by Siemens). It was chosen in 2000 by the LHC experiments and since then, it is officially supported by the SCD section.

During past years, WinCC OA became so important for certain parts of CERN that some of the engineers working there participate in its development, adding new features and fixing bugs.

2.1.2 WinCC OA

As mentioned before, WinCC OA was adopted by CERN as the primary SCADA system. There are some of the reasons behind the decision of adopting this software.

- **Openness** – WinCC OA allows programmers to use CTRL programming language and API for developing managers and drivers.
- **Scalability and robustness** – possibility to create very large distributed and redundant systems.
- **Multiplatform** – WinCC OA runs on Windows and Linux. These operating systems are the most often used at CERN. It also supports multiplatform implementations where components running on Linux and Windows can interoperate seamlessly together.
- **Relations with ETM** – CERN has a good relation with the company that develops WinCC OA. It allows CERN to keep a close eye on the development of the tool and participate in it.

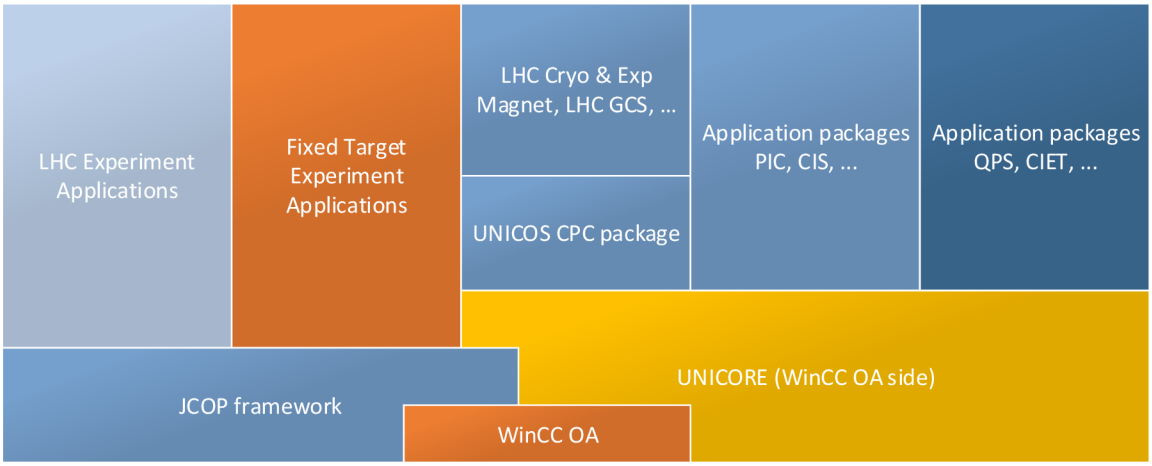

Figure 2.3: Schema of WinCC OA based frameworks and applications of the ICE group.

There are two key general-purpose programming languages that are used in WinCC OA. The first language is C++. It allows programmers to write low-level high performant code, support for new databases or create basically anything that can extend functionality of WinCC OA (drivers, driver plug-ins, managers, control extensions, external widget objects).

Since C++ is widely used programming language, it is already supported by many great and popular IDEs (Visual Studio, NetBeans, Eclipse, JetBrains CLion, . . .). So from perspective of C++ development for WinCC OA, there is no need for another IDE (or plug-in for existing IDE) that would support C++.

The second programming language called "Control script Language" (discussed in Section 3.2.2, hereinafter referred to as CTRL or CTL), is however in a different situation. Even though WinCC OA provides a code editor for this language, the editor lacks of many features commonly provided by modern IDEs. Therefore, it makes CTRL code writing not very efficient and safe.

For this reasons, CERN decided to cooperate with Eindhoven University of Technology and start a research, whose aim was to create a tool (an IDE) that would meet the CERN needs related to development for WinCC OA/CTRL.

2.2 Original IDE Requirements

The original requirements were proposed by WinCC OA developers from the SCD section. The document created for purpose of gathering the necessary information, contained 92 items, covering both functional and non-functional requirements. All the requirements were later reviewed/clarified, consolidated, generalized and grouped together as shown in Table 2.1.

Table 2.1: The original requirements of the SCD section organized in groups.

Criteria		
Functionality	Syntactical	Color coding (Syntax Highlighting) Bulk comment and uncomment ( ) and balancing Real time syntax check Statement (syntax) completion assistance Scope indication Hint provision (common errors and messages) Variable type indication Program header generation Documentation support, templates and syntax check
	Documentation	Online help and tutorial
	Debugging	Run command/functionality Class break points Line break points Exception break points Method break points Variable break points Conditional break points Deactivating break points Step in/out/over Suspend thread/process Cancel process Remote debug
Usability		Indentation Code formatting Graphic editor Code navigation Changing method signatures Call hierarchy Searching functionality “Reorderable Tabs” (window) and docking Folding/unfolding statements Method and variable usages
Performance		No requirements
Interaction with other tools		Extendable by plug-ins (OSGI plug-ins) TDD & code-coverage tools

These requirements were later analyzed, and became the foundation for series of UML and other diagrams that helped the team of developers to better understand what should be done. It also helped members of the team when they were comparing and choosing the base platform and related technologies.

2.3 Additional IDE Requirements

As often happens in IT projects, the initial requirements and specification are not very accurate and do not cover everything. On the other hand, some of the original requirements turn out to be unnecessary, useless or simply outdated.

Because of the inaccurate requirements and their dynamic nature, it was crucial for me, as a developer of the IDE, to have regular meetings. The meetings, with my supervisor Axel Voitier and the SCD section leader Manuel Gonzalez Berges, took a place usually once per week. Their outcome shaped the IDE and expanded my knowledge of WinCC OA/CTRL and the way, how the SCD developers are using it.

During the 14 months, I was working at CERN, a set of new requirements emerged, along with a feedback from the users that helped to improve the IDE (fix some bugs, increase performance and make the UI more usable and user friendly). The most significant feature requests/requirements are listed in Table 2.2.

Table 2.2: List of the most significant requests that emerged during my stay at CERN.

Requests	**Optional Description**
Parameters suggestion for built-in functions	
Run static code analysis from command line	
Create a button, which disables certain warnings	Related to Eclipse preferences window.
Quick-fixes	Undefined variable/function: propose to create it
	Explicit expression casting.
	Graphical objects in *.ctl files - quickfix.
Import organizer	Import organizer: clear unused `#uses` libs.
	Import organizer: on auto-completing function name, add the corresponding `#uses` if necessary.
Compute final type of any expression	Related to type inference.
Static evaluation of basic expressions	If an expression can be evaluated statically, do it.
Static code analysis	Static code analysis: `if/else` statements.
	Static code analysis: `switch` statements.
	Static code analysis: already assigned constant.
	Static code analysis: assigned value is not used.
	Static code analysis: uninitialized variable.
Editor of preferences for code analysis etc.	
Support values for constants	Change grammar the way that all constants have value.
Check function definition parameters	
Plug-in API for Stefan Klikovits Ph.D. project	API for CTRL code dependencies discovery.
Doxygen documentation	

Filter variables in Outline view	Allow users to filter variables in Outline view.
Create logger for Code Analyzer plug-in	
CTRL Code Analyzer – output formats	Support for Jenkins format.
	Support for JSON format.
	Support for XML format.
	Support for HTML format.
CTRL Code Analyzer - support for full JCOP	Support specific structure of JCOP project.
Help for built-in functions	Show user documentation for built-in functions.
Highlight build-in functions with different color	
Parser for *.xml (pnl) files	Open XML files in special XML editor.
	PNL $\longleftrightarrow$ XML panel files conversion.
	CTRL-XML file editor - remember last edited script when tabs are changed.
	CTRL-XML editor - reload file when it was modified in different program.
	CTRL-XML editor - button that can open *.xml *.pnl files in GEDI.
	CTRL-XML editor - code assist for graphical objects.
Unit test skeleton generator	Help CTRL developers with unit tests.
Support for evalScript() function	
Automatic „DebugXX“ template insertion	Help CTRL developers with “inline debugging”.
Remote project editing	
Make simpler „import project“	In Eclipse, create import option specific for WinCC OA.
„Go to“ functionality for libraries	“Go to” functionality should work also for libraries from outside the current WinCC OA project.
CTRL editor - „variable“ can be constant suggestion	To promote immutability, suggest programmers to make certain variable as constants.
„Validation preferences“ for CTRL Code Analyzer	
Functional programming hint - modification of non-local variable	Helps to reduce the number of global variables, which can be modified from outside.
Plug-in for „config“ file	Config - code completion.
	Config - Outline view.
	Config - Show documentation for properties.
	Config - Color picker for code completion.
	Config - make it standalone.
	Config - support loadCtrlLibs.

Config - language for „templates“	Config Templates - provide functionality for config file modification.
	Config Templates - engine for „Config“ files matching.
	Config Templates - JavaScript debugging console.
	Config Template language - support „infinite OR expression“.
	Config Template language - support number of „positive passes“ for properties.
	Config Template language - create „PropertiesRule“.
	Config Template - make it as standalone app.
	Config Template - support JSON output of execution.

At the end of my stay at CERN, there were more than 60 closed JIRA issues related to new features and around 250 improvements.

2.4 Aims of the IDE

As we can deduct from the requirements, the original goal was to create an advanced CTRL code editor and debugger connected to WinCC OA. Such editor should also inherit features like *Reorderable Tabs*, windows docking and OSGI plug-ins extensibility, from some base platform (in this particular case, the Eclipse Platform).

The IDE was later developed in such way that its functionalities covered more than just CTRL language. The most important functionalities that were not mentioned in the original requirements were the following:

- **Complex static code analysis of CTRL language**. This feature was requested because CTRL is a dynamic scripting language and there is no compiler or any other existing tool that could tell programmers that a code contains some bugs/semantically incorrect expressions and other suspicious constructs that could cause problems during runtime.

- A possibility to **run the IDE in standalone mode** on a server. The IDE could be executed from command line and able to check WinCC OA projects (analyze the CTRL code, its dependencies and correctness of WinCC OA “config file”). The result output of the analysis could be in multiple formats (JSON, interactive HTML page, XML) and it could be integrated with Jenkins and used as an input for another analysis.

- **Support for “panel files”** in XML and PNL formats. These files are used in WinCC OA for definition of panel’s GUI and they can also contain a CTRL code.

- One of the non-functional requirements was to provide an API that could be used by other research projects at CERN.

- **Plug-in for WinCC OA “config files”**. This data-driven plug-in was able to check the syntax of WinCC OA configuration files, analyze their contents (check file paths, libraries, accessibility of URLs, type and value range of values, etc.), provide code-completion, display “outline view” and make the overall process of creating and validating “config files” more efficient.

- **A new domain specific language** (DSL), designed for the needs of the SCD section was created. Name of the language was "Config Template Language", and its purpose was to become a tool for programmers that would allow them to make advanced templates.

  These templates would describe how certain "config files" should look like (a small example of the language source code is shown in Figure 2.4). Along with this language, a new Eclipse plug-in that could check syntax of the templates and execute them, was developed.

```
'''print("Hello World template!");'''
[general]
{exactlyOnce} pvss_path = "C:/Siemens/" || "/opt/Siemens"
proj_path = ''(value.toLowerCase().startsWith("c:/projects/"))''
[ui]
showActiveShapes = 0
checkADAuthIntervall = >= 60
```

Figure 2.4: Sample code written in Config Template Language.

So in the end, the IDE became a complex tool, a tool that made certain aspects of WinCC OA development process more efficient and safe.

Chapter 3

Theoretical Base

This chapter provides readers of this thesis a brief introduction in the theory that is good to know in order to understand the practical parts in the following sections. The topics discussed here do not cover all the theory that was used and some of the subjects, considered by me as less important in the context of this work, were omitted. The missing topics and more information can be found in the books and resources listed at the end of the thesis.

3.1 Integrated Development Environments

Integrated Development Environment, commonly known as IDE, is a software tool that facilitates software development. It usually consists of some kind of editor (either code editor, graphical editor or some mixture of both), project/code navigation panel, debugger and test/build tool.

Since the main purpose of IDE is to make programmers more productive and efficient, it usually contains many advanced features that are helpful during the program development. For instance, type hierarchy window, debugging, conditional breakpoints, outline view, code analysis, code refactoring and code completion.

Apart from IDE parts mentioned before, we can often see features that support version control (integration with SVN, Git, Mercurial, ...), file system browsing, connection to databases and application servers and programs deployment. Often occurring and powerful feature is a possibility to extend the IDE itself or underlying platform by another plug-ins.

3.1.1 Plug-ins and IDE Platforms

As mentioned, IDEs often support plug-ins. Plug-ins are software components that extend functionality of another application. The functionality of plug-ins varies. A plug-in, in context of IDEs, may add a small feature or it can provide complex set of features that support a whole programming language (a nice example of such comprehensive plug-in is "IntelliJ Scala plug-in" or "PyDev for Eclipse").

Thanks to plug-ins, some IDEs like Eclipse, NetBeans, IntelliJ or Visual Studio, can be used for various tasks, support multiple programming languages or be tweaked for some special needs of a developer group. Stable and extensible architectures of IDE platforms like the NetBeans Platform or the Eclipse RCP are also used as the base for other desktop applications, completely unrelated to IDEs.

Here is what the official NetBeans website says about their platform: *"The NetBeans Platform provides a reliable and flexible application architecture. Your application does not*

have to look anything like an IDE. It can save you years of development time. The NetBeans Platform gives you a time-tested architecture for free. An architecture that encourages sustainable development practices. Because the NetBeans Platform architecture is modular, it's easy to create applications that are robust and extensible."[1]

3.1.2 Comparison of IDEs

Because the plan is to create the IDE on top of another IDE (i.e. inherit some functionality and reuse as many software components as possible), we need to compare already existing IDEs and their platforms, in order to see, which one best meets the CERN's requirements. Apart from functionalities, we also have to take into consideration licenses, used technologies/programming languages, platform developer[2] etc. (these attributes are later compared in Table 3.1). It is important to realize that all decisions taken at the beginning of development process can have major impacts on the future of the project, so choosing the right platform is a crucial task that should not be underestimated.

The IDEs for comparison were selected from the top of the list of the most popular IDEs for October 2015. The list was published on the following address: http://pypl.github.io. "Vim" was excluded because it is more a text editor than a regular IDE in the context of this thesis.

3.1.2.1 Eclipse

Eclipse was originally created to support Java development. The original codebase is from IBM VisualAge. It is multiplatform, it can be easily extended by plug-ins and the Eclipse environment is highly customizable. The extensibility was one of the main reasons why Eclipse has become so popular. The Eclipse community has created many plug-ins that support widely used languages (C/C++, PHP, Python, Ruby, JavaScript, HTML and many more) and allow users to tweak Eclipse as they need. Over the years, Eclipse has become, thanks to its plug-ins, a universal tool, primarily used for software development.

Eclipse is released under the Eclipse Public License. This open source license is replacement for the Common Public License and it is used by the Eclipse Foundation for their software. It also allows the receiver of EPL-licensed programs to use, modify and distribute the programs (or their modified versions) for free and without any obligation to publish the modifications. From this point of view, this license is well suited for internal development in companies and can be safely used by CERN.

The graphical part of Eclipse is built on top of SWT (Standard Widget Toolkit) framework. This framework was originally created for the needs of Eclipse. At the time of the framework development, GPUs in computers were not as powerful as today, and the alternative framework (Swing) was not fast enough. The advantage of SWT over Swing was the rendering speed. SWT is essentially a layer that calls native code of underlying OS. In the past, such solution was faster than Swing, which uses only functionalities provided by JVM (therefore Swing was slower). Another important implication of calling native code of underlying OS is that SWT applications have native look. This may seem to be an advantage, but in reality it causes more problems than it solves. Firstly, SWT has to be

[1] The NetBeans Platform. NetBeans IDE [online]. Redwood City, California, United States: Oracle Corporation, 2016, 2016-01-05 [cit. 2016-01-05]. Available at: https://netbeans.org/features/platform/index.html

[2] We need to confident that the platform developer will not kill the project any time soon.

implemented for each platform[3]. Secondly, the GUI looks differently and the behavior of UI elements may not be the same on each platform. And lastly, such inconsistencies may confuse users.

3.1.2.2 Visual Studio

Microsoft Visual Studio (abbreviated as VS) was originally used for development of Microsoft Windows applications and web sites/services. Currently, the set of features of VS is much bigger, and VS supports also development for mobile devices (not only Windows Phone but also Android), cloud apps, databases, the whole family of Windows devices and languages like Objective C, Ruby, Python, M, JavaScript, TypeScript and F#.

VS comes in 5 editions: VS Community, VS Professional, VS Enterprise, VS Test Professional and VS Express. The Community and Express editions are free. The difference between these two editions is that the Express edition lacks many features included in the Professional version. It also does not support extensions (plug-ins) and it is focused only to one area of development (Visual Basic, Visual C++, Visual C#, Visual Web Developer and Express for Windows Phone). The Community edition of VS is on the other hand very similar to Professional version and it supports various languages and extensions[4]. The only limitation of this free version is that it can be legally used only by individual developers or small teams (maximum is 5 people in team). In the case of CERN usage, the license is not such a big problem, because CERN is non-commercial organization and it has the license for VS already bought.

As for the UI part of VS, it is based on WPF (Windows Presentation Foundation). WPF is generally considered as a stable and mature framework, but the small issue is that WPF applications can be currently run only on Windows. Linux, FreeBSD, Mac OS X or any other operating systems are not supported, and it seems that even though .NET is becoming more and more open source and multiplatform, there are no official plans (only rumors) to make WPF open source and available on other platforms[5].

3.1.2.3 NetBeans

NetBeans was originally a Czech IDE based on the NetBeans Platform. It is written in Java (primary for Java development) and its modular architecture allows other developers to extend it by plug-ins.

The NetBeans Platform is a framework developed around Java Swing. Its main purpose is to simplify development of complex, extendable and maintainable desktop applications with well-defined architecture. An advantage of applications built on top of the NetBeans Platform is that they can run on all the operating systems supported by JVM and Swing.

Because the platform uses Swing and Swing is completely written in Java and therefore OS independent, the look & feel and the behavior of UI components can be the same on all the operating systems (all widgets are emulations of native widgets). This is the most noticeable difference compared to SWT. Another difference is more internal. Swing uses heavily Model-View-Controller (MVC) design pattern. This design pattern separates the

[3]Since we are not developers of the Standard Widget Toolkit, we do not have to worry about this as long as our platform is supported.

[4]Extensions for Visual Studio are basically the same thing as plug-ins for Eclipse and NetBeans.

[5]Also Mono has intention to support WPF, "At this point, no group in the Mono project has plans to implement Windows Presentation Foundation APIs as part of the project.". http://www.mono-project.com/docs/gui/wpf/

data itself from the way how it is represented to users and how users interact with the data. It is common to create UI using this pattern (or little bit different pattern called MVVM in case of WPF) and many of Java/.NET programmers are used to it. SWT on the other hand does not implement any of the previously mentioned patterns and writing UI using SWT tends to be painful even compared to Swing, or to WPF/JavaFX (personal opinion based on my experience with all the UI frameworks).

One of the big advantages of NetBeans is that the whole project is open-sourced and since fall of 2007, NetBeans is licensed under a dual license of GPL version 2 and CDDL (Common Development and Distribution License derived from Mozilla Public License). This means that NetBeans is free to use for almost anyone, and the license is compatible with the planned usage at CERN.

Table 3.1: Eclipse, Visual Studio and NetBeans compared together.

	Eclipse	**Visual Studio**	**NetBeans**
Operating systems	Windows, Linux, Mac OS X, Solaris	Windows only	Windows, Linux, Mac OS X, Solaris
License	Eclipse Public License	Proprietary , license depends on version	CDDL/GPL2
Based on	Java SE, SWT	.NET, WPF	Java SE, Swing
Common languages for plug-in dev.	Java, Xtend	C++, C#	Java
Module system	OSGi	VSPackage	OSGi and NetBeans-specific module system

3.1.3 Selection of the Platform for WinCC OA IDE

Selection of the platform for the IDE was done by the team at TUE. The team was deciding between Eclipse and NetBeans, the two most popular platforms that can run both on Windows and Linux. At the end of the decision process, Eclipse was chosen as the foundation for the future work.

The TUE documentation states that the reasons behind this decision were the following (each reason is commented by me):

- *"Lower learning curve" – "The learning curve is lower with Eclipse as most of the people are used to Eclipse IDE"*
  - This reason seems to be legit at first glance, but from my point of view, it is not. The fact that some software (let us name it "A") is more popular than some other software (let us name it "B"), does not automatically make the learning curve for "A" lower. Especially in the case when "A" and "B" are very similar (Eclipse and NetBeans are very similar in the way how they are used by users and in terms of their internal architecture).
- *"200+ active projects on top of the Eclipse Platform"*
  - This is a fact, probably found on Eclipse website in section "Eclipse Newcomers FAQ": https://eclipse.org/home/newcomers.php

  - The documentation does not say, why this is an advantage compared to NetBeans (it does not present any related information for NetBeans).

- *"Rich open source community support"*
  - Again, this argument is hard to measure and the documentation does not show any evidence that NetBeans community is "more closed" or "less supportive".

- *"The Eclipse Platform has more options for IDE developing. Xtext and IMP are the most popular plug-ins for IDE developing in Eclipse"*
  - It is hard to prove (and authors of the documentation do not even try it) that "Eclipse Platform has more options for IDE developing". On the other hand, the number of options for development is not really relevant. We should care about quality of the tools, their documentation and whether they meet our needs. From this point of view, Xtext seems to be a reasonable choice. A choice that is not available on NetBeans.

It may be arguable, whether the justification of the selection was objective and rigorous enough, or whether the arguments were quasi-arguments and the motivation for choosing Eclipse was that the team at TUE had more experience with Eclipse than with NetBeans or any other platform. Either way, at the time I was involved in this project, it was already too late and too impractical to rethink such fundamental decision, therefore the development continued for the Eclipse Platform.

3.2 Programming Languages

To fully understand the content of following chapters, it is needed to have an elemental knowledge of what are programming languages and what is the theory the languages are based on. For this reason, a supplementary chapter was written (see Appendix B). The chapter describes formal basis used by parsers/compilers and classification of commonly used programming languages. Here is a quick overview of the chapter:

- **Formal languages** (Chomsky hierarchy) – what is the hierarchy
  - **Regular languages** – description, usage, types of finite-state automata
  - **Context-free languages** – description, usage for parsing, power of context-free languages
  - **Context-sensitive languages** – description, example of context-sensitive language, comparison to Turing machine
  - **Recursively enumerable languages** – description, Turing completeness

- **Classification of real-world programming languages** – choosing the right programming paradigm/style may significantly simplify the solution of a certain group of problems. For this reason, it is important to know some of the main programming paradigms and classification of programming languages. This knowledge is also needed when designing a new programming language (especially domain-specific language (DSL), whose purpose is usually to solve only a limited number of problems related to some domain, practical example of a DSL is in Chapter 5.5). This supplementary subchapter shows what are the commonly known classes of programming languages and a few examples of language classification.

From this point onwards; a programming language, as a term used in this thesis, is meant as a language with well, formally, defined syntax and semantics that can be processed by a Turing machine[6]. If the word *language* (or *programming language*) is used in this thesis, it will usually refer to a high-level programing language that is used by programmers to create algorithms, which can be executed by computers.

3.2.1 Config Template Language

As a part of this thesis a new declarative[7] domain-specific language was created. The purpose of this language is to serve as a tool for verification of WinCC OA configuration files correctness.

The language was designed with respect to the learning curve of the users and their general knowledge of programming. It also endorses concepts that make the code more bug resistant; immutability and independency of verification rules.

We will not discuss here any syntactic and semantics details of the language, this is all covered in the thesis attachments (there is the whole language specification[8]) and addition information can be found in my paper submitted to the Excel@FIT conference[9]. The detailed information about the design and implementation of the Config Template plug-in can be found in sections 4.5.3.1 and 5.5.

3.2.2 Control script Language

"Control script Language", often shortened as CTRL or CTL, is a general purpose programming language used for WinCC OA development. It *"is designed to have certain extensions to support building and operating SCADA systems (view, control, archive), it has specialized functions to handle datapoints, graphical interface, etc."*[40]. The syntax of this language is greatly inspired by C language, but unlike C, it is not compiled, but interpreted and the nature of the language is much more dynamic.

3.2.2.1 Differences between CTRL and C Language

Even though the syntax of CTRL is very similar to C, there are some significant differences. For example, CTLR lacks pointers and structures. Therefore, CTRL programmers have to use `mapping` type, which behaves like associative array (see Figure 3.1) and thus it does not provide an explicit information of what is the structure of the data stored inside `mapping` type variable.

[6] A Turing machine is a mathematical model of computation. It can manipulate symbols on an infinitely long tape according to a finite set of rules (this is an informal, simplified explanation).

[7] Even though the Config Template Language is primarily declarative, it allows programmers to use also other paradigms.

[8] See file Config_template_language_description.docx

[9] http://excel.fit.vutbr.cz/submissions/2016/002/2.pdf

C code

```c
struct Person
{
    char firstName[50];
    char secondName[50];
    char email[100];
    int age;
};

struct Person peter21;
strcpy(peter21.firstName, "Peter");
strcpy(peter21.secondName, "Potter");
strcpy(peter21.email, "Peter.Potter@example.com");
peter21.age = 21;
```

CTRL code

```
mapping peter21;
peter21["firstName"] = "Peter";
peter21["secondName"] = "Potter";
peter21["email"] =
"Peter.Potter@example.com";
peter21["age"] = 21;
```

Figure 3.1: Difference between C `struct` code and CTRL `mapping` construct code.

Another major difference is that CTRL contains the `eval` function. This function behaves similarly as the `eval` function in JavaScript. Because of this function, it is sometimes hard for programmers to determine, how some parts of a CTRL code work and it also makes harder for the IDE to analyze CTRL code in general.

3.2.2.2 Advantages of CTRL Language

List of CTRL language features that are, by some users at CERN, generally considered as good:

- Easy to learn because it is simple and similar to C.
- Can run on Windows and Linux.
- Programmers do not have to allocate or free memory. Memory management is done automatically behind the scenes.
- Possibility to extend language functionality using extensions written in C++[40].

3.2.2.3 Disadvantages of CTRL Language

List of CTRL language features that are by people at the SCD section, generally considered as bad:

- No type definitions. Programmers have to operate with a limited set of data types.
- Performance.
- CTRL language contains some inconsistencies (for example, getters/setters/properties of "graphical objects").
- The lack of some language constructs (structures, enumerators, lambdas, pattern matching,...) and presence of "eval" function makes CTRL code unnecessary long, less clean and for programmers harder to express their intentions[40].
- Too much dynamic languages are not suitable for such important field as control systems[10]. The CTRL code almost never crashes because of type mismatch, the interpreter performs implicit conversions/casting with, sometimes unexpected, results.

[10] My personal opinion based on programming/SW development experience.

- The system of exceptions is not well designed.

3.3 Processing of Programming Languages

In theory, the processing of a source code written in some programming language, consists of several steps. Each step transforms one representation of the source code to another representation (see Figure 3.2).

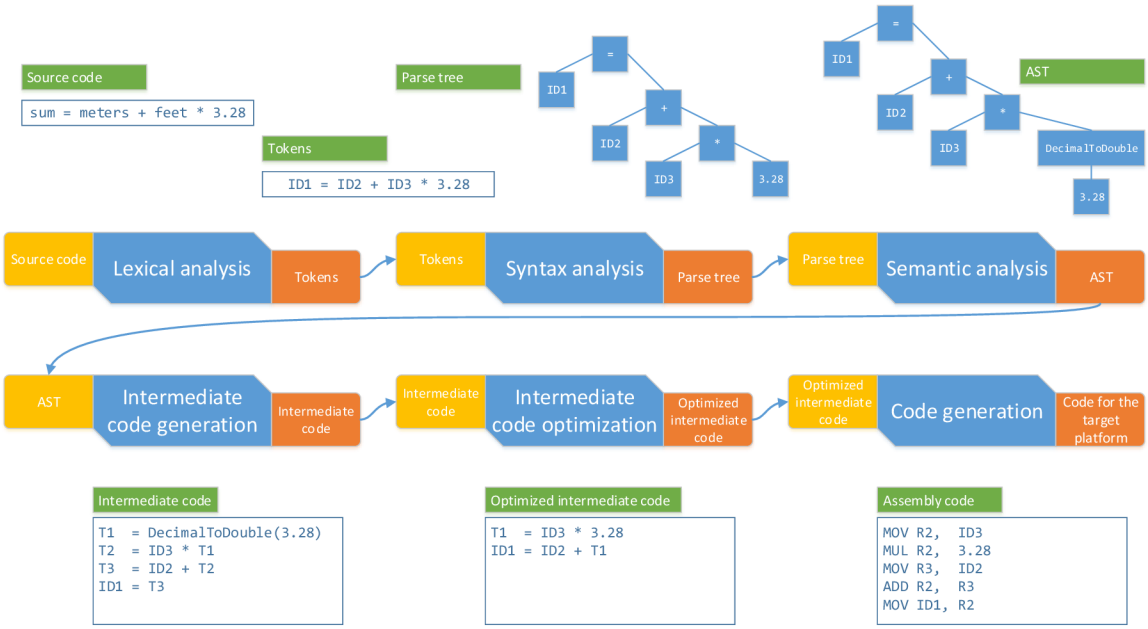

Figure 3.2: Traditional schema of compilers.

1. **Lexical analysis** – this process takes a stream of characters (source code) and produces series of strings with certain meaning (so called tokes). For example, strings like `"class, int, for, while, if, else, var"` can have meaning of "keywords", and strings represented by regular expression: `('a'..'z') ('a'..'z'|'0'..'9')*` can represent a name of `variable/function/class/method`.

2. **Syntactic analysis** – is performed by a component called *parser*. A parser takes as an input a series of tokens produced by a lexer (the component that is performing lexical analysis) and applies to them a set of rules that define the syntax of a programming language. The result of this process is a parse tree (tree-like data structure) that reflects the syntax rules used by the parser.

   - If the source code is syntactically incorrect, the parser can detect it and tell the programmer, what syntax rule has failed and what token caused it to fail (therefore the programmer has a feedback, where in the code is the problem and what is the probable cause).

3. **Semantic analysis** – if the source code had correct syntax, another analysis that checks the meaning of the source, follows. This analysis does not control the syntax[11],

[11] Although it is not common for semantic analysis to control the syntax, there may be cases when a

it usually includes checks whether variables/functions/classes were properly declared and whether certain data types and operations can be used together. The result of this operation is an abstract syntax tree (AST, sometimes also simply called *syntax tree*).

- One of the common examples is that many strongly typed programming languages do not allow programmer to assign boolean values to variables of type float etc.

4. **Intermediate code generation** – this is an optional step, but some compilers create a simple intermediate representation of source code/AST. Such source code representation should be easy to generate from an AST and at the same time, relatively close to the target code (on the other hand, the representation should be still hardware-agnostic). Also a simple representation is more suitable for optimization.

   - An example of an intermediate code is three-address code (often abbreviated as TAC or 3AC).

5. **Intermediate code optimization** – process of intermediate code optimization is trying to change the intermediate code in such way that the resulting native code will have improved certain characteristics. Usually we optimize a code in order to make it run faster, but there are also cases when we want it to occupy less memory during runtime, or consume less energy (this is especially important when it comes to portable devices like smartphones).

6. **Code generation** – this is the last step. It generates code in a language that is close to the hardware level. It is often/traditionally a machine code or some kind of assembly language code[12]. At the end of this process, the result should be executable on target platform.

In practice, the number of the steps can be reduced (some steps are merged together, other can be done automatically) and the whole process is simplified[13]. It is common that developers use special tools that generate lexer, parser and basic semantic analyzer for a language defined by a grammar. Additionally, such tools may provide means that encapsulate other processes like proper semantic analysis, static code analysis, code generation and allow developers to extend the capabilities of the generated language processor.

One such tool that uses grammars to generate lexers and parsers, and provides a framework for programming languages development, is called Xtext. It is an open-source framework, originally created for the Eclipse Platform. Compared to software like ANTLR, JavaCC or Yacc, Xtext is not only a parser generator, but also integrates well with Eclipse, IntelliJ IDEA and Orion (web-based code editor) platforms, and offers features that programmers expect to have when writing a source code in an IDE:

programmer can exploit this process to perform additional syntax checks. This may happen in cases when a parser-generating software is used the grammar language is not suitable for describing certain aspects of the language syntax.

[12]Note that technology has evolved so it is common that the result of compilation process is a bytecode for some virtual machine or some FPGA/ASIC hardware.

[13]On the other hand, really advanced computer languages and their compilers/virtual machines can be extremely complex.

- **Refactoring** – this feature includes renaming of variables, functions and classes (basically anything that is a *Cross-Reference*).
- **Go To Declaration** – the cursor is moved to the position, where the given entity is declared.
- **Template Proposals** – allows programmers to insert commonly used blocks of code. For example, iterating over an array or list is a construct often used in imperative languages, and an IDE can generate and insert such code automatically. Figure 3.3 shows how the template for Java iteration statement looks like.

```
for (int ${index} = 0; ${index} < ${array}.length; ${index}++) {
        ${array_type} ${array_element} = ${array}[${index}];
        ${cursor}
}
```

Figure 3.3: Example of an Eclipse code template for Java language.

- **Content Assist** – by default, Xtext offers a code completion for terminals, which are specified in grammar and their insertion will not make the source code syntactically incorrect.
- **Find References** – the ability to find all references of given variable/function/class or in the case of DSLs, some exotic construct that has references in the source code.
- **Quick Fixes** – this feature gives programmers means to easily fix a broken code. Xtext does not have any default implementation of Quick Fixes (the grammar does not specify, how to fix a syntactically or semantically broken code), but it provides an interface for DSL developers, so this feature can be easily implemented.
  - Examples of quick fixes are for instance: "**explicit casting**" (an IDE can perform type inference and explicitly cast an expression to correct data type) and "**adding import statements**" (if a class or function was not imported, Quick Fix can try to find where it is declared and add corresponding import statement).
- **Outline View**, **Syntax Coloring** and **Bracket Matching** are also supported by Xtext and have a default implementation.

The following subsections of this chapter describe some of the steps that are usually needed, when implementing a basic support for a programming language. Note that the list of steps includes only those that are related to Xtext and lead directly from *"having a source code in the new language"* to *"executing the program"*.

3.3.1 Grammar Definition

The first step, when using a parser generator, is to define a grammar of the programming language we want to parse. It is basically the corner stone for parser generators.

In case of Xtext, the grammar language is a domain-specific language[14], whose purpose is to describe another textual language. Besides using the Xtext grammar language for

[14]Domain-specific language aka DSL

syntax description, it is used for mapping the syntax to a model. The model is in-memory object graph, based on EMF[15] Ecore. In the context of Xtext, this model is also our AST.

3.3.1.1 Xtext Grammar Example

As an example, I have created a simple Xtext grammar that defines the syntax of a DSL. The main purpose of this DSL is to describe a state machine (i.e. states and transitions).

```
1.  grammar marecek.thesis.xtext.example.fsmdsl.FSMdsl
2.     with org.eclipse.xtext.common.Terminals
3.
4.  generate fSMdsl "http://www.thesis.marecek/xtext/example/fsmdsl/
    FSMdsl"
5.
6.  Statemachine :
7.    {Statemachine}
8.     'states' '{'
9.            'start' startState=State
10.           states+=State*
11.           ('fail' failState=FailState)?
12.    '}'
13.
14.    'transitions' '{'
15.           transitions+=Transition+
16.    '}'
17. ;
18.
19. State:
20.    'state' name=ID final?='is final'?
21. ;
22. FailState returns State:
23.    'state' name=ID
24. ;
25.
26. Transition:
27.    'with' character=CHAR 'go from' fromState=[State] 'to'
    toState=[State]
28. ;
29.
30. terminal CHAR: "'" . "'";
```

Figure 3.4: An example Xtext grammar that represents a finite-state machine.

The example grammar (see Figure 3.4) starts with the language name declaration. The name is similar to the naming of packages in Java. It is like this because the generated parser and other classes are Java code, and the code will use this name. Usually we want the name to be unique, in order to avoid any name conflicts[16].

[15] EMF stands for Eclipse Modeling Framework

[16] All the generated classes will be in the following Java package: `marecek.thesis.xtext.example.fsmdsl.*`

The second line says, that our *FSMdsl* extends another Xtext-grammar/language called: `"org.eclipse.xtext.common.Terminals"`. This Terminal grammar already defines few things that are commonly used by many programming languages. For instance, it includes terminals for single-line comments, multi-line comments and IDs. It means that we do not have to explicitly say in our grammar that everything that starts with `"//"` is a single-line comment and that IDs are defined by the following regular expression:
`'^'?('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*`.

As for the line number 4, it is related to `EPackage` generation and we will not cover this topic here. For more information, please read the Xtext and EMF documentation.

The interesting part starts at line 6. Here we specify a rule that represents the highest level of our grammar. It also says that the generated AST will have the root node of type Statemachine[17]. The whole Statemachine rule ends at line 17 and it contains another two mandatory language constructs: states and transitions.

Grammar part responsible for FSM state definition starts at line 8 and ends at line 12. It basically says that the first string has to be `"states"`, then an arbitrary number of invisible characters is allowed and then `{` character is expected.

Next mandatory string is `"start"` (line 9) and it is followed by a State rule, which is assigned to the `startState` feature. Therefore, if the source of a FSM is syntactically correct, an instance of State class will be created and it will be stored in the AST property called `startState`.

After the first mandatory definition of a start state, we can define arbitrary number of other states. This is expressed by `states+=State*`. At runtime, when we work with the AST, the instance of `Statemachine` class will have a property called `states`. This property is a list of all the non-starting states.

The last part of states definition is on line 11 and it is optional. The optionality is expressed by `?` character at the end of the line.

The part of the grammar representing transitions is based on similar concepts as the previous part. The only difference is that by `+` character at the end of line 15, we express that the FSM source code has to contain at least one transition. If no transition is declared, then the generated compiler marks the source code as syntactically incorrect.

The `State` rule starts at line 19. It says that every state declaration has to begin with `"state"` keyword and then a state name follows. The name is an ID string, which was imported from Terminal grammar we are extending. The last part of the line 20 tells us that a declared state may be final. The optionality is again expressed by `?` character at the end of the line. But in order to make the AST easier to work with, we used `?=` assignment. This is a special kind of assignment, which means that if the declared state is final, the `final` feature of the `State` class instance will have true value. Otherwise the feature will have false value.

We skip the description `FailState`, because it is just a `State` rule that cannot be final, and we will have a look at the `Transition` rule. Again the beginning of line 27 uses the concepts already explained. But there is also a new concept related to cross references. The meaning of a `[State]` rule in square brackets is that the names of `States` used in `Transition` rule, are references and they are linked to `name` property of declared `States`, i.e. if we want to use a state of certain name, we have to first declare it.

The last part of the grammar shows how a terminal can be defined. For terminal definition in Xtext grammar, we use the `"terminal"` keyword followed by the name of the

[17]The full name is `marecek.thesis.xtext.example.fsmdsl.fSMdsl.Statemachine` and it extends *EObject* class.

terminal[18] and ":". Then terminal itself is specified using a regular expression.

Source Code Example As an example of source written in the FSMdsl language (see Figure 3.5), we will use a FSM that represents the following regular expression: `ab(ab)*`.

```
// This FSM represents "ab(ab)*" reg. expr.
states {
        start state Start // here we start

        state A
        state B is final

        fail state Fail // if we fail, we go to this state
}

transitions {
        with 'a' go from Start to A
        with 'b' go from A to B
        with 'a' go from B to A
}
```

Figure 3.5: An example of source code written in the FSMdsl language.

3.3.1.2 Grammar Languages from Practical Point of View

From a practical point of view, there are few things I have learned about grammars, that are worth mentioning. All of them were practically used in my project, while working at CERN.

The first is that a grammar does not have to always fully describe the syntax of the language. This is because the grammar language we are using may not be suitable for description of certain aspect of the syntax. In such case, we can create a grammar, that is not as strict as it should be. Therefore, the grammar describes a language that is a superset of the original language and the generated parser accepts also strings that are not syntactically correct. In order to reject the code that is not according the original language specification, we take the parse tree/AST and perform additional checks so we can confirm that the parsed code is correct and meets the language-syntax specification.

The second thing related to grammars and parser-generators is that the bigger/more complex the grammar is, the slower is the parser. In real-word applications, we have to balance between effectiveness of our solution and effectiveness of developing such solutions. In this case, it means that we can write relatively fast parsers by ourselves or we can easily generate slower one from a grammar. Fortunately, there is a compromise solution that does not force us to choose only the first or the second approach. We can write a grammar that is not as verbose as it should be, and then for the things that the generated parser does not do, we write an optimized code by ourselves.

The last thing that is worth mentioning is related to the order of rules in grammars. By changing the order of rules, we can, in certain cases, increase the performance of generated parsers. This is because grammars are usually translated into some high-level language (in

[18] Terminal name, in Xtext, has to be in capital letters.

case of Xtext/ANTRL it is Java) and the source code of generated parsers usually contains a lot of `if-else/switch` statements (see a sample code of the CTRL parser in Figure 3.6). If we reorder the conditional statements in such way that the conditions that are most often evaluated to `true`, are evaluated sooner than the rest of the `if` conditions, we may speedup the parsing process at least a little bit[19].

[19]Note that this optimization method is mostly statistical and also depends on the code we are parsing. Using this technique, we may get some speedup for some type of source code, but we may also lose the performance in other cases.

```
if ( (LA62_0==87) ) {
                alt62=1;
            }
            else if ( (LA62_0==88) ) {
                alt62=2;
            }
            else {
                if (state.backtracking>0) {state.failed=true; return
current;}
                NoViableAltException nvae =
                    new NoViableAltException("", 62, 0, input);

                throw nvae;
            }
            switch (alt62) {
                case 1 :
                    // ../ch.cern.winccoa.ctl/src-gen/ch/cern/winccoa/
ctl/parser/antlr/internal/InternalCtl.g:4007:5: this_EQ_0= ruleEQ
                    {
                    if ( state.backtracking==0 ) {


newCompositeNode(grammarAccess.getEqualityOpAccess().getEQParserRuleCa
ll_0());

                    }
                    pushFollow(FOLLOW_ruleEQ_in_ruleEqualityOp8031);
                    this_EQ_0=ruleEQ();

                    state._fsp--;
                    if (state.failed) return current;
                    if ( state.backtracking==0 ) {

                                    current.merge(this_EQ_0);

                    }
                    if ( state.backtracking==0 ) {

                              afterParserOrEnumRuleCall();

                    }

                    }
                    break;
                case 2 :
```

Figure 3.6: Source code sample of generated CTRL parser.

3.3.1.3 Parse Tree vs AST

While working with parsers, it is important to understand, what is the difference between parse tree (also called derivation tree or syntax tree) and abstract syntax tree (AST).

The most noticeable difference is, that the structure of parse tree reflects the syntax of the parsed language/grammar we have used to generate the parser. AST is on the other hand a higher-level representation, and it does not contain every single detail of the syntax. For example, *grouping brackets* does not usually occur in AST, because the tree structure of AST makes them implicit. The same applies for semicolons at the end of statements.

- We know that if a tree node represents, for example, a `while` statement, then the children of this node represent statements in the while-loop body. No additional information about brackets is needed.

If we look at ASTs produced by Xtext generated parsers, we find out that even though, they look like tree data structures, they are actually graphs. The reason is that Xtext performs scoping and linking. This means that if we have an expression containing a variable and the variable was properly declared and linked, we can jump in the AST to a completely different place or even jump to a model, which represents source code of some other file.

Another difference between Xtext-AST and classical AST is that Xtext-AST is closely related to Xtext grammar language. So in practice, Xtext-AST is somehow half-way between parse tree and AST, and it is not even a real tree, but graph. This AST design has some advantages and disadvantages.

One of the advantages is that Xtext has a class called `NodeModelUtils` and its static methods allow us to access low-level representation of models. It is particularly handy when we need a detailed information about the source code and position/offset of certain nodes. The possibility to dig deeper than ordinary AST allows and seek the details, was often used in *Quick Fix* implementation for CTRL language.

As for the disadvantages, the most evident is that the structure of Xtext-AST reflects the grammar. The alignment of model and grammar is fine in most cases, but there are exceptions. One such exception is the representation of `if-else` statements. Xtext grammar language has problems with description of `if-else` statements (it is called the *dangling else problem*) and model created by generated parser is usually hard to work with. The problem is demonstrated in Figure 3.7, Figure 3.8 and Figure 3.9.

```
SelectionStatement:
   type='if' '(' e=Expression ')'
               s+=Statement (=> 'else' s+=Statement)?
  | type='switch' '(' e=Expression ')' s+=Statement
;
```

Figure 3.7: Grammar of CTRL language responsible for conditional statements parsing (example of the dangling else problem).

```
int positiveFunc (int x){
        if (x < 0)
                return x*(-1);
        else if (x == 0)
                return 1;
        else
                return x;
}
```

Figure 3.8: An example code written in CTRL language. The in-memory model of this source code is shown in Figure 3.9.

selectionStatement	SelectionStatementImpl (id=13465)
e	RelationalExpressionImpl (id=13471)
eContainer	StatementBlockImpl (id=13472)
eFlags	-65532
eStorage	Adapter[2] (id=13473)
s	EObjectContainmentEList<E> (id=13474)
data	Statement[4] (id=13479)
[0]	ReturnStatementImpl (id=13481)
[1]	SelectionStatementImpl (id=13482)
e	EqualityExpressionImpl (id=13488)
eContainer	SelectionStatementImpl (id=13465)
eFlags	-196604
eStorage	Adapter[2] (id=13489)
s	EObjectContainmentEList<E> (id=13490)
data	Statement[4] (id=13494)
[0]	ReturnStatementImpl (id=13497)
[1]	ReturnStatementImpl (id=13498)
[2]	null
[3]	null
dataClass	Class<T> (ch.cern.winccoa.ctl.ctl.Statement) (id=2754)
featureID	2
modCount	2
owner	SelectionStatementImpl (id=13482)
size	2
type	"if" (id=13491)
[2]	null
[3]	null
dataClass	Class<T> (ch.cern.winccoa.ctl.ctl.Statement) (id=2754)
featureID	2
modCount	2
owner	SelectionStatementImpl (id=13465)
size	2
type	"if" (id=13475)

Figure 3.9: Model created by Xtext generated parser (CTRL grammar). The model demonstrates the dangling else problem.

Another major disadvantage is, that if we change the grammar, our algorithms that rely on the AST model structure, may break. For this reason, it is extremely important to make the grammar solid so it does not have to be changed later. Any later changes of the grammar may cause bugs to emerge in our source code and unexpected behavior at

runtime.

3.3.2 Semantic Analysis

Many languages used in IT have a specification that describes not only the syntax, but also the semantics[20] and what conditions/rules need to be met in order to declare a source code as correct. Basically we can say that semantic analysis follows parsing and it performs checks that ordinary language grammars cannot do.

The most common form of checks are type checks and "declaration checks" (i.e. checks controlling whether an entity was properly declared).

Type checks follow the rules of language specification. We can have simple rules, which specify what data types can be assigned to what variables or what operations and data types can be used together without explicit casting, etc. But the complexity of checks can go even further. A language can define rules related to object inheritance and it can also specify a context, where a different set of rules should be applied.

A good example of a programming language that is context sensitive, is Xtend and its `switch` statement. This language is shipped together with Xtext and it has some nice features, which make working with AST models simpler and more elegant compared to Java. One of the nice features is already mentioned `switch` statement. This statement combined together with *type guards*, changes the semantics of "guarded variables". I was using this feature heavily in my projects, when I needed to dig deeper in an AST model structure. A real-world usage of the `switch` statement is shown in Figure 3.10.

[20] As an example of a language specification, we can mention *"C# Language Specification"*. This specification covers, on more than 500 pages, all the aspects C# and it is free to download on Microsoft MSDN website.

```
public static def List<Variable> separateVariables(EObject node) {
   val List<Variable> toReturn = new ArrayList();
   switch(node){
      Variable:{
      // the following list can contain objects of type Variable
      // switch statement changes the meaning of "node" variable
from EObject to Variable
         toReturn.add(node);
      }
      DeclaringList:{
         // .item is specific for DeclaringList type
         if(node.item != null){
            // rest of the code
         }
      }
      VariableDefinition:{
         // .v is specific for VariableDefinition type
         if(node.v != null){
            // rest of the code
         }
      }
      FunctionDefinition:{
         // .pl is specific for FunctionDefinition type
         if(node.pl != null){
            // rest of the code
         }
      }
      IterationStatement:{
         // .doeo is specific for IterationStatement type
         if(node.doeo != null){
            // rest of the code
         }
      }
   }
   // rest of the code
}
```

Figure 3.10: Demonstration of complex type checking and context dependency in Xtend language.

One last example of a language that relies heavily on advanced semantic analysis is F#. This impure-functional language has rather simple syntax[21], but a powerful type system (see the type system demonstration in Figure 3.11) and type inference based on *Damas-Milner's Algorithm W*. Because of F# relative simplicity, immutable data types and strict rules defined in the specification and checked by the F# compiler, the language is generally considered as a safe choice for scientific and financial computations[22], and it is getting more

[21]There is also a web page named *"F# syntax in 60 seconds"*. This page demonstrates how easy it is to understand F# syntax, http://fsharpforfunandprofit.com/posts/fsharp-in-60-seconds/

[22]There are two books that discuss F# usage in science and finances: *"F# for Scientists"* (by Jon Harrop, isbn:1118210816) and *"F# for Quantitative Finance"* (by Johan Astborg, isbn:1782164634).

and more popular in the past few years[23].

```
type Shape =
    | Triangle of A : float * B : float * C : float
    | Rectangle of height : float * width : float
    | Circle of radius : float

let matchShape shape =
    match shape with
    | Rectangle(height = h) -> printfn "Rectangle with length %f" h
    | Circle(r) -> printfn "Circle with radius %f" r
    | Triangle(a, b, c) -> () // either we have to match all the possibilities
explicitly
 // | _ -> () // or we have to use statement takes care of all the remaining
possibilities
 (*
    If the match..with expression doesn't handle all the possible values, the
compiler will not compile the code.
    Notice that we don't have to specify that "shape" expression is of type
Shape.
    F#'s powerful type inference is able to deduce the type automatically.
 *)
```

Figure 3.11: Demonstration of strict rules in F# specification and type inference capabilities.

3.3.2.1 Static-type Checking

As already mentioned, it is usual that general-purpose languages have specified constraints for types. The process, which verifies that data types used in source code/program are within the defined constraints, is called *type-checking*. If type-checking is performed at runtime, then it is termed as *dynamic type-checking*. Otherwise if a type safety verification is done during compilation, it is termed as *static type-checking*. And since this chapter is primarily focused on semantic analysis and compilers, we will discuss mostly static type-checking.

Let us start with the advantages of static form of type-checking. The most important benefit is that checking a program at compilation time or continuously, while writing source code, can detect many bugs early in the development cycle. And the sooner a bug is found, the easier and less expensive it is to fix it.

Another advantage is code optimization. When compiler knows about data types of variables and values passed to/returned from functions, etc., it can prove that the program does not need any additional type safety checks, and the result (binary) code may be optimized and run faster.

From programmer's perspective, statically typed languages combined with an IDE, make source code refactoring easy and relatively safe. For example, if we change the return type of a function/method, the IDE/compiler will tell us, what part of the code was broken by this change. If we do the same in source code written in a language with dynamic type-checking, we would have to manually find all the places affected by the change, and if we miss something, the program may crash at runtime (or behave in an unexpected way). The

[23]Source of F# popularity data: http://www.eweek.com/developer/microsofts-f-language-number12-with-a-bullet.html

difference between languages that can detect invalid code at compilation time or at runtime (and crash because of it) is shown in Figure 3.12.

The disadvantage of static type-checking is that it generally makes compilers slower. On the other hand, dynamic typing does not slow down compilers so much, and it also allows interpreters to load source code at runtime and use eval function[24].

As regards CTRL language, it is considered to be dynamically typed. It is because the language is interpreted (there is no compiler that would make type-checking) and it allows all kinds of implicit casting/conversions. The impossibility to automatically check the code before runtime was one of the reasons, why the research at TUE started and why the IDE was developed, i.e. to analyze CTRL code during development process and detect certain classes of bugs related to incorrect data types.

```
------ F# code ------
type Dog(name) =
    // changing any of the "member names" would break the code at compilation
time
    member this.name = name
    member this.haf = printf "Haf haf, I am %s" name

let getAnimal name = new Dog(name)
    // new Cat(name) would also cause compilation errors

[<EntryPoint>]
let main argv =
    let dog = getAnimal "Dája"
    dog.haf // if we change "haf" to "meow", the compiler will complain
    0
------ JavaScript code ------
function Dog(name) {
    this.name = name;
    this.haf = function () { // if we change the method name, we break the code
        print("Haf haf, I am " + this.name);
    };
}

function getAnimal(name) {
    return new Dog(name);
    // return new Cat(name); // if the function returns Cat, it would break the
program at runtime
}

var dog = getAnimal("Dája");
dog.haf(); // if a Cat object is returned, the program crashes when the code is
executed (cats do "meow" not "haf")
```

Figure 3.12: This examples demonstrate, how easy it is to introduce a bug in code that performs type-checking at runtime and what is the level of resilience against such class of bugs in languages with strict type system and static type-checking.

[24]It is possible to have an `eval` function even in statically typed languages, but it is not common and such functionality requires advanced uses of algebraic data types.

3.3.3 Code Execution

In the context of practical programming, code execution is the ultimate goal we want to achieve. It is the reason, why we write programs and why some people spend so much effort with designing a programming language and a developing tools for it.

There are two ways, how a program can be executed. We either interpret it or compile it into a machine language. Both approaches have their positives and negatives, and as we will see later in this chapter, there are technologies that combine interpretation and compilation together.

3.3.3.1 Compilation

Compilation is, generally speaking, a process that converts source code from one form/language into another. In case of many traditional languages like C/C++, the final result of such conversion is an executable/library file[25] in a machine code.

There are several advantages of traditional compilers (plus assemblers and linkers) that produce machine code. Firstly, the machine code can be executed directly by underlying hardware. It means no unnecessary overhead compared to interpreters. As a bonus, compilers often perform optimizations so the resulting code runs faster on the target platform.

A big weakness of machine code is portability. The machine code has to be usually produced for each platform separately. It is not simply possible to build a machine code for x86 architecture and run it directly on ARM processors.

Compared to interpreters, the deployment of machine code is also simpler. Assuming that we have all the binary files needed (compatible with our platform), we can, in many cases, just copy the binary files and run the program. Interpreted programs require on the other hand an interpreter that has to be first installed.

As regards the application areas, compiled languages are often suitable for small embedded systems and hard real-time applications. The reason is that machine/native code is typically faster than interpreted one, consumes less memory and the performance is stable. For example, using Python for hard real-time applications is not advised, since the garbage collector (GC) can run at any time during the program execution and the duration time of garbage collecting cannot be predicted.

3.3.3.2 Interpretation

We say that a program is interpreted, when it is executed by another computer program, so called interpreter. Interpreters are thus an additional layer that abstracts the hardware.

To execute a program, interpreters use typically one of the following strategies:

- AST interpretation (Dartmouth BASIC, old versions of Lisp).
- Intermediate code interpretation (Ruby, Perl, Python).
- Just-in-time compilation[26] (.NET Framework).

It is not a rule that an interpreter has to use any of these strategies or just one of them. In fact, some interpreters combine multiple strategies. For instance, Java Virtual Machine[27]

[25] To make the text more compact and focused mainly on compiling, I have omitted an information about object code, linking and other details related to these topics.

[26] Just-in-time compilation – aka JIT.

[27] Java Virtual Machine is commonly known as JVM.

has the ability to interpret Java bytecode and also perform just-in-time compilation. The advantages and disadvantages of interpreters are highly dependent on the strategies and underlying technologies they are using. In this chapter, we will not go through all the possible interpreters, their characteristics, positives and negatives, we will focus on one concrete example that is extensively used.

The example is the .NET Virtual Machine. I chose it, because it well illustrates how modern virtual machines work, and in certain aspects, it is very similar to the Java Virtual Machine, therefore readers can get a glimpse of what is going on when the IDE is running.

.NET Virtual Machine – CLR Common Language Runtime aka CLR is a virtual machine that manages the execution of programs written for the .NET Platform. It provides features like assembly loading, memory managements (GC), thread synchronization, exception handling and security. All these features can be used by any programming language that targets CLR, and many of the languages can also seamlessly communicate with each other. This was achieved by defining Common Language Specification (CLS, see Figure 3.13) and compiling all the .NET languages to Common Intermediate Language (CIL/IL[28]).

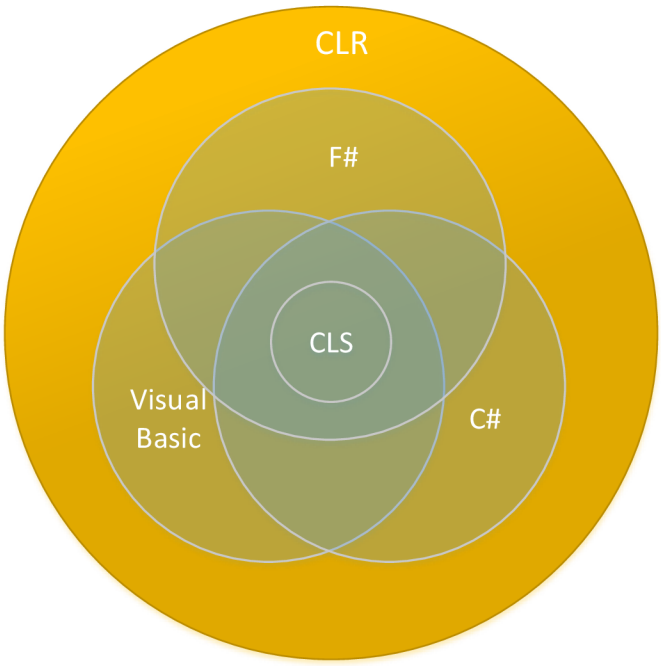

Figure 3.13: All .NET compatible languages use features from CLR and offer a superset of Common Language Specification.

As for the execution itself, the virtual machine (CLR) loads assemblies, which are made of platform-agnostic[29] CIL and metadata, and then converts the CIL into native CPU instructions. The compilation of CIL is done by a JIT compiler at runtime[30], and the JIT

[28]The terminology in .NET world is a little bit confusing and unstable (it is changing over time). One of the examples is the name CIL/IL (Common Intermediate Language and Intermediate Language). Different sources use different names and even Microsoft is not consistent in this case.

[29]Platform-agnostic - in sense of hardware and operating system.

[30]It is possible to compile .NET programs before execution and then deploy already compiled code. The

compiler can optimize the code so it runs faster on the available hardware (CPU specific instructions etc.).

The overall process of code execution in CLR (summarized in Figures 3.14 and 3.15) is, of course, much more complicated. For those who want to learn more about the technology and internals of CLR, there is a book called *CLR via C#* and source code of .NET Core Runtime available on GitHub (https://github.com/dotnet/coreclr).

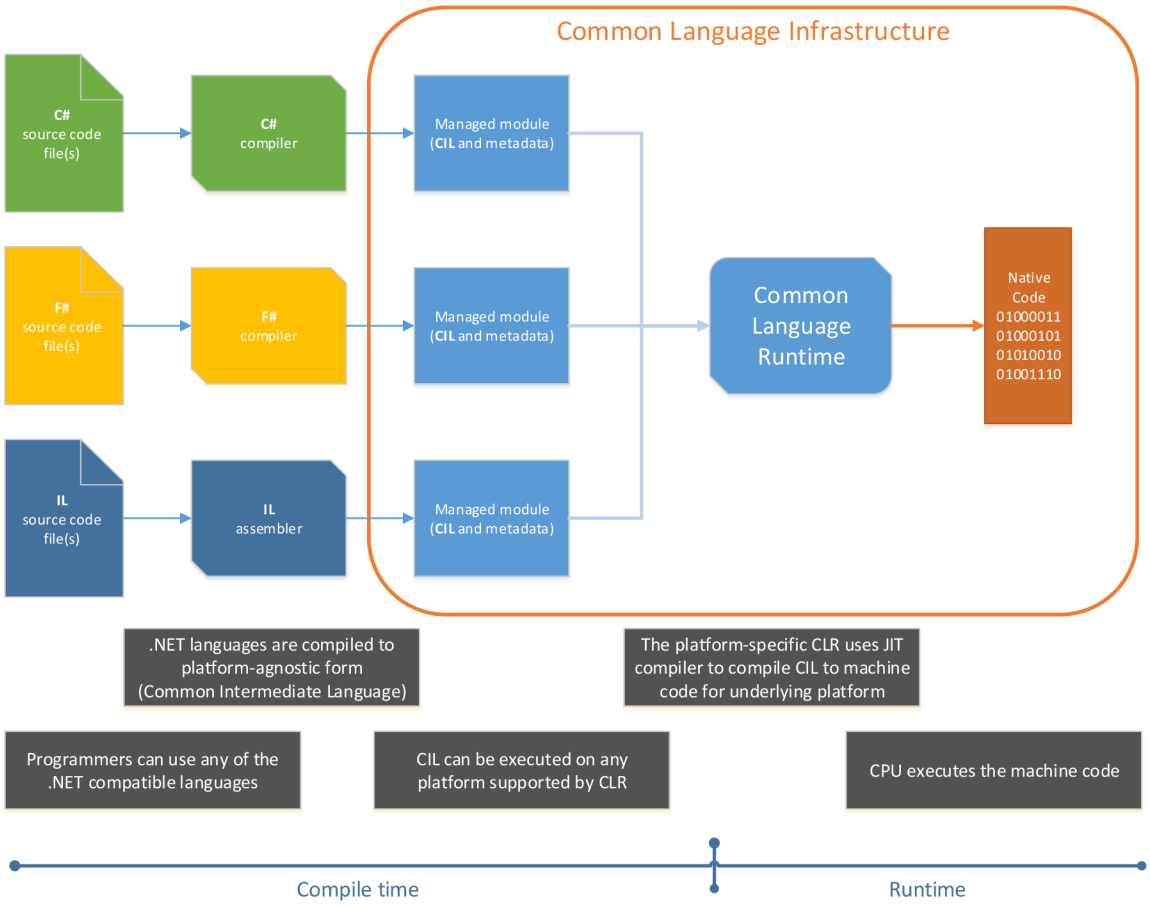

Figure 3.14: High-level schema of the .NET Platform.

disadvantage is that the compiler cannot use CPU specific instructions and the program is on average slower. The advantage is that the startup of pre-compiled programs is faster, since JIT compilation is not needed anymore.

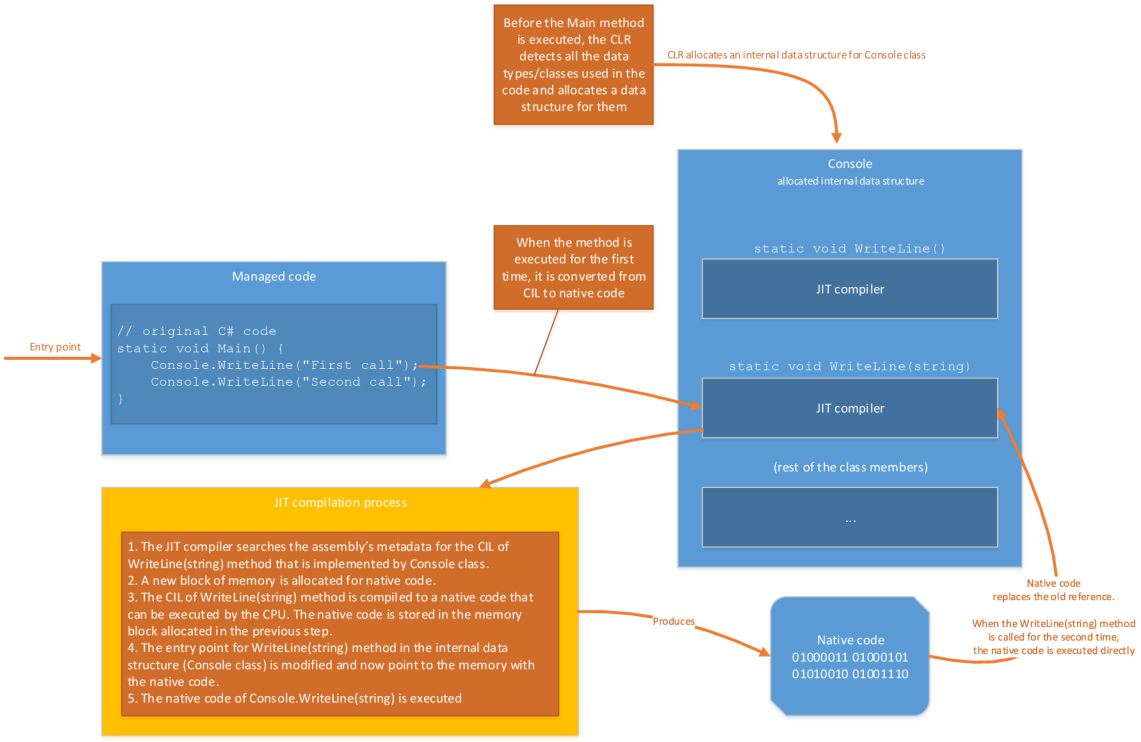

Figure 3.15: A demonstration of what is happening in the CLR when a method is called for the first time.

The advantages of the .NET Platform are the following:

- **Good performance** – since the CLI is compiled into native machine code, the program can run directly on hardware and the performance is comparable to Java/JVM and C++/GCC. Furthermore, the JIT compiler (applies both to .NET and Java) can use CPU specific instructions to optimize the code, and the virtual machine can have statistics about the runtime and make some additional optimizations (this is something, what compiled C++ code cannot do).
- **Platform independent** – many of the core technologies used at .NET are standardized and can be used for arbitrary platform (see project Mono[31]).
- **Compile once – run everywhere** – all .NET compatible languages are eventually compiled to platform-agnostic CIL. This means that we need to compile the code only once and then we can deploy it on any platform with compatible CLR.
- **Variety of programming languages** – .NET (and JVM) are not limited only to one programming language, and so .NET developers can often choose the language that suits their needs. Also it is usually easy to interact with code written in other languages.
- **Garbage collection** – garbage collection is a process that cleans up the memory that is not used by the program anymore[32].

[31] Mono is an open-source, partial implementation, of .NET technologies. For instance, it includes C# compiler and CLR that can run on multiple platforms (Windows, Linux, OS X, Android, Xbox 360).
[32] CLR allows both managed and unmanaged memory access.

As for the characteristics that are considered as negative, here are the major ones:

- **Not truly multiplatform** – .NET was originally created for Microsoft Windows and the Windows-only development continued for many years. Lately (2015) the .NET Framework was partially open-sourced and .NET Core Runtime[33] presented to developers community. This was a major step for pushing the technology to other operating systems, but it is still not on as many platforms as Java.

- **Slower startup** – when a code runs for the first time it has to be compiled by JIT compiler. This problem with the delay was partially solved with the *Native Image Generator* (Ngen.exe) that compiles all CIL code into native code before application start-up. As for Java, it has no such possibility to generate a native code before that application starts. On the other hand, JVM does not have to use JIT all the time. It can directly interpret the bytecode so there is a speedup when rarely used pieces of code are executed and not compiled to the native from.

[33].NET Core Runtime aka CoreCLR.

Chapter 4

Design of the IDE

After gathering and analyzing all the functional and non-functional requirements from CERN, it was time to design the IDE. The design is an important part of the whole development process. It sets the course of the future work and it has a big influence on the user experience.

The entire design contains not only the architecture of the IDE, but also related technologies, programming languages and programming styles[1]. So in order to create the IDE, we need to know what is the architecture of all the software we are going to use, how to connect individual pieces together and what are the characteristics of the technologies.

4.1 Overall Decomposition

It was crucial for the IDE to put it into the context of usage at CERN. We had to clarify the relations between the Eclipse Platform, the plug-ins that needed to be developed or installed in order create the WinCC OA IDE and connection to SIMATIC WinCC Open Architecture and the rest of WinCC OA/SCADA related tools used at CERN.

[1] For example, the Config plug-in was designed and written in such way that functionality like semantic analysis, code assist and editor-provided documentation was completely data-driven.

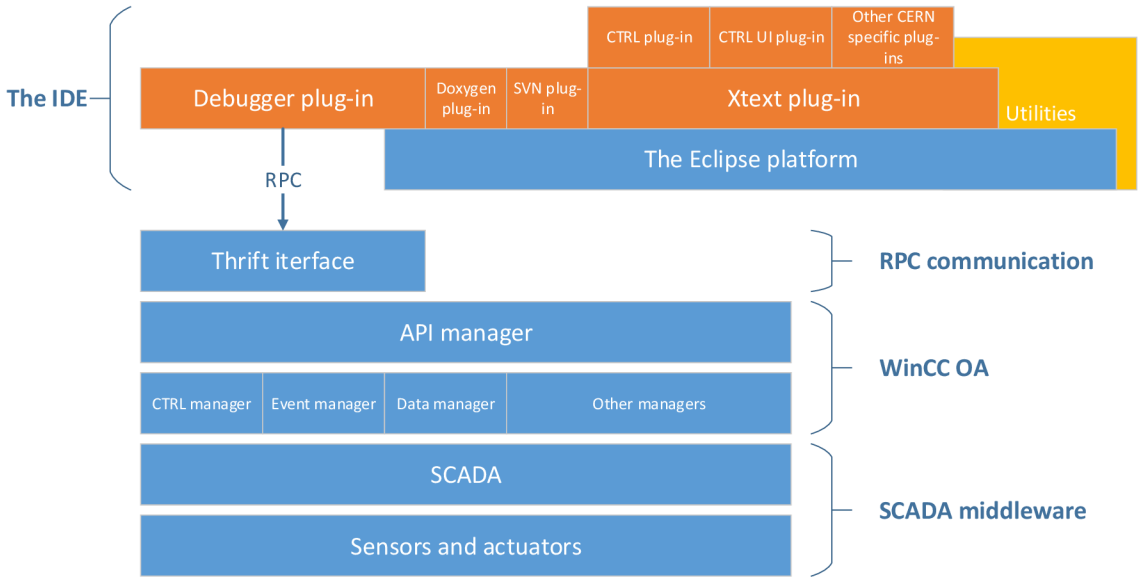

Figure 4.1: High-level strucutral decomposition of the IDE and related components.

The most important requirements for the IDE were related to CTRL programming. The IDE should increase the productivity of developers writing CTRL code, and help them to manage the code. Basically the IDE should be comparable to other modern development environments.

So ultimately, it was decided to build the new IDE on top of an existing platform. A platform that would give us the possibility to use stable architecture and inherit some functionality (project management, windows docking system, installation of additional plug-ins, search & replace etc.). The selected platform was the Eclipse Platform and the framework for CTRL language integration was selected Xtext.

As shown in Figure 4.1, the whole high-level design is split into several parts. All the parts are hierarchically structured and they are aligned in such way that they naturally fit together. Here is a brief description of the layers in Figure 4.1:

- **The IDE** – the IDE itself is a collection of plug-ins developed for the Eclipse Platform. Besides the plug-ins developed at CERN, the IDE contains also third-party plug-ins that extend the functionality.
- **RPC communication** – RPC is the way, how the IDE Debugger communicates with the actual WinCC OA CTRL debugger and interpreter. The communication is based on the Apache Thrift, which is a software framework for cross-language (Java for the IDE and C++ for WinCC OA Debug manager) service development.
- **WinCC OA** – WinCC Open Architecture gives the possibility to build managers (in this case, the manager responsible for debugging) and use the WinCC OA API.
- **SCADA middleware** – SCADA software built on top of WinCC OA platform.

4.2 The IDE Architecture

As already mentioned, the IDE itself is based on the Eclipse Platform. It is essential to know what is the architecture of the platform (see Figure 4.2) and what are the basic

concepts it uses.

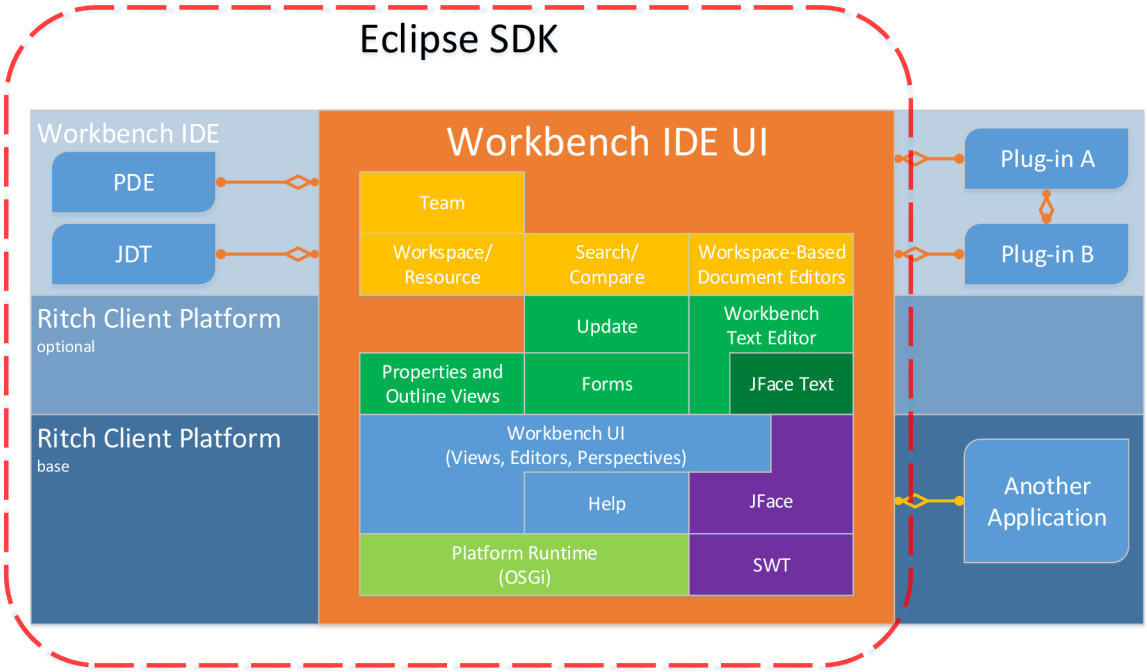

Figure 4.2: The Eclipse architecture.

When a user starts an Eclipse instance, the Platform runtime searches through the Eclipse plug-in folder and inspects the plug-ins it contains. Every Eclipse plug-in has an XML manifest. A manifest is a file with declared dependencies, aka connection points. Connection points define what connections are required by a particular plug-in and what connections the plug-in provides.

When the Platform runtime examines the plug-ins, it decides, which one to load. It is good to know that not all plug-ins are loaded during the start-up. This lazy loading strategy has multiple advantages. The first one is the speed. The Eclipse IDE itself is loaded as soon as possible, and so it does not slow down users so much. The second advantage is that lazy loading saves computer memory and therefore it makes Eclipse less resource-demanding[31, 32, 38].

4.2.1 The Workspace

From the perspective of Eclipse plug-in developers and Eclipse IDE users, the concept of Workspaces plays an important role.

For users, the Workspace is just a regular folder with projects. Besides the user's projects, the workspace folder also contains *".metadata"* folder with Eclipse-related settings, configuration and history of changes of all the workspace resources. Of course, users can have as many workspaces as they wish to have, and the workspaces can be completely independent of each other.

From the perspective of Eclipse plug-in developer, the workspace is in control of the user's resources management. Every registered listener interested in changes in workspace, gets notified when a resource in the workspace is modified. This allows developers to create Eclipse plug-ins that keep track of changes in the user's source code and react to these

changes. Eclipse plug-ins can also tag certain projects with a *"project nature"*. The project nature tells Eclipse that the project is a special kind of project. The plug-ins that use the project nature are for instance some Xtext based plug-ins or the Java development tools[2].

4.2.2 The Workbench

The Workbench is the graphical part of Eclipse. It is the user interface based on SWT/JFace (so it has native look & feel). It displays toolbars, menus, perspectives, views and editors (see the screenshot in Figure 4.4).

The structure of the Eclipse Workbench (Figure 4.3) is the following: on the highest level, there is the WorkbenchWindow. This is the application window of Eclipse running instance. If the instance is running on Microsoft Windows, it has *minimize*, *maximize/restore down* and *close* buttons in top right corner.

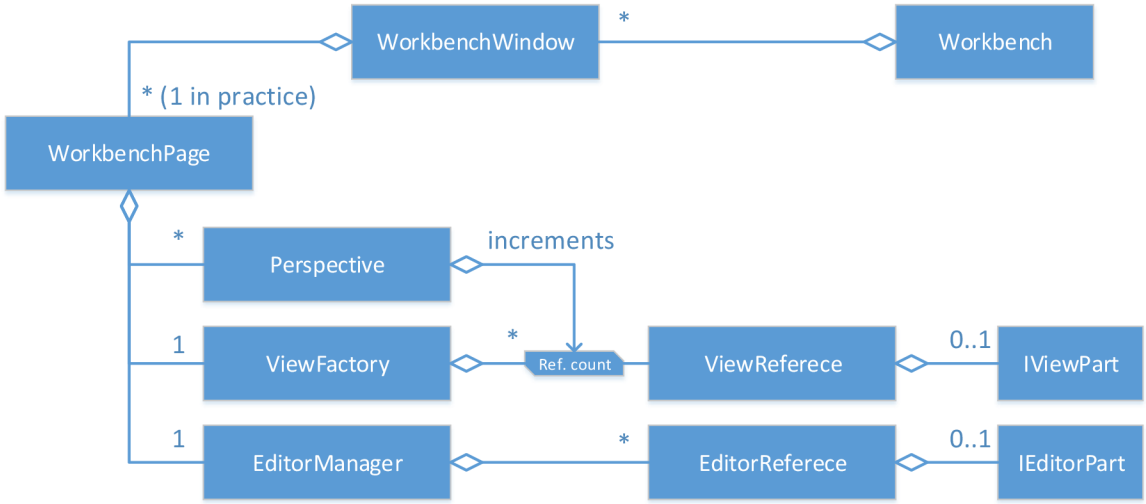

Figure 4.3: Basic structure of the Eclipse Workbench (class ownership).

On the lower level, there is the *WorkbenchPage*. Its purpose is to supply the window contents. Theoretically, there can be an arbitrary number of WorkbenchPages in a WorkbenchWindow. Practically, there is almost always only one instance of the WorkbenchPage.

If we go down in the hierarchy, we see that the *WorkbenchPage* holds one instance of `EditorManager`, one instance of `ViewFactory` and a *set of perspectives*. For now, we will focus on the Eclipse perspectives.

From user's point of view, a perspective contains editor (e.g. Java code editor, CTRL code editor, etc.) and views (Outline view, Package Explorer etc.). In reality, perspectives own only a layout, and the WorkbenchPage manages the views and editors.

As for editors and views, they are related to `IEditorPart` and `IViewPart` interfaces. Implementations of these interfaces, the actual views and editors, are instantiated lazily when they are really needed.

[2]More information about the Eclipse project natures can be found in the Eclipse documentation: http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2-FresAdv_natures.htm

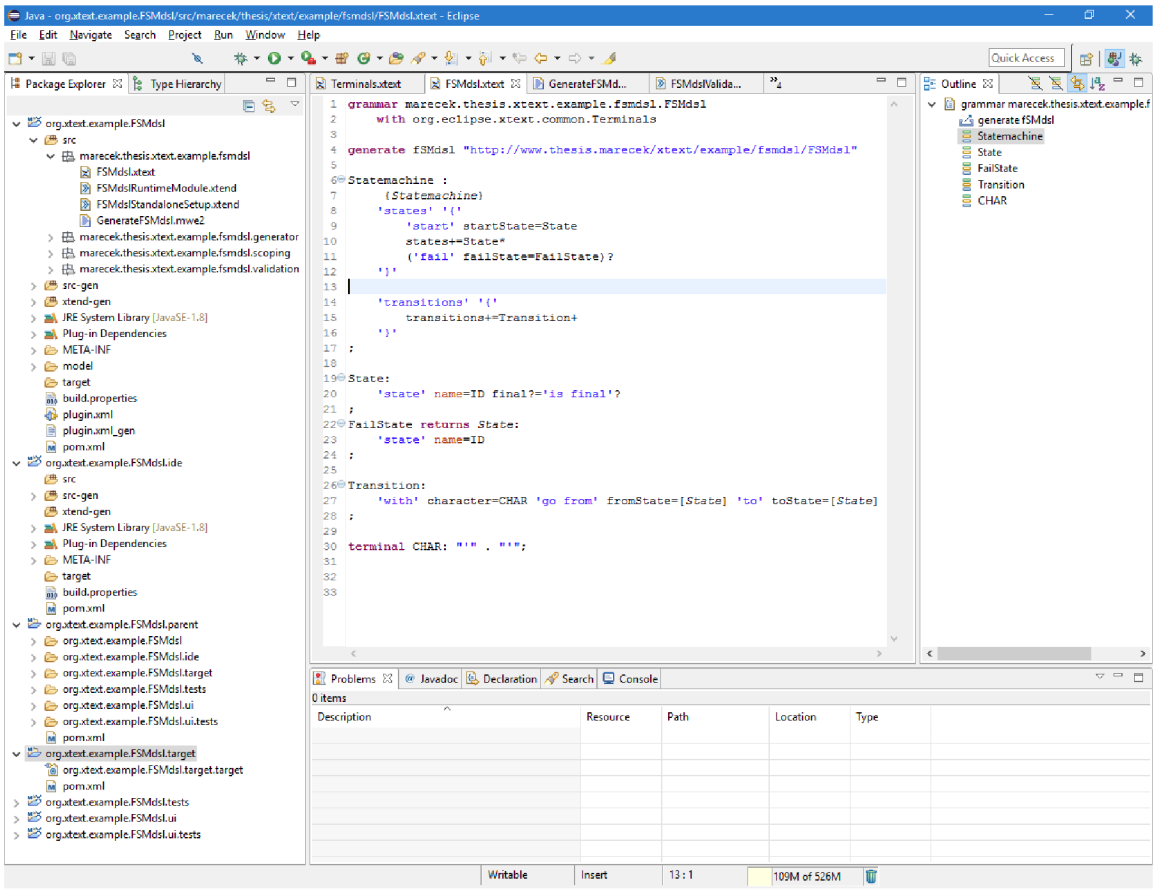

Figure 4.4: The Eclipse Workbench. The screenshot shows the default layout when working with Xtext project.

4.2.2.1 The WinCC OA IDE Perspective, Views and Editors

The IDE (the plug-ins developed at CERN + third-party plug-ins; see Figure 4.5) is aligned with the principles and architecture mentioned in previous chapter. The CTRL plug-in has its own perspective, which gives users the default layout, suitable for WinCC OA projects development, and it brings new views and code editors.

When it comes to Xtext, it offers an excellent interoperability with the Eclipse Platform. By default, it gives developers standard implementation of Outline view, syntax coloring, code formatting, code assist, search tool and many more. The framework itself is deeply integrated with Eclipse, and at the same time, it separates developer from writing "an Eclipse code". Therefore, developers do not need to know a lot about the Eclipse Platform, until they have to develop something that Xtext does not support[3]. Then the knowledge of the Eclipse architecture and principles, plays an important role, and developing Eclipse code blindly, without that knowledge, may not end well[44].

[3]During the IDE development, there was a problem with CTRL libraries that were outside the Eclipse workspace.

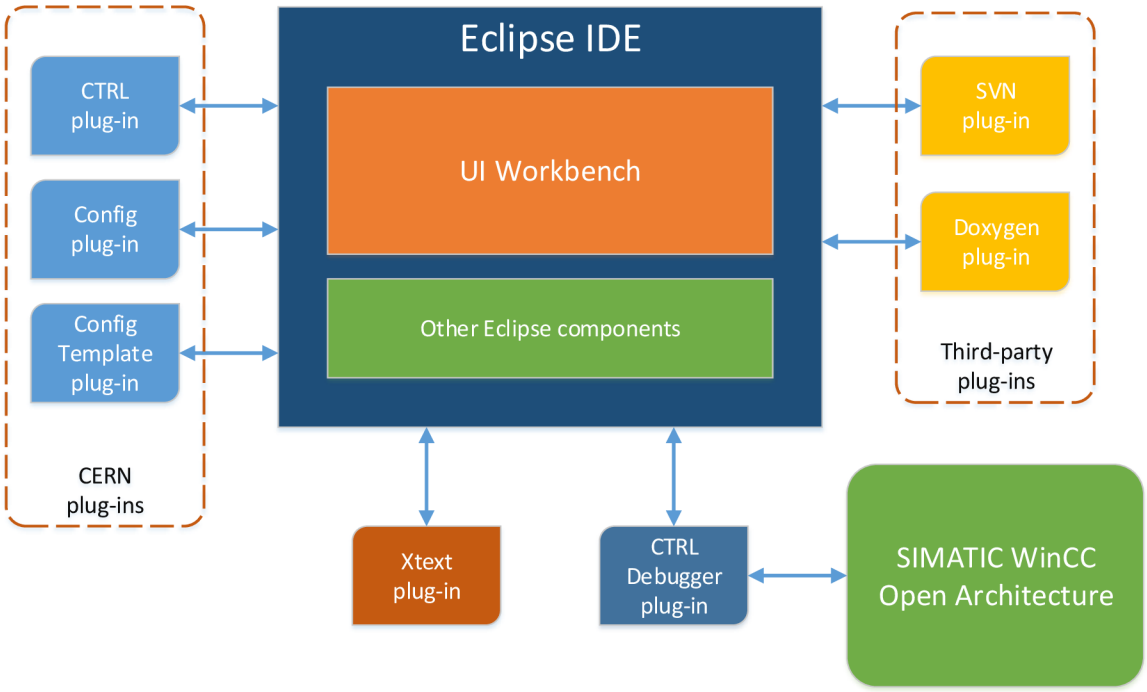

Figure 4.5: Concrete architecture of the IDE.

4.3 Functional Decomposition of the IDE

Functional decomposition of the IDE is based on the functional requirements already discussed in Chapters 2.2 and 2.3.

On the highest level, we can classify the functional requirements as:

- **General** – provided by default by Eclipse or provided by some third-party plug-in.
- **CTRL code related** – CTRL code can be either explored, analyzed, edited, executed or debugged (see Figure 4.6).
- **Config files related** – Config files can be explored, edited and analyzed in the context of current WinCC OA project.
- **Config Template related** – Config templates can be edited, debugged and executed.

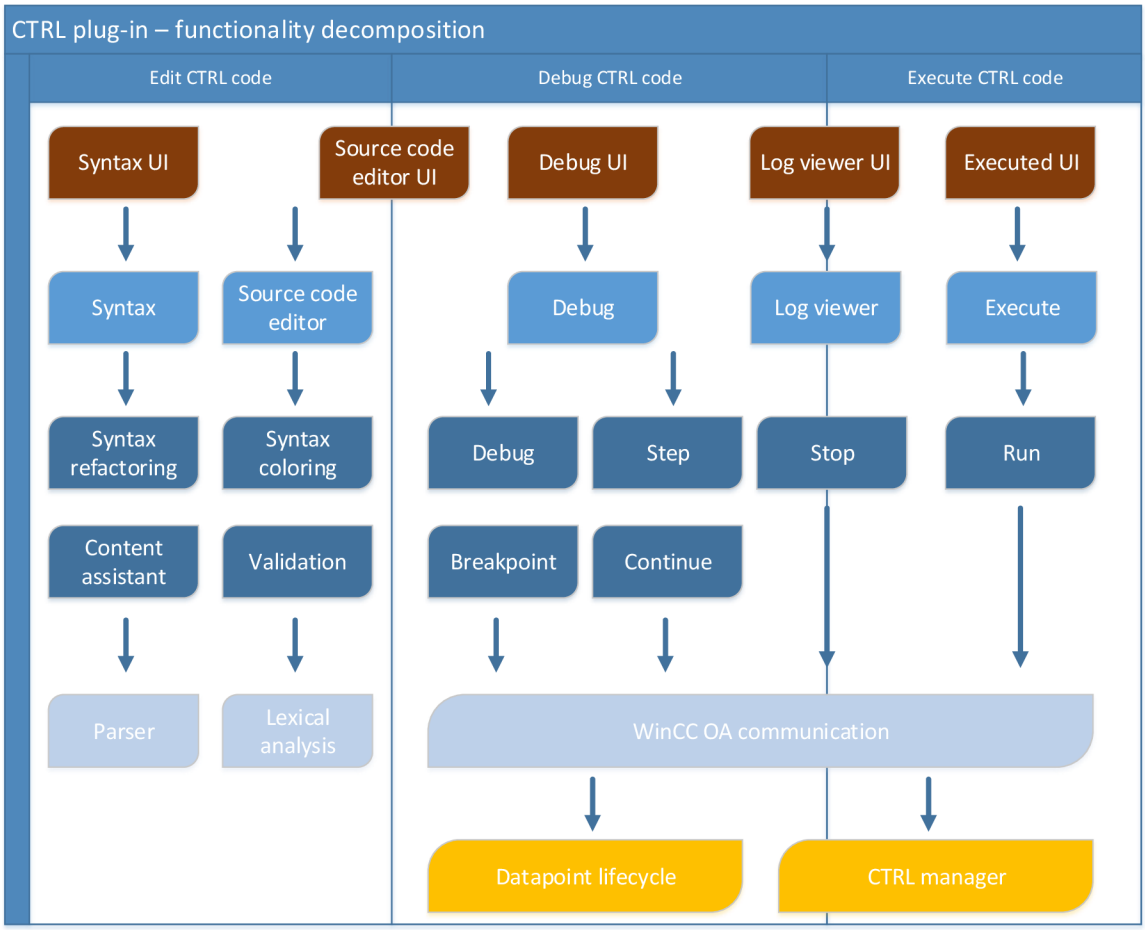

Figure 4.6: Original functional decomposition of CTRL plug-in, created by TUE.

The general functionality is already implemented by Eclipse itself or by pre-installed third-party plug-ins. As it comes to the WinCC OA-specific functionality, it is mostly implemented using the Xtext-based plug-ins. Each Xtext-based plug-in has its own implementation of lexer, parser, semantic analysis, code assists etc., and if implemented correctly and according to Eclipse and Xtext rules, it seamlessly integrates with the rest of the IDE.

The CTRL Debugger is a separate Eclipse plug-in, relatively independent of the CTRL plug-in. It provides functionality of CTRL code debugging and for this, it needs to communicate with WinCC OA. As a mean of communication, the plug-in uses Apache Thrift to send RPC messages to the WinCC OA Debug manager, the manager responsible for CTRL code debugging. It is important to know about the dependency of this plug-in to WinCC OA. Without a WinCC OA installation and without the special manager, the IDE cannot provide any of the CTRL debugging functionalities.

4.4 The IDE Software Stack

The Eclipse-related software has certain dependencies that must be taken into account when developing and distributing the IDE. Knowing these dependencies is also important not only for programming, but also for future planning. In order to plan the future development, developers and leaders of this project, should be able to keep track of the technologies

and components used in the IDE software stack. If a component is about to be deprecated, unsupported or the compatibility with future versions is going to break, the planning process should take such changes into account.

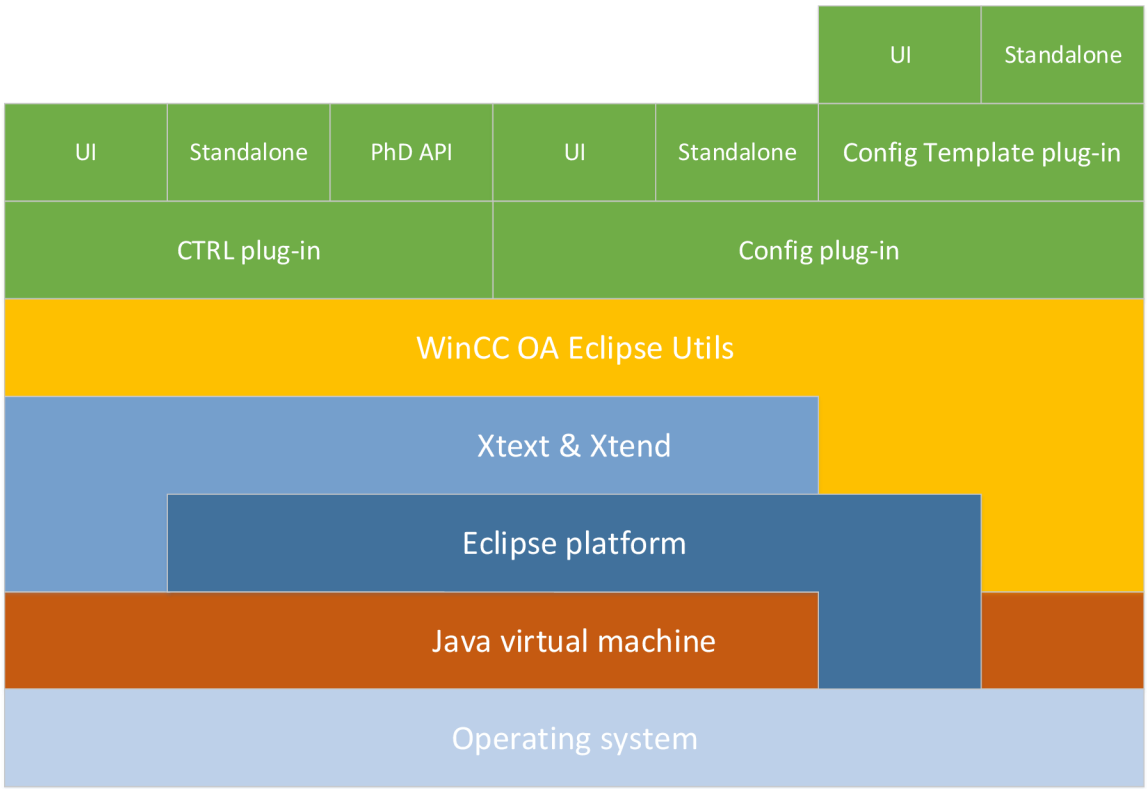

Figure 4.7: Software dependencies of the IDE. Software in upper layers depends on the software in lower layers.

In case of the IDE software stack (shown in Figure 4.7), most of the components depend on Java virtual machine. The used JVM has to be compatible with Java 8 and its main purpose is to provide runtime environment for the Eclipse Platform.

Apart from the Eclipse Platform being dependent on the JVM, there is also a hidden dependency on the underlying OS. This is because the GUI of the platform uses SWT, and SWT has to be implemented for every OS Eclipse should run on.

Till version 2.9, Xtext was purely dependent on Eclipse. Currently the dependency on Eclipse is not mandatory, since Xtext 2.9 is also available for IntelliJ IDEA.

On the level of the CERN plug-ins, there is a heavy dependency on Xtext. This dependency is here because most of the plug-ins use Xtext/ANTLR to generate a parser, and the Xtext framework itself is used for implementing DSL-specific features. Apart from the Xtext dependency, some of the plug-ins or their parts are integrated directly with the Eclipse Platform, and so changing the Eclipse Platform would most probably break the code.

Another type of dependency is between the CERN plug-ins themselves. First of all, one of the plug-ins is a utility, shared among most of the DSL plug-ins. This utility provides WinCC OA specific functionalities, classes responsible for the IDE logging[4], unified code

[4] All the CERN Eclipse plug-ins should do the logging in the same way.

analysis output, access to the Eclipse preferences and it also encapsulates some Xtext functionality[5].

Secondly, some DSL plug-ins are dependent on each other. One of the plug-ins is the Config Template plug-in. Since its engine has to interpret the templates, there is also a need for WinCC OA "config files" parsing. Without the model of a config file, the engine cannot perform the template matching, i.e. the Config Template plug-in depends on Config plug-in.

There are also some other software dependencies, but most of them are not critical. For instance, the CTRL Debugger plug-in depends on WinCC OA Debug manager, and the IDE-integrated XML $\longleftrightarrow$ PNL conversion also relies on installed WinCC OA. But even when these dependencies are not satisfied, the IDE will still work fine, only some functionalities may not be available to its users.

4.5 The IDE Plug-ins Design

This chapter describes components of the three main plug-ins: CTRL plug-in, Config plug-in and Config Template plug-in. Note that presented diagrams and their commentary cover only the most important components and dependencies of the plug-ins. Unimportant details were omitted, and curious readers can find them in the source code in the attachments.

It is also worth noticing the similarities between the diagrams (see Figure 4.8, Figure 4.9 and Figure 4.10). For example, all the plug-ins depend on `Utils` plug-in. This is because all the plug-ins use a unified way of logging and the class responsible for logging is in `Utils` project. Also every plug-in has its standalone (headless) version so it can be executed directly from command line, therefore no interaction with Eclipse GUI is required.

Another similarity among the diagrams is Xtext-related. By default, every Xtext plug-in consists of three parts:

- *Parser part* – contains a grammar, scope provider, code validator, etc.
- *UI part* – interacts directly with users using UI elements. It contains features like Outline View, Code Assist, Code Coloring etc.
- *Tests* – this part is used for automatic testing. We can have here tests for the grammar/parser, code validator, etc.

4.5.1 CTRL Plug-in Design

CTRL plug-in is the most comprehensive of all three plug-ins. The diagram in Figure 4.8 shows core components of the plug-in and also the most significant dependencies among the components.

Let us start with CTRL Parser, which is a component responsible for CTRL source code parsing. It consists of the grammar and Convertors. Convertors are Java classes that convert certain terminals into specific Java objects. These Convertors are then utilized by the generated parser and the objects they produce are used in the AST.

Next big component is responsible for static code analysis aka validation in Xtext terminology. The most important class here is `CtlValidator`. An instance of this class is used every time when Xtext wants to analyze the parsed AST. The validator class then

[5]The functionality of the utility project is much wider than described here, and it really covers most of the functionality that can be shared among CERN DSL plug-ins.

depends on various different classes, like helper classes responsible for AST manipulation or a class that provides an information about validation preferences (so the validator knows what analysis should be performed).

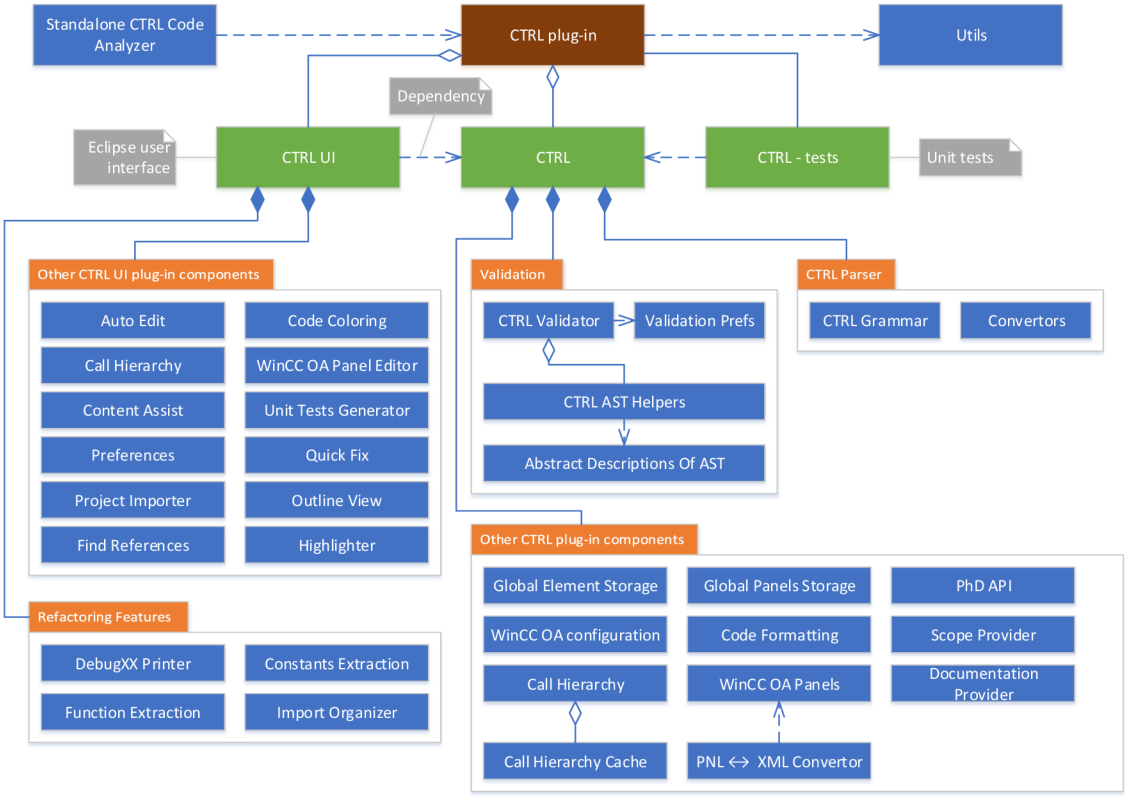

Figure 4.8: Component diagram of CTRL plug-in.

The components of the UI part are responsible for direct interaction with users. They usually represent IDE features accessible via some buttons, windows, panels or code editor. It is good to know that even though these components are related to Eclipse GUI, they usually do not require any knowledge of SWT. It is because Xtext framework is designed in such way that it separates visual part of the components and their behavior. Nevertheless, some CTRL plug-in features are beyond the scope of what Xtext is capable of and they were written using SWT and Eclipse API directly.

4.5.2 Config Plug-in Design

Config plug-in is focused on three main objectives. The first is validation of config file contents. The validation is mostly data-driven and it relies on a component that provides meta-information (for more information, see Chapter 5.4.2).

Second objective is the code navigation. Some config files were big and programmatically generated, and so users needed a way to easily navigate through config file contents. The means of navigation are Code Coloring and Outline View components. Coloring makes the code easy to read and Outline View makes easy to see the whole config file structure.

The last objective is to provide inline documentation for users. The components responsible for this are Content Assist and Hover. Content Assist provides documentation

when user is writing a config file contents. Hover component provides documentation when a user hovers mouse over certain parts of config file contents.

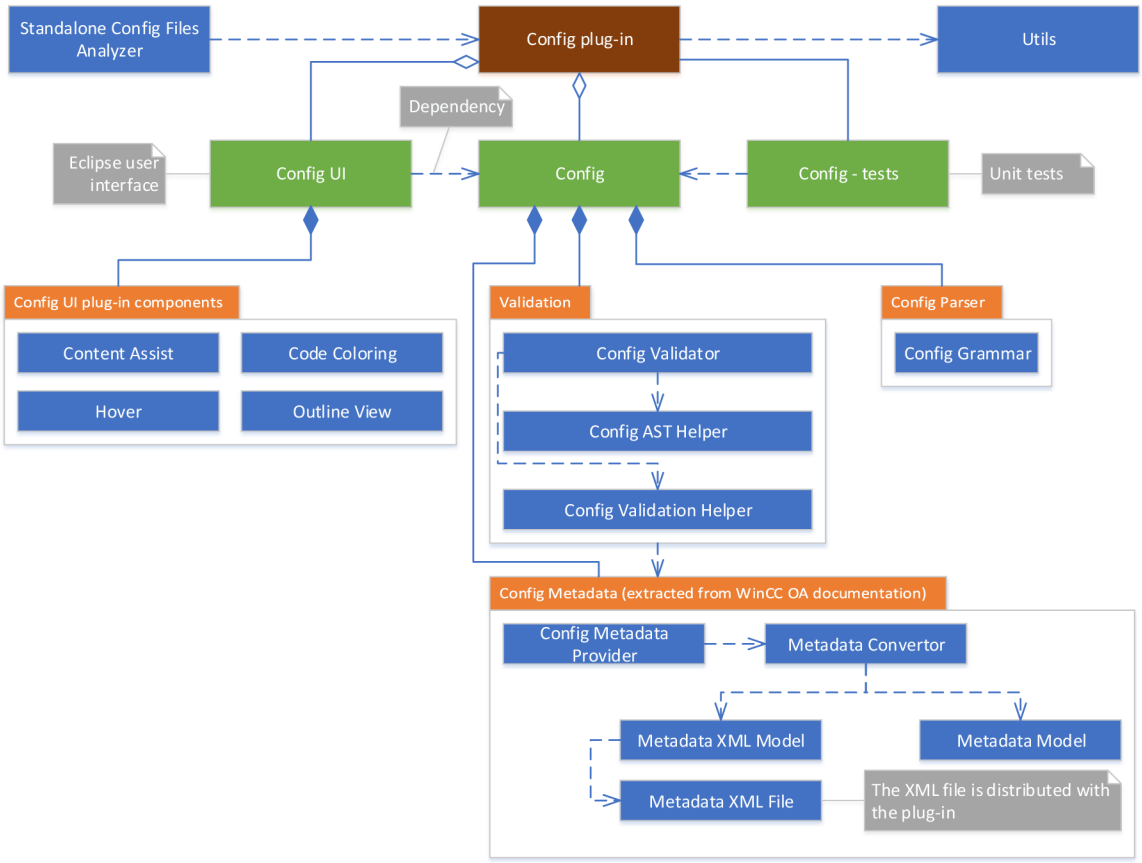

Figure 4.9: Component diagram of Config plug-in.

4.5.3 Config Template Plug-in Design

Config Template plug-in is primarily focused on config template execution. This is why most of the plug-in components (see Figure 4.10) are related to config template engine and Nashorn engine (JavaScript execution).

Apart from the engine itself, there are:

- **Model Convertors** – convert ASTs of Config Template code and config files into another, more appropriate representations (see Section 5.5.2.1).
- **Result Printer** – prints the results of the execution. It can generate JSON output so other programs can process it.
- **Code Context** – a small component (set of classes) that provides an information about the position/context of an error.
- **Engine Async Executor** – this component is able to run the engine in a background thread and it is used together with *Template Execution Window* (it increases of Eclipse GUI responsiveness).

A unique thing about this plug-in is that it is the only CERN-developed Xtext-based plug-in in the whole IDE that depends directly on another Xtext-based plug-in (Config plug-in). This means that the plug-in cannot be installed/used separately without Config plug-in. The dependency is here because the engine has to understand the contents of config files and Config plug-in is already able to parse it.

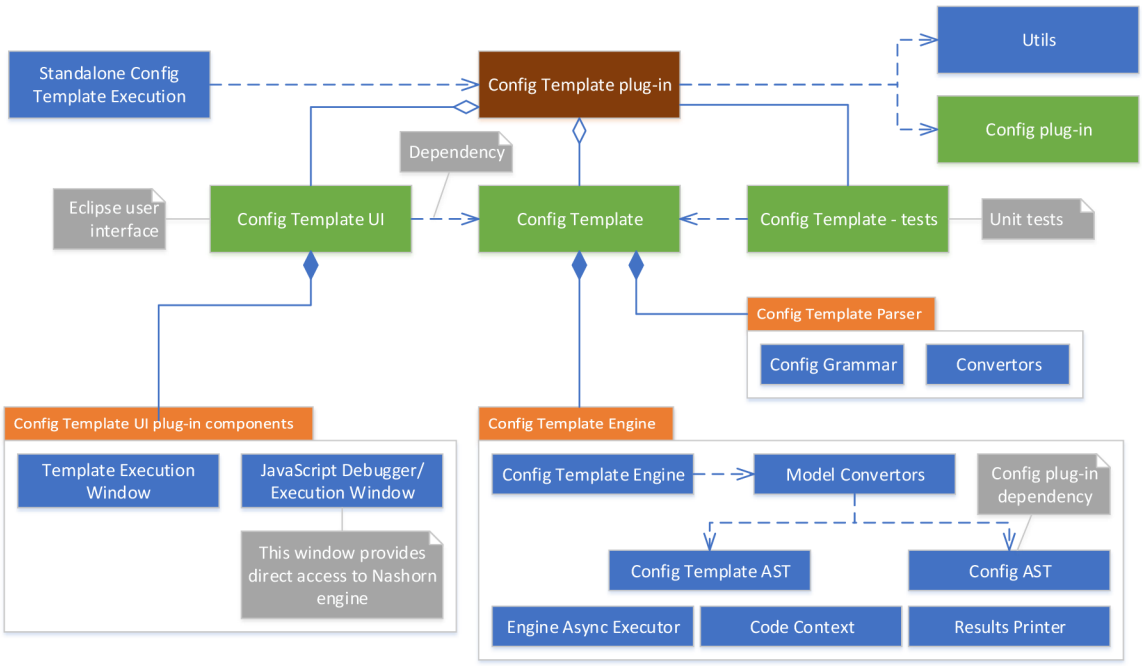

Figure 4.10: Component diagram of Config Template plug-in.

4.5.3.1 Config Template Engine

From the beginning, the language was intended to be a part of the IDE and it should serve as a verification tool of WinCC OA configuration files. For this reason, the engine/interpreter technology for the language, had to be multiplatform and easy to use, without installing any additional software apart the one shipped with the IDE.

Because of the requirements, Java and Xtext were selected as the base technologies (shipped with the IDE, multiplatform, relatively fast).

Another decision that needed to be made was: what will be the execution strategy? Should the engine interpret only the AST, should we translate the code to some simple intermediate form or directly to Java bytecode? At the end, the engine (see the schema in Figure 4.11) became a mixture of all three strategies, taking the best things from all of them.

- The basic layer of the engine was responsible for converting the AST into similar, but even more abstract and immutable[6] tree-like model. The engine then went through the model and interpreted individual nodes.

[6]The model was made immutable because my intention was to use the model safely among several threads and increase the speed of the engine.

- Since the language allowed programmers to “inject JavaScript” code[7], it made sense to use an already existing JavaScript engine. And since we already had the JavaScript engine and we needed the language to cooperate with the JavaScript part, it was only logical to convert some non-JS-parts of the language into JavaScript.
- The engine itself was shielded from the implementation of underlying technologies, but it is good to know, that the used JavaScript engine (Nashorn) compiled the code in memory and then directly passed the bytecode to the JVM[8]. The JVM could later interpret the bytecode or compile it to the native code and run it.

The details of the engine will be described later in this thesis in Section 5.5 and the source code is also available as part of the attachments.

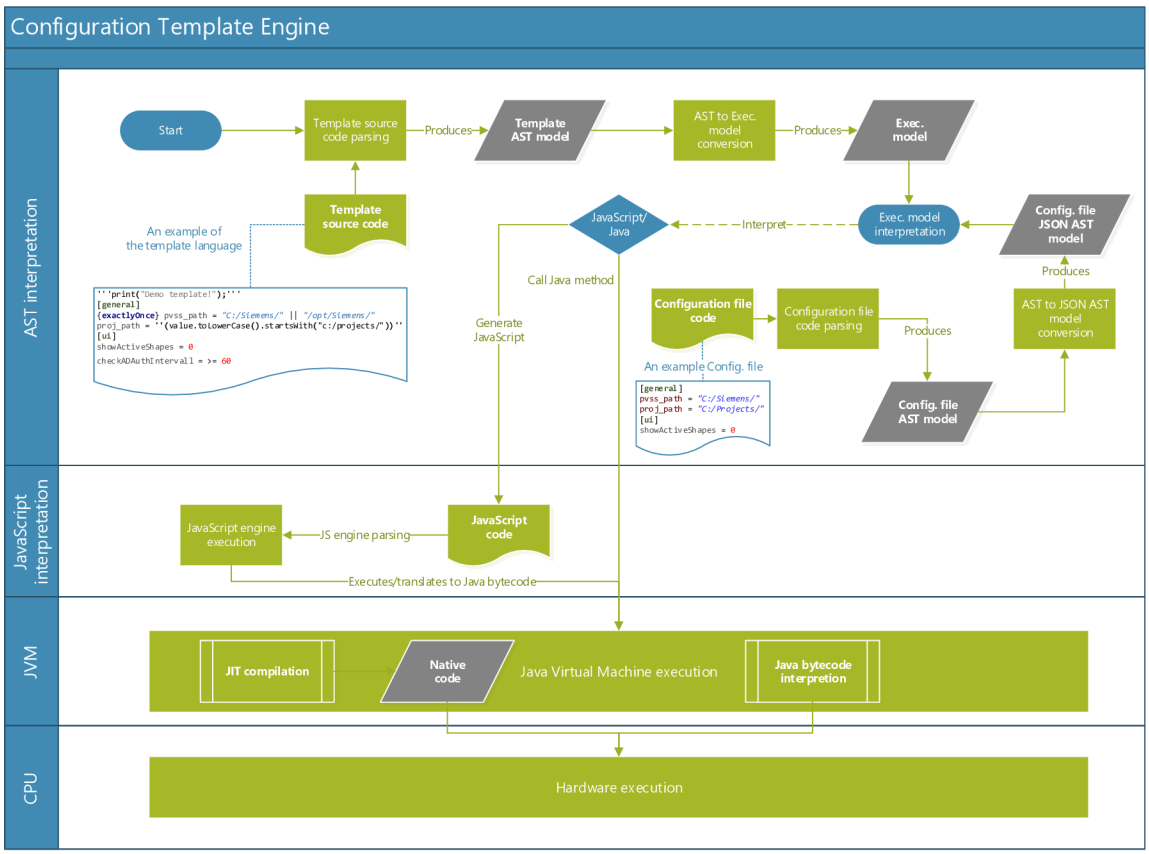

Figure 4.11: Schema of the engine created for Configuration Template Language.

At the end, it is also worth mentioning that the final design of the engine was not the only considered solution. I was also considering whether the engine should use some Xtext technology (apart from grammar definition and parsing). Technologies like Xbase Interpreter or code generation were contemplated, but eventually rejected because of insufficient documentation and flexibility compared to Nashorn and JavaScript.

[7] We could say that Config Template Language is a superset of JavaScript or additional layer built on top of JavaScript.

[8] Java 8: Compiling Lambda Expressions in The New Nashorn JS Engine. WEISS, Tal. Takipi [online]. Takipi, 2014, 2014-02-10 [cit. 2016-01-05]. Available at: http://blog.takipi.com/java-8-compiling-lambda-expressions-in-the-new-nashorn-js-engine/

Chapter 5

Implementation

The contents of this chapter contain topics related to the IDE implementation. Because the codebase of the IDE is extensive, it is not possible to describe every Java class nor every software component. So only the most interesting parts of the implementation are covered here. For more information about missing topics: language/parser testing, UI, SWT, code completion, code coloring, communication with WinCC OA debugger, etc., see the source code in the thesis attachments.

5.1 Used Programming Languages

The core of the IDE is built around the Eclipse Platform, and it is connected to other software components. Some of these components are not Java-based technologies so various programming languages and data formats had to be used during the development.

5.1.1 Java

Most of the code of the IDE is written in Java 8. Even though it would be possible to write Eclipse plug-ins in other JVM compatible languages, Java was chosen as the main language. There are several reasons why Java was the default choice.

The first reason is that Java is the most popular language for JVM and it is very popular language in general[18]. The assumption is that when more people know a programming language, it is easier to find a person who can continue the development.

The other reason is that Java is a relatively simple language. Compared to languages like C#, Java has less "syntactic sugar", it is easy to understand even by non-Java developers and students with basic knowledge of Java programming.

The last major reason is that Java language is developed by a big company (Oracle Corporation), and it is not likely that the Java development and its support will be terminated any time soon. This gives CERN a certain level of certainty in long term planning of the project.

5.1.1.1 Java Disadvantages

Although Java language was used a lot in the IDE and there are good reasons why it was chosen to be the main language, it was not the most suitable language as regards its features. There were three (missing) features of Java that turned out to be rather problematic.

- **Null references** – Java is well known for its *NullPointerException*. In order to make Java code not crashing because of `null` references, it sometimes requires a lot of `if` conditions and checks. This made the code messy, especially while working with Xtext generated ASTs. The code contained too many checks and it made most of the static code analysis algorithms hard to read. The way how they worked was hidden in a flood of null checks.

- **Immutability** – the support for immutability in Java is very limited. There is the `final` keyword that defines an entity, which can be assigned only once, but this is pretty much all that Java offers. Most of the collection classes like `List`, `Set` and `Map` are mutable and do not offer a real immutable version. Also creating own immutable classes requires a lot of boilerplate code, compared to *case classes* in Scala or *discriminated unions* in F#.

- **Pattern matching** – algorithms working with Xtext generated ASTs have to often dig deep into the structure of ASTs. In such cases, it is good to have means that can decompose the tree so developers can easily navigate through it. Unfortunately, Java does not have any convenient way of AST decomposition (`switch-case` statement has very limited capabilities) and so the most direct way is to use a lot of `if` statements with `instanceof` operator in conditions, followed by explicit casting. If Java had constructs like pattern matching or more powerful `switch` statements, it would make it more suitable for AST-related algorithms.

5.1.2 Xtend

Xtend is a statically-typed dialect of Java programming language. It integrates well with existing Java libraries and it compiles to human-readable Java code, not to Java bytecode. Depending on compiler options, it can be compiled to Java version 5 and newer.

Xtend is shipped together with Xtext plug-in and for a good reason. Compared to Java, Xtend provides advanced features useful for DSL development:

- **Property access** – if a Java class has `.getSomething()`, `.setSomething(argument)` or `.isSomething()` method, Xtend converts them into properties. This results in cleaner code when accessing parts of Xtext AST.

  - For example, instead of writing something like `astNode.getChild()` in Java, we can simply write `astNode.child` in Xtend.

- **Null-Safe navigation operator** (`?.`) – this feature is extremely helpful when digging deeper into the AST structure. Instead of writing many checks for null references, we can combine this feature with "property access" and use it to make null-check only once and in very compact form.

  - For example, in the CTRL plug-in, if we want to know, what is the name of a function that is being called, we would have to write 4 null reference checks in Java and it would make the code hard to read. On the other hand, it takes only one short line of code in Xtend to get the function name: `val funcName = fcall?.memberCallTarget?.^var?.name`

- **Elvis operator** (`?:`) – this operator is another null-safe feature. It evaluates expression on the right side if the original expression on the left side is `null`.

– An example: `val funcType = func.returnType ?: 'unknown'`

- **Switch expression** – there are big differences between Java `switch` statement and Xtend `switch` expression. The first difference is already contained in the name. `Switch` in Xtend is an expression, not a statement. It means that it can be assigned to a variable. The type of such variable is then inferred automatically. Another difference is that `switch` expression is not only limited to certain data types, but it can be used with any type of object. Individual cases contain expressions that are evaluated either to boolean value or to another object. If the value is boolean, then the `true/false` value determines whether the case continues. When the case evaluates expression to an object, then the `equals()` method is called to get the boolean value. All these features of `switch` expression make the AST decomposition very easy, but there is one more feature that makes it even better. The feature is called *type guards* and it allows programmers to decompose AST based on types of its nodes. Also whenever a variable passes a type guard, no explicit casting is needed anymore. It saves a lot of boilerplate code and makes the algorithms more readable. On top of that, Xtend comes with one more feature that makes type guard safe to use. It uses static code analysis and type inheritance to make sure that all type guarded cases are reachable and that programmer covered all cases or used default case at the end of `switch` expression[21].

5.1.2.1 Xtend Disadvantages

Although Xtend is a great language for Xtext-AST related algorithms, it has also some disadvantages.

The first disadvantage is the Xtend compiler. It is far from flawless and it is also relatively slow. At some point the compiler had a bug that caused a valid Xtend code to be compiled into invalid Java code. This bug was later fixed, but it showed insufficient level of Xtend testing. As regards the speed, the Xtend language implementation is written in Xtext and compared to JDT and Roslyn, it is slow. The slowness is especially noticeable when working with medium-size and big-size projects or with files that have several thousand lines of code.

From my perspective, the other disadvantage is the language syntax benevolence (for more information, see Section 5.6.2). The syntax allows programmers to write a lot of optional code. I think that the intention of the language designers was to make the language familiar to pure Java developers. But it just brought inconsistency and confusion.

For example, semicolons are optional, data types specification is optional and parentheses are also optional for methods without parameters. All these optional parts of the syntax make the code inconsistent, because Xtend beginners slowly learn the language and move from Java-like form to the lightweight form of Xtend. Also this syntax benevolence causes problems when more programmers are working on the same project and they are using different styles of Xtend.

5.1.3 C++

C++ was not used directly in the Eclipse based IDE. It was used only for building the WinCC OA debug manager. The amount of C++ code is not significant in the whole IDE project, but it should be mentioned since the manager is an important software component that provides a connection between the IDE and WinCC OA.

5.1.4 JavaScript

JavaScript was used in three scenarios in the IDE. All these scenarios took advantage of JavaScript dynamic nature and the possibility to execute it directly on JVM or in web browsers.

The first usage was in the Config Template plug-in. As already mentioned in Section 3.3.3.2, the Config Template Language itself allows programmers to inject JavaScript code and the Config Template engine uses partially Nashorn (JavaScript engine for JVM) to execute the template language code. In this case, most of the JavaScript code is generated at runtime by Java and then executed when needed.

The second usage is in Config plug-in. This plug-in is largely data-driven and the data for this plug-in contains snippets of JavaScript code. These snippets are loaded at plug-in startup as part of an XML file and then executed during Config file validation. Again, the execution of JavaScript code is done by Nashorn engine.

The last usage of JavaScript was more traditional as it was a part of a web page. The IDE code analysis tool is able to generate reports about analyzed code. One of the output formats is JSON. The webpage with JavaScript is able to load such JSON output and provide users an interactive code analysis report with features like searching, sorting, code preview etc.

5.1.5 XML & JSON

XML and JSON formats are used by the IDE for outputs. The reason, why these two formats were chosen, is that both are text formats and both are easy to use by other tools and libraries in common programming languages.

The XML format is used for WinCC OA panel files in XML form (there is also proprietary PNL format), for output of the code analysis and for Config plug-in validation metadata. The advantage of XML is that it can be formally described using XSD (XML Schema Definition) and tools like XJC can take such description and generate JAXB classes from it. These technologies provide Java programmers a comfortable way of loading and storing data in XML format.

The major disadvantage of XML format is that it is too verbose and sometimes hard to read for humans. That is on the other hand where JSON excels. It is a lightweight format, easy to read be people with technical background and it can be integrated with JavaScript.

5.2 Dependency Injection

Dependency injection is a design pattern that is used for reducing dependencies among components of some software system. In practice, it takes away responsibility from classes for creating/getting the objects they need to work properly.

In the IDE, dependency injection is used by Xtext. It provides means of replacing default implementation of Xtext based plug-ins by a different implementation.

For example, the CTRL plug-in has a class called `CtlRuntimeModule`. This class extends `AbstractCtlRuntimeModule` and it allows us to override the default implementation. In case of the CTRL plug-in, `CtlRuntimeModule` injects CTRL-specific implementation of a class that converts parsed terminal symbols into Java objects. The way how the implementation is changed is shown in Figure 5.1.

```
public class CtlRuntimeModule extends
ch.cern.winccoa.ctl.AbstractCtlRuntimeModule {
// ...

        @Override
        public Class<? extends IValueConverterService>
bindIValueConverterService() {
                return CtlValueConverterService.class;
        }
        public Class<? extends IEObjectDocumentationProvider>
bindIEObjectDocumentationProvider() {
            return CtlEObjectDocumentationProvider.class;
          }
// rest of the implementation
}
```

Figure 5.1: Dependency injection in the CTRL plug-in.

Since Java language does not provide any syntax for dependency injection, the creators of Xtext decided to use *Guice*, an open source framework created by Google. The source code and documentation of this framework can be found on GitHub:

- https://github.com/google/guice[1]

5.2.1 Dependency Injection – Developer's Notes

From my own experience, understanding properly the concept of dependency injection and its usage at Xtext can be hard for beginners. Nevertheless, it is crucial to understand at least the basics, since it is important part of Xtext plug-in development.

Besides being familiar with dependency injection, developers also have to know, what classes can be re-implemented and what is the default implementation.

As for the list of classes in Xtext, it can be found in the API Documentation[1]. My favorite procedure of finding the classes I need, is to go to the web page with the API Documentation, use a text search in web browser, and search for the term that best describes the functionality I want to change. For example, if I want to change the way how my Xtext based plug-in provides users the code documentation, I just search for classes or packages that contain the substring "doc" or "documentation" and then I see what the web browser highlighted. In this case, the web browser highlights packages `"org.eclipse.xtext.documentation"` and `"org.eclipse.xtext.documentation.impl"`.

The first package contains interfaces that can be used when we want to create classes with our own implementation (`IEObjectDocumentationProvider` interface). The second package contains the default implementations for the interfaces in the first package (`MultiLineCommentDocumentationProvider` is an implementation of `IEObjectDocumentationProvider`).

It may also happen that an interface has already several implementations provided by Xtext. The example this is the `IScopeProvider` interface. In this case, we can either use

[1] API Documentation for Xtext 2.8 can be found here: http://download.eclipse.org/modeling/tmf/xtext/-javadoc/2.8/

the dependency injection to use the Xtext implementation that meets our needs most, or we can use it to inject our own implementation.

One last thing that was extremely useful when I was replacing a default implementation with my own, was to know, how the default implementation works inside and how it is used by other classes at runtime. Of course, there is the API Documentation with brief descriptions of classes, interfaces and methods, but in many cases, such description is not enough and sometimes it is just missing.

For this reason, I was using the Xtext source code to see what is the implementation of the classes and I was also using the Eclipse (Java) debugger in order to explore how are certain classes used by other parts of Xtext.

5.3 CTRL Plug-in

CTRL plug-in was the first plug-in of the IDE I was working on. The original "proof of concept" version somehow worked. It had the grammar of CTRL language, and it also had some basic code analysis (`CtlValidator` class) and some other feature implemented (like Outline View, `CtlOutlineTreeProvider`).

Apart from lack of functionality and bugs, the biggest problem with this version was, that it was slow and it was not able to deal with real WinCC OA projects used at CERN. It was not really useful for WinCC OA development at CERN.

This chapter covers the most challenging parts of the plug-in development. Such as parsing, linking, scoping, code analysis, speed and memory optimizations. Note that since the implementation was not always straightforward, you will probably have to also see the source code in the attachments to understand all the details.

5.3.1 Parsing

Parsing is the heart of CTRL plug-in. If this part does not work properly, other components of the plug-in may suffer from it or do not have to work at all.

5.3.1.1 CTRL Xtext Grammar

The CTRL parser was generated from the grammar. It is important to realize, that generated parsers are usually slower than hand-written optimized parsers and so the grammar should contain only what is absolutely necessary for parsing a source code into an AST. This is especially crucial when our parser has to parse big files.

In case of WinCC OA projects at CERN, some of the CTRL source files had over 10 000 lines of code. Editing such files was not a pleasant task for SCADA developers since the response of the IDE was slow. And all the slowness started with the parser.

Making the CTRL Parser faster There were several tricks I did to the CTRL grammar, but the one with biggest impact on speed was removing all integer constants from the grammar.

Figure 5.2 shows a CTRL grammar sample responsible for integer constants parsing. The meaning of this grammar part is the following: integer constant is either integer/-long number defined by INT_CONSTANT and LONG_CONSTANT terminals, or it is a variable of one of the following names ("EOF", "R_OK" ...).

```
INTEGERconstant:
  {INTEGERconstant} (
    value=INT_CONSTANT
  | value=LONG_CONSTANT
  | T_LINENUM
  | T_SEARCH_PATH_LEN
  | 'EOF'
  | 'R_OK'
  | 'W_OK'
  | 'X_OK'
  | 'F_OK'
  | 'SEEK_SET'
//  ... other constants
```

Figure 5.2: Part of the CTRL grammar responsible for constants parsing.

The whole list of variables which have meaning of an integer constant was rather long. It contained almost 500 names. When the parser was parsing WinCC OA projects[2], it consumed more than 30% of its time only to parse these integer constants.

After removing the constants in form of variable names from the grammar, the 30%+ overhead was significantly reduced (see Figure 5.3 and Figure 5.4). The new overhead of integer constants parsing was less than 0.22%.

Hot Methods

Filter Column: Stack Trace

Stack Trace	Sample Count	Percentage
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleINTEGERconstant()	685	30.22%
org.antlr.runtime.CommonTokenStream.LT(int)	257	11.34%
org.antlr.runtime.CommonTokenStream.skipOffTokenChannels(int)	88	3.88%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleAssignmentExpression()	56	2.47%
org.eclipse.xtext.parser.antlr.XtextTokenStream.LT(int)	56	2.47%
org.antlr.runtime.BaseRecognizer.match(IntStream, int, BitSet)	55	2.43%
org.eclipse.xtext.nodemodel.impl.NodeModelBuilder.addChild(ICompositeNode, AbstractNode)	34	1.50%
org.eclipse.xtext.parser.antlr.XtextTokenStream.LA(int)	31	1.37%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.rulePrimaryExpression()	30	1.32%
org.eclipse.xtext.parser.antlr.AbstractInternalAntlrParser.afterParserOrEnumRuleCall()	30	1.32%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleSelectionStatement()	28	1.24%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleStatement()	23	1.01%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleUnaryExpression()	22	0.97%
ch.cern.winccoa.ctl.validation.CtlASTHelper.tryToExtractVarIbleOfGivenNameFromEObject(EObject, String)	22	0.97%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.rulePostfixExpression()	20	0.88%
java.util.HashMap$TreeNode.getTreeNode(int, Object)	19	0.84%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleConstant()	19	0.84%
org.eclipse.emf.ecore.impl.MinimalEObjectImpl.fieldIndex(int)	18	0.79%
org.antlr.runtime.BaseRecognizer.pushFollow(BitSet)	17	0.75%
org.eclipse.xtext.parser.DefaultEcoreElementFactory.create(EClassifier)	15	0.66%
org.eclipse.emf.ecore.impl.BasicEObjectImpl.eGet(EStructuralFeature, boolean, boolean)	13	0.57%
org.antlr.runtime.DFA.predict(IntStream)	12	0.53%
org.eclipse.emf.ecore.util.EContentsEList$FeatureIteratorImpl.hasNext()	11	0.49%
org.eclipse.emf.common.util.Pool$AccessUnit$Queue.pop(boolean)	11	0.49%
org.eclipse.emf.common.util.AbstractTreeIterator.next()	10	0.44%
org.eclipse.emf.common.util.Pool.access(boolean, Pool$AccessUnit)	10	0.44%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleSTRINGconstant()	10	0.44%
com.google.common.base.Predicates$InstanceOfPredicate.apply(Object)	10	0.44%
com.google.common.base.Objects.equal(Object, Object)	10	0.44%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleExpressionStatement()	9	0.40%
java.util.HashMap.putVal(int, Object, Object, boolean, boolean)	8	0.35%
java.util.HashMap.getNode(int, Object)	8	0.35%

Figure 5.3: Profiler results of CTRL plug-in without optimized parsing of integer constants.

[2]In order to make profiling results objective and consistent, the WinCC OA project for profiling was always the same and it contained around 200 000 lines of various CTRL code.

Hot Methods

Filter Column: Stack Trace

Stack Trace	Sample Count	Percentage
org.antlr.runtime.CommonTokenStream.LT(int)	1,275	27.69%
java.util.BitSet.get(int)	355	7.71%
org.eclipse.xtext.parser.antlr.XtextTokenStream.LT(int)	220	4.78%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleAssignmentExpression()	131	2.84%
org.eclipse.xtext.parser.antlr.XtextTokenStream.LA(int)	115	2.50%
org.antlr.runtime.BaseRecognizer.match(IntStream, int, BitSet)	112	2.43%
org.antlr.runtime.BaseRecognizer.pushFollow(BitSet)	106	2.30%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.rulePostfixExpression()	78	1.69%
org.antlr.runtime.DFA.predict(IntStream)	78	1.69%
org.eclipse.xtext.nodemodel.impl.NodeModelBuilder.addChild(ICompositeNode, AbstractNode)	75	1.63%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.rulePrimaryExpression()	51	1.11%
org.eclipse.emf.common.util.Pool.access(boolean, PoolSAccessUnit)	49	1.06%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleStatement()	49	1.06%
java.util.HashMap$TreeNode.getTreeNode(int, Object)	41	0.89%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleUnaryExpression()	38	0.83%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleExpression()	38	0.83%
java.lang.AbstractStringBuilder.<init> (int)	37	0.80%
org.eclipse.xtext.parser.antlr.AbstractInternalAntlrParser.afterParserOrEnumRuleCall()	35	0.76%
org.eclipse.xtext.parser.DefaultEcoreElementFactory.create(EClassifier)	34	0.74%
org.eclipse.emf.ecore.util.EContentsEList$FeatureIteratorImpl.hasNext()	32	0.69%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleSelectionStatement()	32	0.69%
org.eclipse.emf.ecore.impl.BasicEObjectImpl.eInternalResource()	28	0.61%
org.eclipse.emf.ecore.impl.MinimalEObjectImpl.fieldIndex(int)	23	0.50%
org.eclipse.emf.ecore.impl.BasicEObjectImpl.eGet(EStructuralFeature, boolean, boolean)	23	0.50%
org.eclipse.xtext.parser.DefaultEcoreElementFactory.set(EObject, String, Object, String, INode)	19	0.41%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleAssignmentOperator()	18	0.39%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.synpred117_InternalCtl_fragment()	18	0.39%
ch.cern.winccoa.ctl.validation.CtlASTHelper.tryToExtractVaribleOfGivenNameFromEObject(EObject, String)	18	0.39%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleConstant()	17	0.37%
ch.cern.winccoa.ctl.ui.syntaxcoloring.CtlSemanticHighlightingCalculator.getStyleForNode(INode)	17	0.37%
java.util.HashMap.getNode(int, Object)	15	0.33%
ch.cern.winccoa.ctl.parser.antlr.internal.InternalCtlParser.ruleSTRINGconstant()	15	0.33%

Figure 5.4: Profiler results of CTRL plug-in with optimized parsing of integer constants.

Speeding up parser is a good thing. But we still need it to work properly. In this case, the speed-up means that we sacrificed parser ability to recognize built-in constants. So we have to implement this plug-in feature somewhere else, and the implementation has to be faster than the one that was using the grammar.

The new way of implementing this feature in the plug-in was using a process called *linking*. This process creates virtual connections between used variables/functions and their declarations and it is later described in Section 5.3.2.

5.3.2 Linking and Scoping

The linking in Xtext is a feature that connects certain nodes in AST model. In other words, it is resolution of cross-references. From the perspective of CTRL plug-in users, linking combined with hyperlinking allows them to see, where is a certain variable or function declared and where are its usages.

From perspective of Xtext-based plug-in developer, linking of a variable/function means that an algorithm has to find a node, where is the variable/function declared. In case of CTRL language, the variable/function declaration can be local (in `for` statements, as a *function parameter*, etc.) i.e. in current source code file, or it can be imported from another file. In such case the search algorithm has to go through all the possible places where the variable/function declaration can be present and create a connection for it.

In Xtext, developers usually do not override the default linker implementation. Instead they are working with another abstraction layer the liker relies on. The abstraction layer is called scoping and it implements `IScopeProvider` interface. In reality, developers should not inherit the from `IScopeProvider` directly, but rather their implementation should inherit from `AbstractScopeProvider` class.

The `IScopeProvider` interface has only one method. This method takes a context for scoping, a reference for which to get the scope, and it returns an object whose class implements `IScope` interface. The scopes can be nested (there are many different implementations of this interface), and they return another abstracted objects, etc. As you can see, **things are getting more and more complicated and the main goal we wanted to achieve is hidden under several layers of abstraction**. This is a **common prob-**

lem in Xtext and that is why I recommend to rather debug the source code of Xtext than trying to understand the concept only from the provided documentation.

Shortly, an `IScope` object returns objects (nodes) that are visible from certain part of the AST (we call it *context*) and it also return names and qualified names for these objects[16, 45].

5.3.2.1 Interaction between Linker and Scope Provider

The simplified overview of interaction between liker and scope provider in Xtext can be summed in the following 5 steps:

1. The linker is about to resolve a cross-reference. It gets a context in form of an `EObject` (node of AST), an `EReference` object that contains information about the feature and the type, and it also gets an `INode` object that is used for getting the textual representation of the context (see Figure 5.5).

   - Let `con` = context in the AST (for example: `ch.cern. ... .Expression`).
   - Let `txt` = text in the context (variable or function name, let us say `arrLength`).
   - Let `fea` = feature of the context node (for example: `var` is a feature of context of type `Expression`).
   - Let `type` = type of the context feature (for example: `ch.cern.winccoa.ctl.Fv`).

2. The linker then calls scope provider in order to get the scope for elements that can be assigned to the feature `fea` (to `var` feature in AST node of `Expression` type) in the context `con`.

3. When the scope provider returns the scope, the linker searches the scope and it is trying to find an element whose name or qualified name matches the text `txt`. In our example, the scope should contain variable declarations accessible in the context and one of the declarations should have declared the variable of name `arrLength`.

   - Note that the Xtext documentation states that the scope returned from a scope provider should contain *"**all** target candidates for the given object and cross-reference"*[17].

4. If the linker finds the right object (i.e. variable of name `arrLength`), it creates a connection (link) between the used variable and its declaration (another node in the AST).

5. In case of failure (the declaration for the variable was not found), the plug-in displays users a message that it could not resolve the reference.

context	ExpressionImpl (id=474)
c	null
e	null
eContainer	RelationalExpressionImpl (id=1556)
eFlags	-262140
eStorage	Adapter[2] (id=1559)
var	FvImpl (id=510)
eContainer	null
eFlags	64
eStorage	URI$Fragment (id=513)
name	null
ref	EReferenceImpl (id=481)
node	LeafNode (id=523)

Figure 5.5: An example of an information given to linker in CTRL plug-in.

The simplified overview of linking and scoping makes probably, to most people, more sense than a lot of interfaces and abstract classes. But the question that many developers may ask is: *"Why should we create a scope that contains all possible candidates for linking in the given context? Isn't it waste of memory and computing power? Doesn't it make things unnecessary complicated?"*.

The answer to these questions is *"it depends"*. Computing the scope with all possible candidates for linking may be, in some cases, easier than implementing own linker. Especially when we are implementing some relatively simple DSL. The creation of scope can be in such cases just a few lines of code. Additionally, if the amount of parsed source code is small, we do not have to worry about lack of memory or performance.

On the other hand, if we are using Xtext to build a plug-in for some general-purpose or somehow complicated language, we may reach the memory and performance limits relatively soon. In case of CTRL language, some source code files have over 10 000 lines of code. The code itself contains thousands of variables and function calls. The scope for each of these variables and function calls can contain many thousands of items, because the source code can import various libraries and in every context, hundreds of built-in variables and functions are accessible.

5.3.3 Linking and Scoping in CTRL Plug-in

The linking and scoping in CTRL plug-in is done in such way that it primary focuses on performance and memory usage. The implementation is not as clear as it could be, but it was the first time I had to implement linking/scoping and I had to make it as fast as possible.

As an Xtext beginner, I was not sure whether I should change the implementation of the linker or the scope provider. All the books, tutorials and documentation[3] I have read, lead me to the `IScopeProvider` interface. Of course I asked myself the questions about performance and necessity to return all assignable nodes in the scope. I also asked the questions: *"What will happen if I return only one node in the scope? Only the declaration*

[3]In 2014, the Xtext documentation was not really good and sometimes I felt confused by it. This lead me to make certain decisions, I do not consider as best from current perspective. Fortunately, the documentation improved a lot since then, and now it is much cleaner for Xtext beginners.

of variable/function. Would the linker still work? The linker takes only one node from the scope anyway. Would it make the whole process faster?".

The answer is: *"Yes, the linker still works with only one node and it is faster."*. Basically what I did was that I have implemented a linker on the place where the scope provider should have been implemented, and I returned a scope to the default linker implementation with only one node to choose from.

The solution I have created really worked. And it worked well. It was much faster and reliable than the previous one. But instead of naming it `CtlScopeProvider`, it should have been name `CtlLinkingService` and it should have been placed in a little bit different position in the plug-in architecture.

5.3.3.1 The Scope Search Algorithm

The algorithm used in the CTRL scope provider is trying to find the declaration of a variable or a function call.

It first searches in the local file; the AST in which is the given context. It starts from the context node (`EObject`) and it goes up in the AST hierarchy. It has to go through all the places where the variable/function for the given context can be declared and when it matches the variable/function name with the right declaration, it returns a scope with such declaration.

If the algorithm for local scope was not able to find the appropriate declaration, the search continues in the libraries imported by `#using` statements. Apart from the information, what libraries are imported, the algorithm needs also the information about the location of the libraries. This information is in the *config file*, which should be part of every WinCC OA project.

Assuming that we have the a list of imported libraries and we know, where to find them, we can continue the search. The libraries are searched for all the public variable and function declarations and again, the algorithm is trying to find the match.

If the algorithm fails to find the declaration in libraries, there is still a possibility that the used variable/function is *built-in*. It can be, for example, one of the constants mentioned already in Section 5.3.1.1 or a built-in function described in the WinCC OA documentation.

Unfortunately, CTRL is a dynamic programming language. The implication of this language feature means that a function can be loaded at runtime and then used by some CTRL code. This is a big problem in context of static code analysis and linking. There is simply no direct information available about whether the called function was declared, what arguments it accepts and what is the data type it returns.

To solve this linking problem, the scope provider examines the program flow and tries to find programming constructs that ensure that the used function is available in current context. The algorithm responsible for this analysis and linking is described in Section 5.3.3.3.

5.3.3.2 Linking of Built-in Variables and Functions

The CTRL language has some specific aspects that make linking and code analysis challenging.

The first aspect is that CTRL programmers can call built-in functions to which the CTRL plug-in has no access. These functions are described only in the WinCC OA documentation. What makes them even more specific is that the built-in functions are often overloaded, and overloading itself is not supported by CTRL language.

To deal with such functions, a class called `BuiltInFunctionsAndVariables` was created. This singleton class provides fast access to the built-in functions and variables. Instead of returning parts of AST with variable/function definitions, this class returns instances of `BasicVariableDescription` and `BasicFunctionDescription`. These classes contain all the information needed during linking and code analysis. For example, objects of `BasicFunctionDescription` class store information about function name itself, return data type, optional documentation and many more. For us is interesting that the class also contains parameters in form of `List<BasicFunctionParamsDescription>`. This representation of parameters can describe all kinds of overloading and programmers can even use it to express that the parameters should have some pattern and can repeat multiple times.

So at the point when the scope provider cannot find any suitable local or library definition, it uses the singleton instance of `BuiltInFunctionsAndVariables` class to get the necessary information for scoping and linking of built-in functions and variables. But even though this algorithm seems to be relatively straightforward, it is a little bit more complicated. The issue here is that only parts of AST(s) can be linked together, i.e. `BasicVariableDescription` or `BasicFunctionDescription` are not derived from `EObject` class and therefore they cannot be used for linking directly. For this reason, the classes for description contain a field that references to an `EObject`. If the `EObject` is missing the scoping algorithm can convert descriptions of variables and functions into fake AST nodes and bind this fake model to a resource.

5.3.3.3 Linking of Undeclared Functions

Another aspect of CTRL language is that it is a dynamic language and functions can be loaded at runtime. It is not an easy task to foresee and handle this situation during linking and scoping. The CTRL scope provider is however able to link some of such undeclared functions.

The way how this special kind of linking is done is the following: first the algorithm ensures that the called function will be available at runtime, secondly a fake AST for the function definition is created and linked to the original model.

In order to ensure that a function is available at runtime, the algorithm analyses the flow of the AST/source code that is being linked, and it searches for specific conditional statements (`if` statements).

If some of the `if` statements have a condition that ensures the existence of the linked function, we can safely create a fake AST for the function definition and perform the linking.

As for the `if` condition itself. The algorithm that performs the analysis relies on CTRL built-in function called `isFunctionDefined(funcName:string)`. According to the WinCC OA documentation, this function returns true if the function of given name exists. Otherwise it returns false.

The sample code in Figure 5.6 shows, how this linking functionality works in the IDE.

```
void main (){
	foo(true); // the IDE complains that the function 'foo' was not
found in the scope

	if(isFunctionDefined("foo")){
		foo(1, 2, 3); // function 'foo' is properly linked
	} else {
		foo(1); // the IDE complains that the function 'foo' was
not found in the scope
	}

	if(isFunctionDefined("foo")){
		bar(1, 2); // the IDE complains that the function 'bar'
was not found in the scope
	}
}
```

Figure 5.6: Demonstration of linker's ability to link functions that are accessible only at runtime.

It might be worth mentioning that this kind of linking provides the subsequent static code analysis only the information that the called function will exist at runtime. Therefore, the analysis cannot simply know what are the allowed arguments for these functions and what data type these functions return. Because of these limitations, it was decided that the code analysis will not be checking the arguments of such functions, and their return type will be always `anytype`. The `anytype` type is compatible with all other types and it does not cause false warnings during the analysis.

5.3.4 Performance Optimizations

The crucial part of the IDE was performance. From user's perspective, the IDE had to provide code analysis, context assist and many other features in real-time. Any significant delay was negatively judged by users, because it slowed down their work and caused frustration.

When I was developing the plug-in, it soon became clear that many components need the information about the source code in loaded WinCC OA projects. Components like scope provider, code analyzer and call hierarchy feature, needed fast access to information about the WinCC OA project libraries, their public functions, variables, data types, etc. So it was logical to create a central place, which would provide such data.

This place became a singleton class called `GlobalElementsStorage`. This class was designed in such way that it provides as fast access to its data as possible. It also does not not waste memory and it is thread-safe. But the access speed is still the most significant feature of this class.

To make this fast and thread-safe, classes like `ConcurrentHashMap` and variants of concurrent `Sets` and `Lists` were heavily used. In order to lower the memory footprint, these collections contained usually only metadata about parsed files. There was also one collection that was storing whole ASTs, but it was more of a temporary information and the idea was to remove it as soon as possible (this cache of ASTs was introduced as a speed-up for Call Hierarchy feature and to remember what files were loaded in Xtext resources).

As already mentioned, the class can handle read and write operations among multiple threads. This is very useful during scoping and code assist. In order to speed the parsing up, I have used the advantage of modern CPUs and made the parsing parallel. Obviously it was not possible to parallelize the Xtext generated parser, but it was possible to parse each file in a separate thread. So when the linker was searching for a variable or function declaration in libraries, I was able to load all the libraries with probable occurrence of the declaration in separate threads.

Similarly, when an IDE user requests the code assist for the first time, a background thread is started and it makes sure to parse all needed CTRL files in parallel. These files are later analyzed and their public variables and functions are converted into a metadata, stored in the instance of `GlobalElementsStorage` and provided back to the context assist.

Another optimization is that the class is keeping records of what files have been parsed already and what files are being parsed. So if multiple plug-in components running in separate threads request information about certain CTRL files or libraries, the class responsible for providing this information knows that some files do not have to be parsed again and some may not have been parsed yet, but they are already in the process of parsing.

5.3.4.1 Cache Invalidation

To speed up the plug-in, classes like `GlobalElementsStorage` and were `CallHierarchyCache` created. These classes worked basically like caches. However, the problem with caches is that they have to be up to date. When a CTRL file is changed and the cache contains an old information, it may lead to an incorrect behavior of the plug-in.

To keep track of resource changes, Eclipse allows developers to register an `IResourceChangeListener`. This registered listener then gets notified via an `IResourceChangeEvent` class instance[19].

Even though the resource tracking provided by Eclipse is nice and relatively easy to use, the CTRL plug-in uses a different approach. Instead of registering an `IResourceChangeListener`, it takes as a source of CTRL files changes the `CtlValidator` class. The validator class is called automatically every time a user edits a CTRL file in the IDE.

The advantage is that this class has access to already linked AST and when it notifies its listeners, it can provide them the AST model. Therefore, there is no additional overhead in form of parsing, linking and scoping when the listeners want to know what changed in the source code/AST.

The disadvantage is that such type of resource tracking cannot detect changes made by external programs. For example, if a user edits CTRL library file in Notepad++, the `CtlValidator` class is not called and therefore components that rely on it will not get the notification that something has changed. Luckily it happens rarely, so this type of notification and cache invalidation worked well.

5.3.5 Memory Optimizations

Speed optimizations do not always go hand in hand with memory optimizations. Usually we trade speed for memory and vice versa. However, the case of CTRL plug-in optimizations was different. The difference was that even though we wanted to achieve as much speed as possible and the caches created for this purpose consumed some memory, we could still make some drastic memory optimizations.

It may be surprising that the source of memory optimization were the caches. Concretely, the `GlobalElementsStorage` class allowed me to save a lot of memory. The reason

is that this cache does not contain whole parsed ASTs, but rather meta information about them. Often the plug-in components need to know only public variables and functions available from loaded/parsed CTRL libraries. The rest of the information is irrelevant.

Because the whole ASTs are needed rarely, it makes sense to remove them from the memory. The plug-in has a dedicated thread, whose responsibility is to find unused ASTs and remove them (unload the resources). If the IDE users do not work with the IDE for a while, the cleaning thread terminates itself to frees additional 1-2 MB of memory.

In order to make the IDE work properly, it was crucial that the thread responsible for resources unloading, did not unload any used resources. EMF models (i.e. ASTs) are very sensitive to bad resource manipulation. If a source code/AST is open in the CTRL editor and it is somehow linked to another EMF model, which is not attached to any resource, the editor displays an ugly error similar to:

> *The feature 'var' of 'ch.cern.winccoa.ctl.ctl.impl.ExpressionImpl@37fdfe80 {platform:/resource/WinCCOACourse131007/scripts/demo.ctl #//@ed.2/@cs/@sb/@sod.1/@item.0/@i/@i/@memberCallTarget}' contains a dangling reference 'ch.cern.winccoa.ctl.ctl.impl.FunctionImpl@329c62a3 {#//@ed.252/@f}'.*

The result of keeping only metadata about the source code and ASTs that are really needed, resulted a big in memory reduction. Originally when the IDE loaded a huge project (JCOP Framework), it consumed around 1.7 GB of memory. When the optimizations were implemented, the used memory was only around 200 MB (see Figure 5.7).

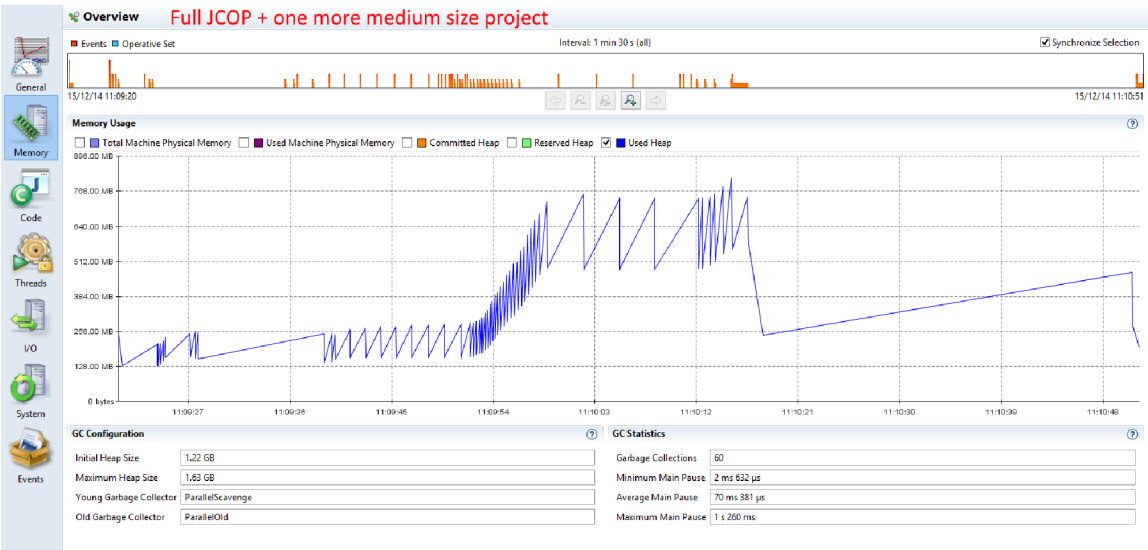

Figure 5.7: Results of profiling of the IDE with implemented memory optimizations.

5.3.5.1 Xtext Resources

The Xtext framework works with EMF resources. EMF resources are models of source code (i.e. ASTs). These resources can be stored in a resource set. If a resource is needed, the IDE can search the set and get the EMF model that was already parsed. The problem here is that a resource set may be big and it can consume a lot of memory. For example, if a WinCC OA project is open in the IDE and the build action is triggered, the resource

set contains resources/ASTs of all CTRL files. Depending on the project size, the memory occupied by created resource set can be significant and it can degrade the speed of the IDE[4].

5.3.5.2 Heap Limits for JVM

The Eclipse Platform and loaded plug-ins are executed by JVM. When using JVM to launch a Java application, several arguments can be passed to the JVM instance to adjust its behavior and execution. One of the arguments is `-Xmx<size>[g|G|m|M|k|K]`. This argument tells JVM to set the maximum heap size. The heap in this context is the part of the computer memory where Java objects are stored. This memory is also freed by the garbage collector[13].

The purpose of garbage collector is to find objects, that are not needed anymore and free the memory so it can be reused. The more objects a Java program creates and the more memory it consumes, the harder the garbage collector has to work. There are several types of garbage collectors in the JVM. Each of them with a little bit different properties and suitable for different kinds of applications. But there is one thing that is shared among all of them. When a garbage collector is working, the Java application is stopped. This is called as *stop-the-world*. Such behavior effectively means that when a Java application causes the garbage collector to do its job too often, the performance of the application degrades.

The knowledge of the JVM options and the behavior of the garbage collector is important, because the Eclipse installation folder contains a file named *eclipse.ini*. This file is responsible, among others, for passing arguments to JVM and one of the arguments is already mentioned `-Xmx`. The default value of this arguments depends on Eclipse version (usually it is 512-1024 MB) and it may have a big impact on Eclipse performance.

Before the memory optimizations mentioned in Section 5.3.5 were implemented, the Eclipse instance was able to consume over 1.5 GB of RAM when certain kinds of WinCC OA projects were open. This amount of consumed memory combined with limited heap caused the garbage collector to work so often and so long that according to the profiler records, it consumed over 90% of JVM computing time and it caused the IDE to be extremely slow. Basically all the CPU performance was consumed by the garbage collector.

The performance issue was solved by increasing the maximum heap size and later on by the memory optimizations. The lesson here is: if an Eclipse instance is slow for no obvious reasons (the implemented algorithms should theoretically perform well), it might be worth using a profiler to see the behavior of the garbage collector and increase the maximum memory heap if needed.

5.3.6 CTRL Static Code Analysis

Static code (program) analysis is a type of analysis that is performed without running the analyzed code. In context of Xtext, it is also called *static analysis* or *validation*.

The CTRL code analysis of was the most comprehensive part of the CTRL plug-in. Its purpose was to guide developers to write CTRL code according the CERN-defined rules and reduce the amount of bugs in the code.

[4]Personal note: It is surprising that concept and purpose of resources, resource sets and resource set provider is not better explained in the Xtext documentation, since it is such important part of Xtext and it influences not only performance and memory consumption, but also some features provided by Xtext.

The implementation of the code analysis is related to `CtlValidator` class. This class extends `AbstractDeclarativeValidator` and it uses annotations to specify methods that perform checks for certain parts of AST/EMF model. Figure 5.8 shows the code that is responsible for checking whether local variables in function definition are being used.

```
@Check
def void checkUnusedLocalVariables(FunctionDefinition fd) {
   val check = getCheckInfo(UNUSED_VARIABLE_DECLARATION);
      if(check.activated == false){
         return;
      }
   // get all Declarations of all variables used in expressions
   val usedVariableDeclarations = fd.eAllContents
      .filter[
         switch(it){
            Expression:{
               return it.^var instanceof Variable && it.^var.eContainer instanceof
DeclaredVariable;
            }
            SynchronizedExpression:{
               return it.name instanceof Variable && it.name.eContainer instanceof
DeclaredVariable
            }
            default:{
               return false;
            }
         }]
      .map[
         switch(it){
            Expression:{
               return it.^var.eContainer as DeclaredVariable;
            }
            SynchronizedExpression:{
               return it.name.eContainer as DeclaredVariable;
            }
         }
      ].toSet;

   val variableDeclarations = fd.eAllContents
      .filter[it instanceof DeclaredVariable] as Iterator<? extends DeclaredVariable>;

   variableDeclarations
      // filter only declared variables that are NOT used in function body
      .filter[usedVariableDeclarations.contains(it) == false]
      .forEach[
         raiseIssue(check.severity, "variable '"+it?.v?.name+"' is unused", it,
CtlPackage.Literals.DECLARED_VARIABLE__V, check.ID);
      ]

}
```

Figure 5.8: An Xtend code responsible for finding unused local variables inside a function definition.

As you can see in Figure 5.8 the code analysis is mostly about going through the AST and doing something with its nodes. In this case, the annotated method has one parameter of `FunctionDefinition` type and it returns `void`. Such method signature tells Xtext: *"I have the @Check annotation so when the static analysis is being performed, call me. And since I expect as an argument an object of FunctionDefinition type, call me every time you find such object in the analyzed AST."*.

As for the algorithm itself, it detects all unused local variables declared using `Declared-`

`Variable` grammar rule. At the very beginning, the algorithm gets an object with an information about the check. The object contains an information whether the check should be performed, what is its ID, severity and some additional data if need. This allows users of the plug-in to specify how strict should be the static code analysis and what kind of things it should check and how. This topic is more discussed in Section 5.3.6.8.

When it is decided that the check should be performed, the algorithm continues. It traverses all nodes/sub-ASTs in the function definition and it searches for Expressions that use some variables. Since the variables should be already linked to their declarations (node of `DeclaredVariable` type), we gather those declarations and it gives us a set of variables that are used.

Then we go through the function definition again and we search for all declared variables. At the end we have two sets. One set contains variable declarations that are used in the function definition and the second set contains all variable declaration (i.e. second set is the superset of the first one). At the end we use these two sets to compute what variable declarations are not used in the function definition and for them, we display an error message.

This particular example of a check demonstrates the general idea behind most of the other checks (analysis). We take the whole AST or its subtree, go through the model and gather some information about the source code. Then we analyze this information and print a message if needed. Also since AST is a recursive data structure, many of the algorithms used for code analysis are recursive or they use a special iterator that can traverse the tree.

5.3.6.1 Basic Grammar-related Checks

The first type of code analysis is related to the language grammar itself. As mentioned in the previous chapters, it is not a good idea to create full-blown Xtext grammars. It makes generated parsers slower. Such grammars are also harder to develop, read, maintain and generated EMF models may be too complex to work with. Additionally, the error messages generated by a hand-written code analysis make more sense for users of the IDE. For this reasons, the CTRL language grammar describes a superset of the actual language and the subsequent code analysis provides additional restrictions related to the syntax.

So the basic checks make sure, that the code is written according the language syntax specification. Here is a list of some of the basic checks:

- **Jump statements** – jump statements contain `return`, `break` and `continue` commands. All three are part of `JumpStatement` rule (it extends `Statement` rule) in the grammar. Since `Statement` rules can be used almost everywhere in the function body, the plug-in has to ensure that they are used only in places where it makes sense. For example, an error message is displayed to users when a `continue` statement is not inside some kind of loop.

- **Labeled statements** – similarly to the jump statements, labeled statements `case` and `default` are a subtype of `Statement` rule so they can be in all the places as the rest of statements. Again, a restriction was made and this time it checks whether labeled statements are directly inside a `switch` statement.

- **Synchronized variable** – variable definitions cannot be `synchronized`. Only function definitions can be declared as synchronized. The reason for this check is that almost all the access modifiers are the same for variable declarations and function

declarations so the grammar uses the same grammar rule for both and it is the job of the code analysis to tell users that variable definitions cannot be synchronized.

- **Global function** – this is analogous to the case of synchronized variables. However, in this case, it the check controls that a function definition is not declared as global.

5.3.6.2 Function Call Checks

One of the checks that was important for CTRL programmers, was related to function calls. When a function was called with wrong arguments, it could crash during runtime or cause an unexpected behavior of the program.

For this reason, the static code analysis checked the number of arguments passed to the function, the types of the arguments and whether they serve only as an input for the function, or they are used also for getting outputs from the function (in/out arguments).

Besides arguments, it also checked what is the return type of the function and whether and how is the returned value used in the source code. For example, if a function returned some value and the value was ignored, the analysis displayed an *info* message informing users that they should probably do something with the returned value.

As for the arguments checking, the algorithm began with conversion of the function definition into a more abstract form (see `BasicFunctionDescription` in Section 5.3.3). This abstract form was easier to work with and it was also suitable not only for function descriptions but also for method descriptions (see CTRL graphical objects in Section B.2.1). So one algorithm was able to check function calls and method calls.

The algorithm first compared the number of arguments with the number of parameters. If a function or method required some arguments, but it was called without any, an issue was raised. Else if the function/method call had any arguments, the algorithm continued. It compared all the arguments with an abstract description of declared parameters. An issue was raised when the types of arguments did not match and implicit conversion was not possible, or more arguments were required. In such case, the IDE displayed all the information about the required arguments.

The last part of the algorithm checked whether there are some superfluous arguments and for each one of them displayed a warning.

Besides the logic described here, the algorithm calls methods of so called *"helper classes"*. There are several types of such classes and their purpose is to encapsulate AST operations that are needed at multiple places, or just abstract the logic of certain algorithms. They also make the IDE more resistant against potential bugs caused by grammar changes because they abstracted the AST, so not all algorithms depended directly on the structure of the generated EMF model.

5.3.6.3 Expression Checks

One group of checks was related to expressions. This checks usually controlled whether operators and their operands are compatible. As a demonstration, see the example in Figure 5.9.

```
void foo (file myFile){
        DebugTN(myFile+false); // the IDE is complaining: Additive
operation '+' on types: 'file' and 'bool'.
}
```

Figure 5.9: A demonstration of an expression check in the IDE.

This example shows that using + (plus) operator on a variable (expression) of type `file` and `boolean` value (also expression) is not allowed. The result of this check is that the IDE shows users a message, informing them that there is a potential problem.

I should also note that some of the checks are stricter than it is required by the language specification. Basically the CTRL interpreter is able to do all kinds of implicit conversions (casting). I did not test it all, but from what I have seen, it looks like the interpreter is able to convert almost every data type into some other data type and all these conversions are without crashing the execution. This may lead to an unexpected program behavior. And that is why the IDE allows only implicit conversions that make sense and do not lose precision.

For example, if a programmer is trying to assign `long` or `float` value/variable to a variable of type `int`, the IDE will raise a warning. The way how to avoid such warning is to use an explicit casting, where programmers basically say: *"I am aware that I may lose precision or get some strange result, but in this particular case, I want to do the casting anyway."*.

From the implementation point of view, there is an interesting *helper method* that is heavily used not only by this check. This method takes as an argument an AST node of type `Expression` and returns an object of `Optional<BasicTypeName>` type. In other words, this method computes the data type of the given expression, regardless of its structure, i.e. the expression can be composed of many other expressions and function calls.

Opposite to this method, there is also a method that computes the expected data type of an expression. It is very useful especially for the quick fix functionality, when the IDE can help users with correct explicit casting. The way how the expected data type is computed is reversed to the previous method that computes a data type of an expression. Instead of using recursion and digging deeper into the AST, this algorithm goes up in the tree structure and it is looking for certain positions in the AST.

For example, if a parent of a given node is a return statement, we determine the expected data type by looking at the return data type of the function definition the return statements is in. Another example is that if the given node is an argument of a function call, the algorithm finds the function signature and returns the data type of relevant parameter.

5.3.6.4 Unreachable Code Detection

Unreachable code is part of the source code, which can never be executed[6]. There are usually several reasons, why some parts of source code can be unreachable. There may be a conditional statement with a condition that is always true, or false. Or there can be an infinite loop that prevents the program to continue. Quite typical is also usage of statements that can make a jump in the program execution. It can be `continue`, `break` or `return` statement or a statements that always throws an exception.

There are three main reasons, why unreachable code should be detected and eliminated:

- **Parsing time** – the CTRL code is interpreted, and so the interpreter has to first parse the code. Since it has to parse also the code that will never be executed it wastes the precious time and slows down the actual code execution.
- **Code readability** – if source code contains unreachable parts, it makes it bigger and harder to read for people. It may also cause confusion and hide some bugs.
- **Instructions caching** – if the unreachable code is not eliminated by a compiler or interpreter, it occupies a memory that could be used for instructions that can be actually executed. The instructions are usually stored in a small cache with fast access time and it speeds up the execution. If the cache is filled up with useless instructions, the CPU has to get the instructions from slower caches or RAM at it may significantly slow down the code execution.

Detection of unreachable code usually involves a control flow analysis. In case of the algorithm responsible for unreachable CTRL code detection (see demo CTRL code in Figure 5.10), the algorithm tracked the flow of analyzed program, but it was not a proper control flow analysis as described in compilers-related books[27]. The reasons for simplifying the algorithm were mainly practical. The time for writing the algorithm was limited and there were worries that a proper control flow analysis could slow down the whole CTRL code analysis too much.

Fortunately, the simplified analysis turned out to be efficient enough and it was able to detect most of the common cases of unreachable code.

```
string foo(int x) {
        if(x >= 0) {
                switch (x) {
                        case 0: {
                                return "zero";
                        }
                        case 1: {
                                return "one";
                        }
                        default: {
                                return "too big";
                        }
                }
        } else {
                return "negative";
        }
        return "unknown"; // statement is unreachable
}
```

Figure 5.10: Simple demonstration of unreachable code.

The control flow based approach of unreachable code analysis was not the only one used by the CTRL plug-in. There were also algorithms that could determine whether a condition is always true or always false. Again, these algorithms did not formally check every possible combination of input values and outcomes from called functions. They were simplified versions focused on speed and they tried to detect as many potential bugs in CTRL code as possible during the limited time.

```
void foo(){
        const int x = 2 * 1;
        const int y = 8 + (3 * 0);
        const int z = x * y / 4; // 'z' is 4
        if(z > 4){ // condition in "if" statement is always false
                DebugTN("This code will never be executed.");
        }
}
```

Figure 5.11: Demonstration of an analysis, which is able detect that some conditons are always true or always false.

The examples in Figure 5.11 and Figure 5.12 show how the detection of *always true* or *always false* conditions works. The first example demonstrates the usage for unreachable branches in conditional statements. And the second, more complex example, demonstrates detection of infinite loops. The algorithm can detect that when a condition of `for` statement is always true and the only statement that can break the loop is inside a branch that can never be executed, it deduces that the loop iterates forever.

```
void foo(){
        const int x = 2 * 1;
        const int y = 8 + (3 * 0);
        const int z = x * y / 4; // 'z' is 4
        for (int index = 1; z == 4; index++) { // iteration statement
is infinite loop
                switch (index) {
                        case 1: {
                                continue;
                        }
                        default: {
                                break;
                        }
                }
                if(z != 4){ // condition in "if" statement is always
false
                        return; // this return statement will never break
the loop
                } else {
                        continue;
                }
        }
}
```

Figure 5.12: Demonstration of analysis, which is able detect infinite loops.

5.3.6.5 Dead Code Detection

Dead code is part of the source code, which can be executed (unreachable code cannot be executed), but the result of the execution is never used. In other words, if we remove the dead code from a program, the semantics of the program will not change. It is just waste

of computation time[5].

On the other hand, the execution of dead code can cause a program to crash. For example, if a dead code contains a division by zero, the runtime environment may throw a runtime exception, causing the program to crash.

The IDE contains two relatively fast algorithms for dead code detection. The first one was already mentioned at be beginning of Section 5.3.6 (see Figure 5.8). It finds variables that are not used in the source code. So when a programmer makes some computation that does not cause any side effects and stores the result of this computation in a variable that is never used, the code can be probably removed or refactored.

The second algorithm controls `ExpressionStatements` and it is trying to determine whether all parts of the expression cause any side effects.

The examples of dead code detection are presented in Figure 5.13. The first highlighted line represents a result of computation that is stored in a variable, which is not used in the program anymore. The second highlighted line represents an expression that consists of parts that cause no side effects and can be removed (`x` + result of `DebugTN` function is not stored and used, so `x` + can be safely removed without altering the semantics of the program). The third highlighted part represents an expression (numeric operation) that is computed at the beginning of every iteration, but its computation is useless and it just slows down the execution.

Similarly, as most of the previous algorithms, the priority was the speed. The algorithms were not built for a compiler but for an advanced text editor that had to be responsive. For this reason, the dead code detection algorithm does not use *liveness analysis* (a form of *dataflow analysis*) whose typical time complexity is between $O(N)$ and $O(N^2)$ (the worst possible case is $O(N^4)$)[27]. Therefore, the dead code analysis used in the CTRL plug-in does not detect all possible problems with dead code.

```
void foo(int x){
        int y = x*3; // variable 'y' is unused, so the whole statement
can be removed

        // some parts of this statement may not cause any side effects
        x + DebugTN("Hello world!");
        for(;;10/5){
        }
}
```

Figure 5.13: Demonstration of dead code detection.

5.3.6.6 Other Checks

Apart from the checks and analysis mentioned in the previous chapters, the CTRL plug-in contained many other checks whose purpose was to make CTRL code more readable, performant and with less bugs.

At the end of my stay at CERN, the CTRL plug-in contained around 80 different checks, many of them configurable in the IDE preferences. The following list highlights some of the cheeks:

- **Modification of constants** – immutable variables (like: `const int x = 0;`) or literals (like: `"asd"`, `123`, `false`) cannot be modified. If a constant is modified, the

program crashes at runtime. For this reason, the IDE check that no constants are modified. The IDE also promoted the concept of immutability and it automatically suggested programmers that certain variables can be declared as constants.

- **Shadowing of variables and functions** – shadowing of variables and functions was considered by majority of SCD section members as a bad thing. To discourage programmers from using the concept of shadowing, the IDE checked that no variables or functions, local or imported from libraries, are shadowed.

- **Checking of import statements** – one of the checks was related to import statements in CTRL code (`#uses "..."`). The check controlled that the names of variables and functions from imported libraries do not cause any collisions, and that all import statements have a justification. So if a library was imported, but the local code did not use any of its variables or functions, the useless import statement was marked as unused and users were advised to remove such statement. To make it easier for users to manage the import statements, the IDE had a functionality that automatically added necessary import statements and removed unused ones. This made CTRL code cleaner and faster since the CTRL interpreter did not have to load unused libraries at runtime.

- **Array index check** – the index of arrays in CTRL language starts at 1, which is different from most of general purpose programming languages, where the array index starts at 0. For this reason, the IDE controlled initialization of index variables in `for` loops.

- **Variable initialization** – the IDE checks whether every variable is initialized before it is used. Constant variables had to be initialized at the declaration statement. Mutable variables had to be initialized either in every single branch that preceded their usage or they had to be passed to a function as so called *in/out* argument. Demonstration of this check can be seen in Figure 5.14.

```
void foo(int x){
	// constant has to be initialized at the place of its
declaration
	const bool b = false;

	int a;
	bar(a); // is passed to function as in/out argument

	int y;
	if(x > 0){
		y = x;
	} else {
		y = 0;
	}

	// variable 'y' has to be initialized in every single branch
(even nested branches)
	// otherwise the IDE will display a warning
	DebugTN(y, b, a);
}

void bar(int &param) {
	param = 5;
}
```

Figure 5.14: Initialization of variables in CTRL code.

5.3.6.7 Converting EMF into real AST

Some of the static code analysis/check algorithms were relatively simple, some of them were complicated. But there was one thing, they all had in common. If an algorithm had to analyze an `if` or `switch` statement, then the core of the algorithm became hidden behind a lot of strange, hard to grasp, code that had to deal with the unfriendly EMF model structure (for more information, see the *dangling else problem* in Section 3.3.1.3).

For this reason, it was decided that the parts of the EMF model, that are hard to analyze, will be converted into a more abstract form, which is better for code analysis algorithms. For example, `if` statements were converted into an immutable class that contained an optional condition (if the condition was missing, it was not an `if` statement, but `else` statement) and the contents of the relevant branch. Related `if/if-else/else` statements were then stored in a class that contained immutable list.

Such data structure allowed to significantly simplify some algorithms, as demonstrated in Figure 5.15. The algorithm returns `true/false`, depending on whether a function of the given name was checked for existence in the given context. The algorithm itself is relatively straightforward. It takes an iterator that can walk through parents of the context, and filters only those parents that are `if` statements. Then it checks whether there exists at least one `if` statement, whose condition makes sure, that a function of the given name exists at runtime.

```
/**
 * This method is trying to determine, whether there is an
"isFunctionDefined(...)" check before
 * the function is called.
 * Example:
void main(){
   const string funcName = "foo";
   if(isFunctionDefined(funcName)){
      foo(1, 2, 3); // contextNode is here
   }
}
 *
 */
def static isInsideFunctionDefninedCheck(String functionName, EObject
contextNode) {
   val parents = ASTHelper::getParents(contextNode);
   return parents
      .filter[CtlASTIdentifier::isIFstatement(it)]
      .exists[
         val ifStatement = it as SelectionStatement;
         val abstractIF =
CtlASTIfStatementHelper::convertIfStatementToAbstractForm(ifStatement)
;

         /**
          * Try to find context in IF content and select right "IF
branch"
          */
         val parentIf =
abstractIF.IFs.findFirst[it.content.eAllContents.exists[it ==
contextNode]];

         /**
          * If the condition is not present,
          * it is probably ELSE statement, so we don't care about it.
          * Otherwise we check the IF condition.
          */
         if(parentIf != null && parentIf?.condition.present){
            return isPositiveFunctionDefinedCheck(functionName,
parentIf?.condition.get);
         }
         return false;
   ];
}
```

Figure 5.15: An algorithm written in Xtend that takes advantage of abstract representation of `if` statements.

If the algorithm would not work with the abstract representation, it would be approximately 3 times longer, hard to read and maintain even for people with deep knowledge of EMF models and Xtext grammar language (for more information about how is are conditional statements handled, see classes `CtlASTIfStatementHelper` and `CtlASTSwitch-`

`StatementHelper`).

Converting EMF into real AST The main problem of EMF model is that it is generated and tightly bounded to grammar. Therefore, a change of grammar causes changes of EMF model and this could produce hard to detect bugs.

It happened several times during the development that the CTRL grammar had to be changed (optimized or fixed) and because of the changes I had to check most of the code that relied on the generated EMF model in order to make sure that nothing is broken.

These problems would vanish if the algorithms would work with a model that is stable, easy to navigate in, strongly typed and with optional links to the EMF model or parse tree.

One of the solved problems would be the *dangling else problem*, easier access to `case` statements, function arguments, etc. and the possibility to get rid of unintentional null pointer exceptions by introducing `Optional` data type. Not to mention that the model could be less memory demanding and immutable so multiple threads could do the analysis at the same time.

Additionally, the AST model could contain optional links to the original EMF model or parse tree so the access to different levels of source code abstractions would be preserved.

In case the grammar was somehow changed, the only piece of code, that would have to be modified in most of the cases would be the layer that provides translation of the EMF model to the AST model. And since this code would be in one place, the chances that a bug was introduced by the change, would be lowered.

As for the AST model implementation, probably the best solution would be to use *discriminated unions*. Discriminated unions are often used to describe more complicated data structures and they are suitable for AST model descriptions. Unfortunately, Java and Xtend do not support such data types. So the AST and related algorithms would have to be implemented in a language like Scala, that supports this concept and it also provides pattern matching, which would make it really easy to navigate through AST models and create algorithms for static code analysis.

Converting EMF/AST into General-Purpose Semantic Model Another considered approach that could solve the EMF-related issues, was to move the model used for code analysis to even higher level of abstraction and increase its reusability.

The idea here is that a general-purpose semantic model could be used not only for CTRL language, but also for other languages. Reading this, one could say that we already have such model, it is called EMF model and it is used by Xtext. Therefore, inventing a new model for AST representation is like walking in circles. This is partially correct, but EMF models do not have any semantics and they are too generic.

The model I am proposing would be composed mostly of nodes with well defined meaning. For example, there would a node that represents conditional statements. Such node would contain an information about the statement conditions, branches and in theory it would allow users to inject some additional meaning using other objects. So at the end, analysis tool developers could take their language-specific AST and convert it into the generic model in such way that the semantics of both models would be the same. The generic algorithms in background could automatically do all kinds of control flow, dataflow and other types of analysis.

Again, this is just an idea. I did not do any research in this area, I do not know if such model already exists (I was not able to find it), in case it really exists or it is possible to implement such concept, I do not know what are the potential drawbacks of this idea. I

am just saying that from my point of view, this could solve some problems I had to deal with during the CTRL code analysis.

5.3.6.8 Standalone Static Code Analysis

The information provided by the CTRL code analyzer turned out to be beneficial for programmers, especially when they were writing a code according the SCD section-defined standards. Even little things like keeping the names of variables under certain length or cleaning up the code by removing unused variables and unreachable parts of the code, increased the overall code quality.

In order to do the analysis globally, on SCD WinCC OA projects, it was decided that the analysis should be executable also from command line (see Figure 5.16). For this purpose, a new *headless plug-in* was created (see class: `ch.cern.winccoa.ctl.codeanalyzer.-Application`).

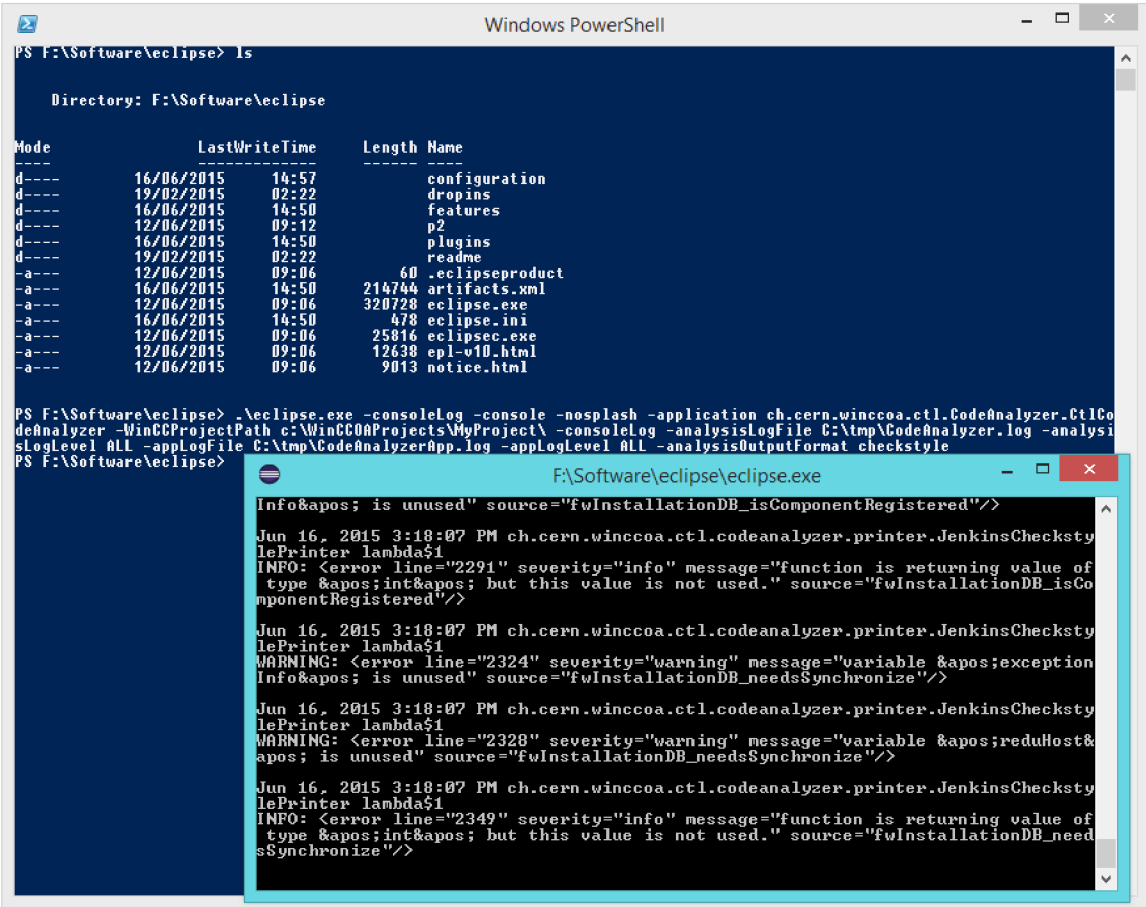

Figure 5.16: Screenshot showing the CTRL code analysis executed from a command line.

This plug-in allowed programmers to specify what is the WinCC OA project that should be analyzed, what should be the format of the output (checkstyle, XML, JSON, HTML (see Figure 5.17), human-readable), where to store the analysis result, log files, etc. and what are the preferences for the code analysis.

The possibility to run the analysis from console was crucial for the integration with Jenkins (a continuous integration tool) that could run the analysis periodically and then

provide reports, showing whether the number of warning and errors is increasing or decreasing over time.

An interesting way of using the headless version of CTRL analysis emerged when a short-term trainee (student) in the SCD section was given a task to analyze the length of variable and function names used in CERN CTRL code. He used the plug-in ability to load analysis preferences from a file and he altered the preferences in such way, that the code analysis produced a warning every time it detected a variable or function with a name longer than 0 characters (which was always). Then he used a Python script to extract the data he need from the CTRL analysis results, and used it to do his own analysis of the IDs length.

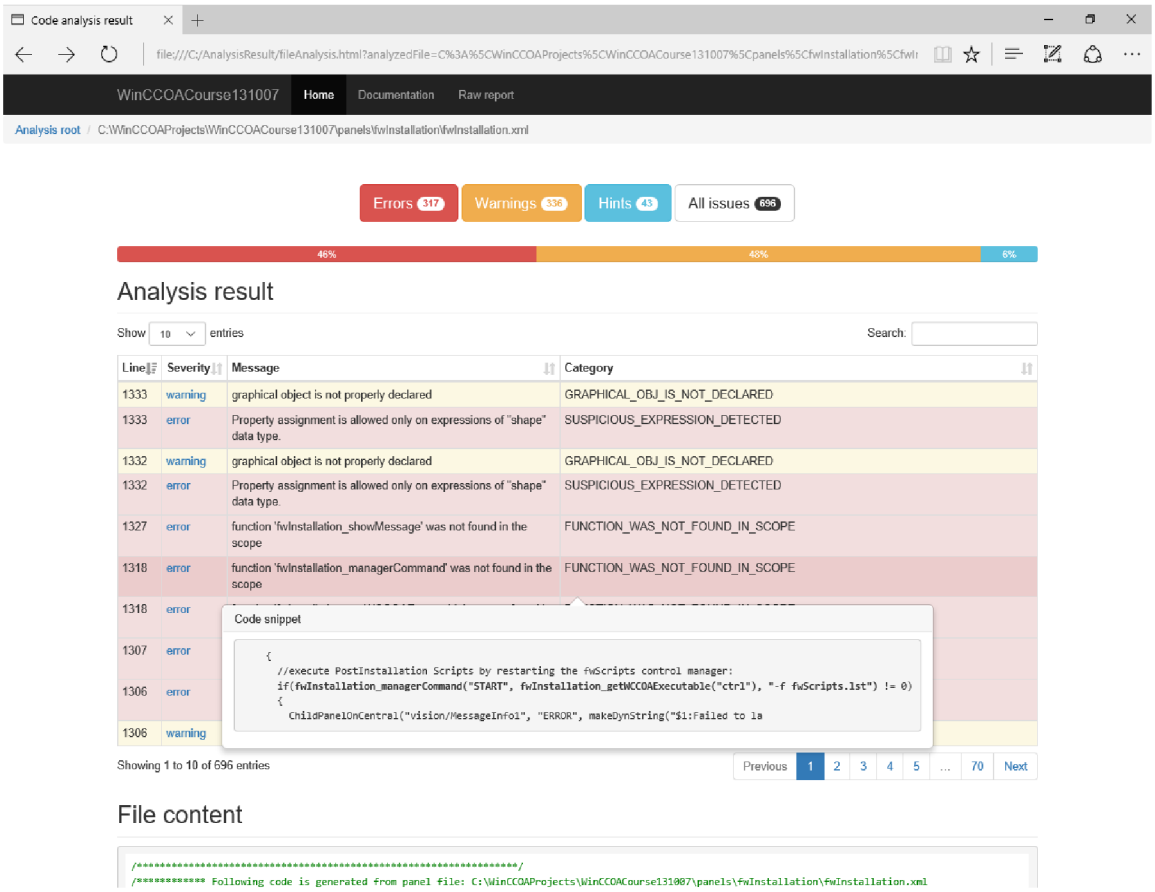

Figure 5.17: Result of CTRL code analysis in HTML format.

5.3.7 Panel Files Support

The WinCC OA software used so called *panel files* for GUI part of SCADA applications. The panel files were available in two interchangeable formats: PNL and XML, both describe GUI elements (panels) and their actions (using CTRL code).

The PNL format is proprietary with no public documentation. To decode its contents a reverse engineering would have to be done. However, this approach could lead to bugs and it was not considered as safe. The XML format was on the other hand easy to decode and convert into in-memory model using standard Java technologies.

The very reason, why the IDE supported the panel files, was that they also contained

CTRL code. The created *panel file editor* showed in Figure 5.18, provided access to the CTRL code of individual GUI elements of the panel file and their events (each GUI element supported several events, for example *Clicked*, and each event could contain a CTRL code that was executed when the event was triggered).

The editor itself was composed of two sub-editors. The one showed in Figure 5.18 allowed users to select GUI elements, their events and edit the CTRL code using the embedded CTRL editor. The second sub-editor (*XML Editor* tab in bottom part in Figure 5.18) provided access to the XML representation of panel files and it allowed users to change the GUI contents of the panel.

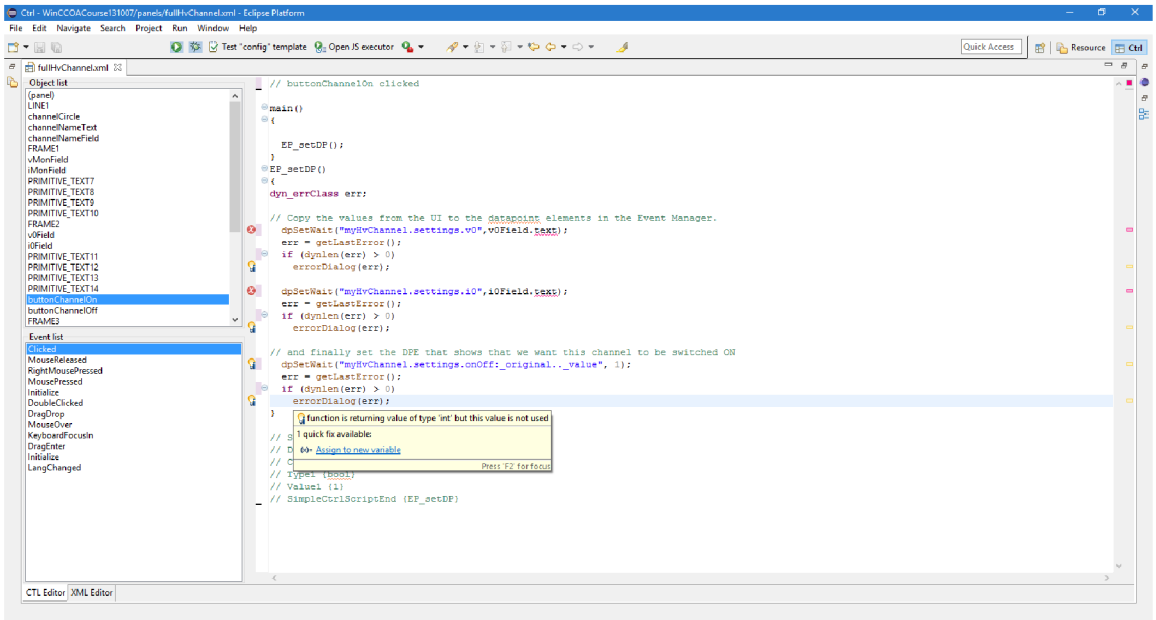

Figure 5.18: Demonstrative usage of panel file editor in the IDE.

As for the PNL format, the IDE used the ability of WinCC OA to convert PNL files into XML format and vice versa. Unfortunately, this feature relied on calling WinCC OA binary files, so it was available only when it had access to the WinCC OA installation folder.

For more information about the implementation of the panel editor related code, see the following classes: `CtlXMLMultipageEditor`, `PNLToXMLUtil` and `XmlUtils`.

5.4 Config Plug-in

The purpose of the Config plug-in was to provide WinCC OA developers means that help them to create and edit configuration files for WinCC OA projects. These configuration files, aka *config files*, are important part of WinCC OA, because they configure SCADA applications.

There is various information contained in config files but usually the configuration is related to project paths (dependencies on other WinCC OA projects/applications), names of libraries that should be loaded, port numbers etc. A small example of a config file is shown in Figure 5.19.

```
[general]
pvss_path = "C:/Siemens/Automation/WinCC_OA/3.11"
proj_path = "C:/Documents/ThesisWorkspace/WinCCOAProjects/
WinCCOACourse131007"
proj_version = "3.11"
langs = "en_US.iso88591"
distributed = 1
[ui]
#begin fwAlarmHandling
aesShowDistDisconnections = 0
LoadCtrlLibs = "fwAlarmHandling/fwAlarmHandling.ctl"
LoadCtrlLibs = "fwAlarmHandling/fwAlarmHandlingScreenGroups.ctl"
#end fwAlarmHandling

[mod_2]
maxOutputQueueSize = 600000
addUnicosMarker = 65534
tcpServerPort = 7000
```

Figure 5.19: A small demonstration of how a configuration file for WinCC OA can look like.

5.4.1 Simplified Config Grammar

The grammar for config files is rather simple. The total length is around 60 lines of code and the part that has the biggest impact on the structure of generated EMF model consists of just three rules (see Figure 5.20).

```
ConfigModel:
        sections+=Section*
;

Section:
        '[' name=KEY ']'
        content+=Property*
;

Property:
        ('('param=(SUBSTITUTION|KEY)')')? name=KEY '=' values+=Value+
;
// ... rest of the Config grammar
```

Figure 5.20: The part of the Config grammar that had the biggest impact on the AST structure.

Basically a config file can contain an arbitrary number of sections (`[section_name]`) and each section can contain an arbitrary number of properties with some values assigned (`propName = "string value" 132`). So fairly simple language with equally simple grammar.

To make it even simpler and more user friendly, the grammar does not contain rules for

new lines. Essentially every section name and every property should be on a separate line. The problem was when the grammar contained the *new line* rules and users did not put section names and properties on separate lines. In that case the Config parser could not parse the code and the IDE displayed a cryptic message that supposed to tell the users to put the statements on separate lines.

To solve this issue, the *new line rule* was removed from the grammar and put into the language *validator* that could then analyze the AST and display a human-readable error message if needed.

5.4.2 Data-Driven Approach

Probably the most interesting thing about this plug-in is, that many of its core features are data-driven. The idea is that we provide the plug-in some metadata (knowledge) about configuration files and the plug-in will be able to use this metadata to validate the actual configuration files and provide a code completion and documentation to the users.

The advantage of data-driven approach here is that it makes it easy to modify the behavior of the plug-in. We do not have to change any Java or Xtend code. We simply change the data that drives the plug-in and the plug-in changes its behavior automatically. Theoretically we do not even have to compile the plug-in. It is enough to open the compiled JAR file, find the metadata in text form a change it.

5.4.2.1 Metadata Format

The default choice for the metadata format was XML. This format has the advantage that it is easy to process by Java Architecture for XML Binding (aka JAXB) and it is a text format, so it can be altered by people with basic knowledge of XML.

Furthermore, it is possible to formally describe XML elements using XSD (XML Schema Definition). One of the usages of XSD was that we used it to precisely describe the format of the plug-in metadata. Therefore, it was clear how the format should look like and it reduced the probability of misunderstanding the documentation of the format.

We also used the XSD of the data format to generate annotated classes for the Java model. The tool for the classes generation is called XJC (included in JDK) and it makes it very convenient to work with XML documents in Java.

```
<config-metadata>
   <section>
      <name>general</name>
      <extends>all_sections</extends>
      <content>
         <property>
            <name>pvss_path</name>
            <number-of-occurrences>x == 1</number-of-occurrences>
            <description>Defines the path to the directories, which
includes the WinCC OA static files
(such as executables, error texts, icons, etc.).</description>
            <values>
               <value>
                  <type>string (file)</type>
               </value>
            </values>
         </property>
         <property>
            <name>pmonPort</name>
            <description>The port for the TCP/IP communication with
Pmon.</description>
            <values>
               <value>
                  <type>integer</type>
                  <default-value>4999</default-value>
                  <range>x &gt;= 1024 &amp;&amp; x &lt;= 65535</range>
               </value>
            </values>
         </property>
      </content>
   </section>
</config-metadata>
```

Figure 5.21: A small sample of XML data for Config plug-in.

As shown in Figure 5.21, the structure of the metadata for Config plug-in was similar to the structure of config files. It describes individual sections, properties and their values. Apart from the basic information, the metadata contained also information about types of the values, how many times a property can be repeated, or what is the range for certain values.

To make the metadata really descriptive, it also contains pieces of JavaScript, which is later interpreted by the plug-in. For example, to specify that a value can be greater or equal than 1024 and at the same time lower or equal than 65535, it is enough to write: `x >= 1024 && x <= 65535` (see Figure 5.21). The variable `x` is then substituted by the real value at runtime and the expression is evaluated to a boolean value.

Additionally, the XML format allows inheritance. This is particularly useful when some sections are supersets of other sections. In such cases it is enough to specify that a certain section extends some other section and it can or cannot contain certain *properties*. An example of inheritance can be seen in Figure 5.21, where the `general` section inherits from `all_sections` section. Apart from making the metadata file more maintainable, the inheritance also saved the memory and increased the speed of the XML file parsing.

5.4.2.2 Extraction of the Metadata from WinCC OA Documentation

Defining the XML metadata format and implementing its parsing was barely half of the work that needed to be done. It was also needed to create the actual metadata for the plug-in.

In order to do so, somebody had to take the WinCC OA documentation and rewrite it to the XML format. Luckily, the documentation is available in HTML format and the part describing the entries of configuration files is put in an HTML table (*CFG_DOKU_en.html*).

To extract the necessary information, from the documentation, a Python script was written by my colleague Łukasz Góralczyk. This script uses the *Beautiful Soup* library to pull the metadata out of the HTML documentation and converts it into the XML format. The resulting XML metadata was more than 11 000 lines of code long and it still needed some final touches to make it work properly.

The final adjustments of the metadata file were done in Notepad++ and they mainly consisted of using regular expressions to find pieces of text and replacing them with some other text. To check whether the file contains a valid XML contents, the `ConfigMetadataProvider` class was created. This class prints out the problems that occurred during XML parsing and it was later used for loading the data at runtime.

5.4.2.3 Using the Metadata

Having the XML metadata loaded in the Java model generated from the created XSD was helpful, but it still was not enough to make well maintainable data-driven plug-in. The problem was similar to the CTRL plug-in. Having a model that represents data (CTRL code or XML data) does not always mean that the model is suitable for being used by certain algorithms.

For this reason, the XML model is converted into a different type of model. The structure is similar to the config AST and the XML metadata model. The model contains parts that represent sections, sections contain information about properties and properties have information about possible values. The difference here is that this model does not only describe some metadata (i.e. contents of the XML), but it also knows the meaning of the metadata.

For example, the new representation of the data contains the information about value range. It contains string `"x >= 1024 && x <= 65535"` and this string can be (and is) provided to the component responsible for displaying the documentation. But additionally it knows what this string means so it can answer the question: *"is this value in the given range?"*.

The conversion of the XML model to the new one was really convenient and it made writing the algorithms for configuration files validation, code assist and inline help providing, much easier. For more information about the implementation, see the following classes: `ConfigMetadata`, `ConfigMetadataProvider`, `ConfigMetadataConverter`, `ConfigEntries` and `ConfigValidator`.

5.5 Config Template Plug-in

An introduction of how the Config Template plug-in works was already covered in Section 3.3.3.2. Now, we will have a closer look at some of the core components. For more

information about the Config Template language itself, see Section 5.5.1.1 and the language specification (described in *Config_template_language_description.docx*, attachments).

5.5.1 Config Template Grammar

The Config Template language has a different purpose than the Config language itself. While the Config language describes WinCC OA configuration files and its Eclipse plug-in makes sure that the configuration files are according the official WinCC OA documentation, the Config Template language describes what should be the contents of the configuration files and it is much more powerful than the XML metadata mentioned in Section 5.4.2.

One of the main requirements raised by my supervisors was that the new language has to be similar to the Config language. So for the basic usage, users do not have to learn anything new. For this reason, the new language became practically a superset of Config language (something like TypeScript is a superset of ECMAScript 5[20]).

Someone could think that when we have a language that extends another language, we can simply use some `extends` keyword in the new grammar, add few more syntactic rules and it is done. Unfortunately, it does not work like this. The grammar (and so the generated EMF model/AST) is very different from the Config grammar. The only thing that was reused, were the terminals.

5.5.1.1 Designing the Language

In order to write the grammar, the new language had to be first designed. This part of the development process was especially tricky since it was the first programming language I was ever designing, and I had to create it from scratch.

There were almost no books that I could use for guidance in this area. And since the basic requirements were very specific, I could not even find any programming language that could be used as an inspiration.

I have first started with the *config file* syntax and then I was adding a new syntax whose purpose was to satisfy the requirements. The first syntax draft looked similar to *config file* with optional Java annotations. But with more requirements, the syntax became less clean and I had to start again. Basically every new iteration of the language was discussed with potential users and at the end of almost every discussion a new feature request was raised.

This drove me to a realization that the users want the language to be able to do almost anything. It looked like in most of the cases, they are satisfied with the basic syntax and features, but they were always able to come up with cases when they needed to do something special. Something that was hard to express using a pure declarative language.

After a few iterations of the language design, I came up with an idea to split the language into three "layers". The basic and simplest layer consisted of *config file* syntax. It gave users the possibility to copy-paste the contents of a *config file* into a template file. Such template then described the config file contents. The advantage of this layer is that it is really simple, users do not need any additional knowledge and the template can be easily extended.

```
[general]
eventPort = 1212
[ui]
```

Figure 5.22: An example of basic Config Template language syntax.

The meaning of the template in Figure 5.22 is that a tested config file has to contain a `general` section with a property `eventPort` and the property has to have value `1212`. Additionally, the config file has to contain an `ui` section with arbitrary contents.

The second layer enriches the first one. It allows users to specify what can be the values of some properties or how many time can be a property repeated in a config file. An example of this syntax can be seen in Figure 5.23. The example represents a template that requires config files to have a `general` section and this section cannot contain `pmonPort` property with values `2048` or `4096`. This section can also contain one or more *proj_path* properties and each of these properties has to have a `string` value starting with „`C:/WiCCOAProjects/"`.

```
[general]
{never} pmonPort = 2048 || 4096
proj_path = startsWith "C:/WiCCOAProjects/"
```

Figure 5.23: A demonstration of extended syntax of Config Template language.

The last layer is the one that gives users the ability to do almost everything. It is a combination of special syntax and JavaScript. We will not discuss all the details here but there are a few interesting features of this layer that are worth mentioning. The first feature is that using a JavaScript code, users have access to the whole AST of currently tested config file (the AST is provided by the engine in form of a JavaScript object). Therefore, users can write very sophisticated templates. The second feature is that a JavaScript code can be easily used to call the JVM. This implies that all Java libraries we have, are available to Config Template programmers.

```
[general]
{''(passed == 0 && failed >= 0)''} pmonPort = 2048 || 4096
proj_path = ''(value.startsWith("C:/WiCCOAProjects/"))''
```

Figure 5.24: A demonstration of how a JavaScript code can be used in Config Templates.

The last noticeable feature is that the syntax allows users to inject a JavaScript code into the two previous layers. An example of this syntax is in Figure 5.24, where the attribute `never` is replaced by a JavaScript code of the same meaning: `"(passed == 0 && failed >= 0)"`.

The examples in this chapter show only portion of the language syntax and features. Language features like immutability, which makes it easy and safe to share parts of template code among multiple templates, are not even mentioned here. For more information, see the language specification in the attachments.

5.5.1.2 Config Template Grammar Implementation

When I knew, how the language should look like, I could finally implement the lexer and parser, using the Xtext grammar language and Nashorn JavaScript engine.

The reason why I used these two software components to parse the whole language is simple. It would not make any sense to implement JavaScript parser using Xtext. First of

all, it would be really difficult to write the whole JavaScript grammar. It would also make the Xtext-generated parser much slower, and in order to executed it by Nashorn engine, I would still have to translate parts the AST back to JavaScript source code.

So at the end, the language grammar implemented in Xtext covers only two of the three layers described in Section 5.5.1.1. The JavaScript layer is simply skipped and stored in the AST as a string. The terminal rules responsible for skipping the JavaScript code are in Figure 5.25.

```
terminal SCRIPT_CODE:
        "'''" -> "'''"
;

terminal ALGORITHM_CODE:
        "'''{" -> "}'''"
;

terminal EXPRESSION_CODE:
        "'''(" -> ")'''"
;
```

Figure 5.25: Terminal rules in Config Template grammar. These rules effectively skip all the JavaScript code.

The whole grammar is around 260 lines of code long and it can be found in *ConfigTemplate.xtext* file in the attachments. The other grammar related code is in `ch.cern.winccoa.configTemplate.grammar.converter` package and it mostly contains classes that convert the terminals in Figure 5.25 into JavaScript code (they cut off the special quotes: `"'`, `"{` and `"(`).

5.5.2 Config Template Engine

Config Template engine is the interpreter of Config Template language. A schema of how it works was already presented in 3.3.3.2.

The engine requires two AST models as its inputs. The first model represents the Config Template source code (instance of `ConfigTemplateModel` class) and the second model represents WinCC OA configuration file (instance of `ConfigModel` class).

The dependency of the engine on `ConfigModel` class means that the Config Template plug-in cannot be built separately and it has to be distributed together with the Config plug-in. Fortunately, this was not an issue since the dependency was described in the plug-in manifest and the plug-ins were officially distributed together anyway.

5.5.2.1 Model Conversion

Both, Config Template AST and Config AST were not suitable for a direct interpretation/execution. For this reason, the engine converts both ASTs into different forms.

There are two core convertors: `TemplateModelConverter` and `ConfigModelConverter`. Both work similarly. They contain number of methods and each method can convert one part of the AST. The conversion process usually starts with the AST root and it walks the

tree structure, converting subtrees into a new type of model and assembling the results of the method calls into a new tree-like model.

The result of the conversions are two models. The model created from a Config Template is ready to be executed, and the model created from a config file can provide information about the config file in JSON format (it is pretty much config file AST in JSON).

5.5.2.2 The Role of Immutability

An important feature of the models created for the purpose of execution is that they are immutable. The reasons are performance and thread-safety. Immutable data can be shared among multiple threads without any need for data locking or synchronization. This makes the code faster to execute and easier to maintain. Additionally, immutable models can be reused, which saves memory and also results of certain computations can be cached.

From purely programmer's point of view, the biggest problem with implementation of immutable classes, was Java language itself. Java does not go well with the concept of immutability. Of course, it is possible to use `final` keyword, but it is not enough. Many classes require usage of collections and collections in Java are mutable.

It is possible to use `Collections.unmodifiableList(java.util.List<? extends T>)` method, but it is a poor man's solution. First of all, the method returns only an unmodifiable view of given list. That means, if the original list is changed, the data of the view also changes.

Secondly, the method returns an object of `List<T>` type, which does not prevent other programmers to call `add(E e)` method and crash at runtime. This a big problem if programmers do not read the documentation and rely only on IDE code assist.

And lastly, the code looks ugly and it is hard to read. If we really want to create a list, that cannot be modified at runtime, we have to write a code similar to this: `Collections.unmodifiableList(new ArrayList<>(someOtherList))`.

The solution is to use an external library. For example, *Guava* is a very good library created by Google. Unfortunately, adding too many dependencies in a project may not be a good idea from long-term perspective. Some libraries may not work in future versions of Java, their development may be canceled etc. For this reason, it was decided that the engine will use only the classes from the IDE and JDK.

5.5.2.3 Engine Execution

The architecture of the engine is a mixture of AST interpreter and JavaScript (Nashorn) interpreter. We will not discuss the details here, but the fundamental principle is that the model representing the template is partially translated to JavaScript and executed by Nashorn engine.

The translation of certain parts into JavaScript is important because the Config Template may contain a JavaScript code, which has to interoperate with the template and vice versa. For more information about the implementation, see `ConfigTemplateEngine` class.

As regards the engine performance, it was not an intention to create a highly-optimized interpreter. However, the performance is more than sufficient for the usage at CERN. The standalone version of the plug-in is able to detect multi-core processors, and when more config files are passed as an input, the execution in done in parallel (multiple threads share the same *execution model* of the template, because the model is immutable).

5.5.2.4 Engine Output

The output of the engine execution consists of three parts: *STD_OUT*, *STD_ERR* and *failed-rules stack trace*.

STD_OUT and *STD_ERR* are channels for developers. Developers can use them to print out their own debug messages, contents of exceptions, etc. It is analogous to *stdout* and *stderr* in C language. The main thing about these channels is that the engine does not check them. So even if there is something in the *STD_ERR* channel the matching of the template and the config file may still succeed.

The part of the output that gives users the answer whether the template matching succeed, is so called *failed-rules stack trace*. If the stack trace is empty, then the *root rule* of the template succeeded and the template matches the tested config file. Otherwise if there are items in the stack trace, then the template does not match the tested config file and the contents of the stack trace is printed out.

Example of a template that does not match the config files is in figures 5.26, 5.27 and 5.28. An interesting thing about the template is that it uses the built-in `config` variable in order to access the config file AST, then converts the AST-object into a JSON string and then prints the string in *STD_OUT*.

```
'''
var AST = JSON.stringify(config, null, 3);
print("Config AST:\n"+AST);
'''
[general]
pmonPort = > 1616
project_path = "C:/WiCCOAProjects/"
```

Figure 5.26: An example Config Template code.

```
[general]
pmonPort = 8
project_path = "C:/WiCCOAProjects/"
```

Figure 5.27: An example config file.

As for the printed stack trace, it provides a detailed information about the problem and how it propagated all the way to the *root rule*. On the highest level (`Fail level [0]`), we see that a failed rule caused that the template did not match the config file. On level one (`Fail level [1]`), we see that the problem occurred in the `general` section. If we go all the way down to the lowest level (`Fail level [7]`), we can see the exact code that was generated by the engine and executed by Nashorn.

```
Std out:
Config AST:
{
  "general": [
    {
      "name": "pmonPort",
      "values": [
        8
      ]
    },
    {
      "name": "project_path",
      "values": [
        "C:/WiCCOAProjects/"
      ]
    }
  ]
}

Std err:

Fail level [0]: Template does not match given config file.
Fail level [1]: Section rule "sectionName == "general"" failed - "counter guard" script did not return 'true' value:
Executed code: (passed > 0 && failed == 0)
Fail level [2]: Section rule - some sub-rules failed.
Fail level [3]: Property rule "propertyName == "pmonPort"" failed - "counter guard" script did not return 'true' value:
Executed code: (passed > 0 && failed == 0)
Fail level [4]: Property rule - some sub-rules failed.
Fail level [5]: Property value rule failed - "counter guard" script did not return 'true' value:
Executed code: (passed > 0 && failed == 0)
Fail level [6]: Property value rule failed while testing: {"name" : "pmonPort",
"values" : [8]} >>> line: 2
Fail level [7]: JavaScript Expression rule - failed
Additional info - RuleExecutionResult [message=JavaScript Expression rule - failed,
detailMessage=// INIT SCRIPT BEGIN
var __internal_inputData = {"sectionName" : "general",
"propertyName" : "pmonPort",
"value" : 8,
"values" : [8]};
var __internal_inputData_keys = Object.keys(__internal_inputData);

for(var __internal_i = 0; __internal_i < __internal_inputData_keys.length; __internal_i++){
        var __internal_key = __internal_inputData_keys[__internal_i];
        var __internal_value = __internal_inputData[__internal_key];

        var __internal_toEval = "var "+ __internal_key + " = "+JSON.stringify(__internal_value)+";";
        eval(__internal_toEval);
}
// INIT SCRIPT END

//User's code begins here
(value>1616);,
exception=null,
subResults=[],
status=FAILED,
templateContext=null,
configContext=null]
```

Figure 5.28: The output of Config Template engine when it executes code in Figure 5.26 and Figure 5.27.

For people less familiar with JavaScript, the first part of the generated code is basically an initialization of variables available for Config Template developers. The last part is the actual algorithm/rule, created by a developer. The reason why all the JavaScript code is printed out is that sometimes, developers are not sure, where is the problem. With all the JavaScript code available, they can simply copy-paste the code into a debug window (part of the IDE) that is directly connected to the same JavaScript engine used by the Config Template interpreter and do the debugging there.

Apart from the human-readable output presented in Figure 5.28, there is also a possi-

bility to print out the whole output in JSON format. This allows further processing, for example by Python scripts. The option of printing out the JSON format is available via the standalone version of the plug-in. For more implementation details, see `ExecutionResultJSONPrinter` class.

5.6 Working with Xtext and Xtend

Xtext and Xtend were fundamental components of the development process. Both tools were already mentioned in previous chapters. But they were described mostly from theoretical perspective. In this chapter I will look at Xtext and Xtend from a practical point of view and I will describe, what was my experience with both tools.

5.6.1 Xtext Experience

This framework is truly remarkable and it speeds up the whole development process of domain-specific languages. When there is a simple DSL that needs integration with Eclipse and Java, Xtext is the right tool to use. By default, it provides many basic features: AST, cross-references, code assist, outline view, formatting, code coloring, code folding, etc. All derived automatically from the language grammar. This was especially handy when I was working on Config and Config Template plug-ins. There is also almost no need of touching the Eclipse API directly or to work with SWT. All these things are hidden, so the learning curve is relatively steep, and I, as an Xtext-beginner, was able to learn the basic concepts fast. It took me only few days to understand the Xtext project structure, the elementary idea behind dependency injection and EMF models.

On the other hand, using Xtext for more complex languages, or to create some specific plug-ins, is not a good idea. If the language is complex, then the parser performance may not be sufficient and this alone can be cause of serious problems (the whole code editor can be unresponsive). Other problems may cause the generated AST, which may not be suitable for code analysis (see Section 5.3.6.7).

Additionally, doing something that Xtext does not support may turn into a never-ending fight between the developer and the framework. I have to admit that I was really struggling when I needed to implement something that was not aligned with Xtext and Eclipse concepts. First of all, the documentation and books did not cover other than usual use cases. And secondly, the community on Xtext related forums was not really active and helpful. So in the case, when the developed plug-in/language is somehow specific and it could collide with the way how Xtext does it, I would recommend to use something more flexible, like: ANTLR or FsLex/FsYacc.

- ANTLR - http://www.antlr.org/
- FsLex/FsYacc - https://github.com/fsprojects/FsLexYacc

My last Xtext related note is about the enormous abstraction level of certain classes. Sometimes the main purpose of Xtext classes is hidden under many layers of interfaces, abstract classes, inheritance and dependency injection. The result was that every now and then, I had a hard time to understand what certain classes are actually doing, how they are doing it, why is it the way it is and how to extend or replace their implementation so I can achieve my goals. An example of a class that was hard to understand, is `DefaultOutlineTreeProvider`. This class is usually used to adjust the outline view and for me, extending this class was counterintuitive and it required a lot of debugging and testing.

5.6.2 Xtend Experience

As for Xtend language, I like it. It is a really good tool, especially when working with ASTs. It makes the code shorter, more expressive and easier to write. The seamless integration with Java is also a big advantage. It did not force me to stick with only one language, but I had a choice to use whatever I considered to be the best solution for given task.

Probably my biggest complain about Xtend is that the language is not stricter. It gives programmers a freedom of choosing from several programming styles, and this is the root cause of code inconsistencies. Of course, this is more about personal preferences, but from my perspective, stricter languages are safer, more readable and easier to learn.

An example of code inconsistency can be seen in Figure 5.29. The example shows how the same lambda expression can be written in four slightly different ways.

```
var lambda = [String s | s.length]
var (String)=>int lambda = [s | s.length]
var (String)=>int lambda = [it.length]
var (String)=>int lambda = [length]
```

Figure 5.29: Demonstration of lambda expressions in Xtend. All 4 lines are semantically equal.

Personally I like when there are multiple possibilities of expressing the same thing. Depending on situation or my intentions, I can use different programming styles. But each possibility has to be fundamentally different.

For instance, let us have a look at `for` loops and LINQ in C#. I can use both language construct to filter or transform a collection. But both constructs are very different from each other, and they are meant to be used in different situations. Having a lot of optional syntax in Xtend does not give me the possibility to express my intention in different ways. It just gives me a possibility to omit some optional language constructs and make the code inconsistent.

5.7 Results

In terms of the aims that were set (see Section 2.4), the achieved results are satisfying. The developed IDE is functional and even with some known minor bugs, it is ready for real-life usage by developers at CERN.

5.7.1 Personal Observations

From what I observed of how users were using the IDE, the most used feature was the static code analysis of CTRL code. Compared to the official editor, which had a poor support for semantic analysis, the IDE is able to give users feedback about potential bugs immediately and it results in higher code quality. It was positive to see that some programmers really cared about warning and error messages displayed in the CTRL editor, and wrote the new code in such way that no issues were detected anymore.

Of course, there were cases when the analysis was wrong, and it produced false warnings. However, I was usually able to fix or improve the analysis algorithms rapidly and the more feedback I got from cooperative users, the better the IDE became.

The static code analysis was also used for server-side code analysis. The IDE was integrated with Jenkins and it periodically checked CERN WinCC OA CTRL code. Since the old CTRL code was not written in the IDE and according the CERN-defined rules, the analysis produced a lot of warning and error messages. It did not make much sense to rewrite the old code, because it was working properly (it would take too much time and the new, probably untested code, could contain new bugs). The important information from the code analysis was not the amount of warning and error messages, but rather a trend which would indicated whether the code quality increases or decreases over time.

Another features, which I saw that were often used, were Code Assist and inline documentation. Code Assist was very useful because many global variables and functions in CTRL code have long names (with special prefixes) and automatic completion of these names speeded up code writing. Also people did not have to search the WinCC OA documentation or libraries anymore. The Code Assist displayed a list with all the functions and many of them with inline documentation (either extracted from the WinCC OA documentation or from CTRL code).

A small feature that programmers frequently, indirectly, used was the *import statements organizer*. This feature could be called explicitly or implicitly when Code Assist was used to insert a function or variable from a library that was not yet imported (import statement was missing).

The IDE has several features I expected to be used on regular bases: Outline view, Call Hierarchy, Refactoring and Unit tests management. But it turned out that these features are almost never used. My explanation for this is the following: the users did not have any of these features available in the official WinCC OA CTRL editor and so they were not used to use them.

Instead of using Refactoring tools to rename functions/variables, extract functions/constants or automatically generate debug messages, they used the old-fashion way of copy-pasting pieces of code and "Search & Replace" functionality. Similarly, the programmers used scrolling/"Find Text" rather than Outline View and Call Hierarchy.

As regards the Unit tests manager, this feature was probably not used simply because programmers did not write any unit tests. The original idea was to create this feature in order to make code testing easier and faster. The assumption was that we give programmers a tool for code testing and they will write unit tests. Unfortunately, this assumption was wrong and the amount of new unit tests was negligible.

5.8 Detected Problems

This chapter provides a brief summary of the most significant problems I encountered during the 14-month IDE development period. Most of the problems can be divide into three groups: technology limitations, missing/wrong information and bad design decisions.

5.8.1 Technology Limitations

The biggest technological issues were related to the IDE performance. The Xtext-based CTRL parser was the core component, and its slowness affected many of the IDE features. The techniques of how I solved some of the performance related issues are already described in Section 5.3, but I would like to stress again that using Xtext for big and performance-critical projects may not be the best solution, and developers should consider the possibility of writing their own parser, where they have more control over performance tuning. Besides

that, automatically generated AST model may not be suitable for all types of static code analysis.

An interesting issue I did not expect to occur was that Eclipse does not like working with files outside its current workspace. On the other hand, WinCC OA projects may have dependencies on other projects and files located outside the Eclipse Workspace, and this was a bit problematic. Part of the implemented solution was to translate Eclipse URI style into Java `Path` style and vice versa, but it would benefit the future IDE development to revise this implementation and think about a different resolution of this problem.

A set of problems was also related to SWT (Standard Widget Toolkit), a GUI library used by the Eclipse Platform. Firstly, SWT's dependency on underlying operating system required testing on Windows and Linux (Mac OS X was not officially supported by the IDE). And secondly, the roots of SWT go to early 1990s, and even though it evolved since then a lot, many of the fundamental principles and design decisions remain and in my opinion, they make the library obsolete and hard to use. Also the capabilities and richness of the library are poor, compared to WPF or JavaFX.

5.8.2 Missing Information

During the IDE development, the WinCC OA documentation was my main source of information. Unfortunately, it turned out to be an unreliable source. The information contained in the documentation was often in conflict with reality (the documentation claimed something, but WinCC OA did something else) and some information was simply missing (also some parts of the documentation were written in German).

The quality of the documentation partially affected the quality of the IDE. Some features implemented according the documentation had to be changed because, as users pointed out, the documentation is wrong and in reality some features should behave differently.

One of the main consequences of the problems caused by the faulty documentation was a time loss. I had to fix a number of bugs, just because I was using the wrong information, and I had to start verifying that information stated in the documentation is actually true. In case of missing information, I had to ask my colleagues to spend some of their time to tell me what I needed to know in order to continue developing the IDE.

Chapter 6

Conclusion

This thesis presents an Eclipse-based IDE developed during my 14-month stay at CERN (the origin of the IDE is a research conducted by the Eindhoven University of Technology, Netherlands). The main objective was to build a software tool that would help WinCC OA developers at CERN to create SCADA applications in a more effective and safe way.

The most important feature of the developed IDE is a static code analysis of a general-purpose C-like language. The analysis consists approximately of 80 checks, and it is able to detect code defects like: unreachable or dead code, potential loss of precision (numbers), wrong data types casting, uninitialized variables and incorrect function call arguments. The analysis is also internally capable of simple type inference, prediction of what will happen at runtime (it pre-computes values of certain expressions and also tries to determine existence of functions at runtime) and it can be executed from command line and integrated with tools like Jenkins.

Besides the support of the general-purpose programming language, two additional IDE plug-ins, whose purpose is to verify correctness of configuration files for WinCC OA applications, were created. As part of the effort to verify correctness of the configuration files, a new domain-specific programming language was designed and implemented. This language brings new ideas of DSLs usage and shows how it is possible to combine two different programing paradigms and make them seamlessly work together. It is also an example of how technologies like Xtext, Nashorn engine and JVM can be combined together and serve as a language interpreter.

Large part of this thesis is dedicated to the implementation of the IDE. It covers the most important aspects of Xtext-based plug-ins implementation, discovered issues and solutions of these issues. It also illustrates what are the limitations of used technologies (Eclipse, Xtext, Xtend, JVM) and what are the techniques used to overcome or avoid these limits, while the focus is primarily on Eclipse and Xtext performance.

A recommendation based on my experience gained while working at this project is the following: Xtext is a great framework that simplifies and accelerates development of plug-ins supporting programming languages. It uses a grammar language to create not only a parser, but also a Java model for the AST. Additionally, it brings many features like linking, syntax highlighting, code assist etc. out of the box. Unfortunately, I cannot recommend Xtext as a tool for building complex (general-purpose) languages support. It is mainly due the encountered lack of performance, memory consumption, certain rigidity of the framework and there used to be a problem with the quality of the Xtext documentation.

Some issues were also related to the Eclipse Platform and SWT; the foundations of the IDE. From my perspective, working with SWT is not very productive, and it might

be worth trying Xtext for IntelliJ IDEA or web environment (now available since Xtext version 2.9).

As regards the practical impacts, the IDE is used by SCADA developers at CERN, it periodically preforms static analysis of CTRL code on servers (Jenkins) and it is planned to be used more often for verification of WinCC OA configuration files.

6.1 Future Work

There are several ways of how to improve the quality of SCADA applications at CERN. We can improve the static code analysis (the space here is limited since CTRL is a dynamic language and it is hard to analyze without false warning messages – additional research is probably needed in this area), or we can introduce an automatic generation of unit tests (this research is currently conducted at CERN), or we can improve CTRL language itself, make it more suitable for SCADA development and bug-resistant.

Let me elaborate the scenario of CTRL language evolvement. It consists of two parts. The first part is related to the syntax changes and the second is associated with the execution environment of the language.

CTRL is a relatively simple general-purpose language. It is similar to C, but it does not use pointers, user-defined data structures and some of low level features. Its advantage is that it is easy to learn. People who know C/C++/Java/C# are likely to understand the syntax immediately and start writing CTRL code quickly. The disadvantage is that the language does not provide any SCADA-specific syntax that would make development of SCADA applications more effective. In fact, the lack of user-defined structures and classes leads to nasty hacks that make CTRL code hard to read and debug. By expanding the language functionalities, we could take advantage of features like classes, discriminated unions, generics, lambdas, proper exceptions handling, parallel/asynchronous execution and biding of data points. And this is just a brief example of some basic features that are missing and that could make the code shorter, more readable and more suitable for modern hardware.

As for the second part, changing the execution environment would increase interoperability with other systems/languages and it would bring performance benefits (therefore it would reduce so called "performance hacks" that currently occur in the CTRL code base). The idea here is that CTRL code (the old version or a new backwards compatible version) would be compiled into another language that is fast to execute, it is open and runs on multiple operating systems. The most obvious platforms here are JavaScript and Java/.NET bytecode.

By translating CTRL code into JavaScript, we could increase performance by using some of the high-performant engines like V8 and Chakra. Besides that, we could increase web interoperability and take advantage of some web technologies and large number of existing libraries. The disadvantage of this solutions is that JavaScript is a dynamic language and in my opinion, such languages are not suitable for such important area as SCADA applications.

The other possibility is to use Java or .NET bytecode. In this case, the performance boost would be even more substantial compared to JavaScript, and we could benefit from good interoperability with other JVM/.NET languages (Java, Scala, C#, F#). Additionally, we could use variety of Java/.NET libraries and frameworks.

This scenario of possible future development has definitely some drawback that are yet to be discovered. An implementation of such solution would require further research and close cooperation with the vendor of WinCC OA.

Bibliography

[1] Github - google/guice: Guice. https://github.com/google/guice.

[2] CERN member states. https://commons.wikimedia.org/wiki/File:CERN_member_states_.svg, 2001-.

[3] DFA minimization. https://en.wikipedia.org/wiki/DFA_minimization, 2001-.

[4] Type system. https://en.wikipedia.org/wiki/Type_system, 2001-.

[5] Dead code. https://en.wikipedia.org/wiki/Dead_code, 2001-2016.

[6] Unreachable code. https://en.wikipedia.org/wiki/Unreachable_code, 2001-2016.

[7] Eclipse public license - v 1.0. http://www.eclipse.org/legal/epl-v10.html, 2004.

[8] Ook! http://esolangs.org/wiki/Ook!, 2005.

[9] The art of tokenization. https://www.ibm.com/developerworks/community/blogs/nlp/entry/tokenization?lang=en, 2013.

[10] Brainfuck. http://esolangs.org/wiki/brainfuck, 2013.

[11] CERN-IDE project, presentation. 2013.

[12] Software architecture document, 2013.

[13] -x command-line options. https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/jrdocs/refman/optionX.html, 2014.

[14] Java 8: Compiling lambda expressions in the new nashorn JS engine. http://blog.takipi.com/java-8-compiling-lambda-expressions-in-the-new-nashorn-js-engine/, 2014.

[15] Industrial controls. https://wikis.web.cern.ch/wikis/display/EN/Home, 2015.

[16] Language implementation. https://eclipse.org/Xtext/documentation/303_runtime_concepts.html#linking, 2015.

[17] Language implementation. https://eclipse.org/Xtext/documentation/303_runtime_concepts.html#scoping, 2015.

[18] The redmonk programming language rankings: January 2015. http://redmonk.com/sogrady/2015/01/14/language-rankings-1-15/, 2015.

[19] Tracking resource changes. http://help.eclipse.org/mars/index.jsp?topic=%2Forg.eclipse.platform.doc.isv%2Fguide%2FresAdv_events.htm, 2015.

[20] Typescript. 2015.

[21] Xtend - documentation. https://eclipse.org/xtend/documentation/, 2015.

[22] Xtext - documentation. https://eclipse.org/Xtext/documentation/, 2015.

[23] Eclipse newcomers FAQ. https://eclipse.org/home/newcomers.php, 2016.

[24] Nashorn - openJDK wiki. https://wiki.openjdk.java.net/display/Nashorn/Main, 2016.

[25] The netbeans platform. https://netbeans.org/features/platform/index.html, 2016.

[26] Andris Ambainis, Ashwin Nayak, Amnon Ta-Shma, and Umesh Vazirani. Dense quantum coding and quantum finite automata, 2002.

[27] Andrew W Appel and Jens Palsberg. *Modern compiler implementation in Java*. Cambridge University Press, New York, NY, USA, 2nd ed. edition, 2002.

[28] Lorenzo Bettini. *Implementing domain-specific languages with Xtext and Xtend*. Pack Publishing, Birmingham, UK, 1 edition, 2013.

[29] Noemi Caraban. CERN and science for peace. http://cds.cern.ch/record/1950376, 2014.

[30] Dave Fancher. *The book of F#*. No Starch Press, San Francisco, California, 2014.

[31] David Gallardo, Ed Burnette, and Robert McGovern. *Eclipse in action*. Manning, Greenwich, Conn., 1 edition, 2003.

[32] Steven Holzner. *Eclipse*. O'Reilly, Sebastopol, CA, c2004.

[33] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to automata theory, languages, and computation*. Addison-Wesley, Boston, 2nd ed. edition, 2001.

[34] Dexter C Kozen. *Automata and computability*. Springer, New York, 1997.

[35] Robert Liguori and Patricia Liguori. *Java 8 pocket guide*. O'Reilly, Sebastopol, California, first edition. edition, 2014.

[36] Alexander Meduna. *Automata and languages*. Springer, New York, 1 edition, 2000.

[37] Scott Oaks. *Java performance*. O'Reilly, Sebastopol, CA, first edition. edition, 2014.

[38] Rajani Ramsagar. Eclipse IDE primer. https://tecnoesis.wordpress.com/2009/09/13/eclipse-ide-primer/, 2009.

[39] Jeffrey Richter. *CLR via C#*. Microsoft, Redmond, Washington, fourth edition. edition, 2012.

[40] Łukasz Góralczyk. Email communication with Łukasz góralczyk. 2015-2016.

[41] Raoul-Gabriel Urma, Mario Fusco, and Alan Mycroft. *Java 8 in action*. Manning, Shelter Island, 1 edition, 2015.

[42] Lars Vogel. *Eclipse 4 RCP*. Vogella, Hamburg, 2nd ed. edition, 2013.

[43] Axel Voitier. Test suite for the archiver of a SCADA system, 2010.

[44] Stefan Xenos. Inside the workbench a guide to the workbench internals. https://eclipse.org/articles/Article-UI-Workbench/workbench.html, 2005.

[45] Sebastian Zarnekow. Xtext corner #2 - linking and scoping. http://zarnekow.blogspot.cz/2009/01/xtext-corner-2-linking-and-scoping.html, 2009.

Appendices

List of Appendices

A Demonstration Videos **106**

B Supplementary Information about Programming Languages **107**
B.1 Formal languages (Chomsky hierarchy) 107
B.1.1 Regular Languages 108
B.1.2 Context-free Languages 109
B.1.3 Context-sensitive Languages 110
B.1.4 Recursively Enumerable Languages 110
B.2 Classification of Real-World Programming Languages 110
B.2.1 Classification of CTRL Language 112

C Manual **114**

D Notes **115**

Appendix A

Demonstration Videos

To see the IDE in action, you can watch recorded videos on CERN Document Server:

- *WinCC OA Eclipse IDE - How to install* :
  http://cds.cern.ch/record/2105560
- *WinCC OA Config Plug-in – Introduction* :
  http://cds.cern.ch/record/2040873
- *WinCC OA Eclipse IDE - Create and import project* :
  http://cds.cern.ch/record/2105561
- *WinCC OA Eclipse IDE - Code assist and importance of config file* :
  http://cds.cern.ch/record/2105562
- *WinCC OA Eclipse IDE - Code refactoring* :
  http://cds.cern.ch/record/2105563

Appendix B

Supplementary Information about Programming Languages

B.1 Formal languages (Chomsky hierarchy)

Following types of formal languages are based on Chomsky hierarchy (see Figure B.1). This hierarchy is related to the hierarchy of classes of formal grammars. Each class of formal grammar that is on higher level in the hierarchy is more powerful (it can describe as much as classes of grammars on lower levels + something more).

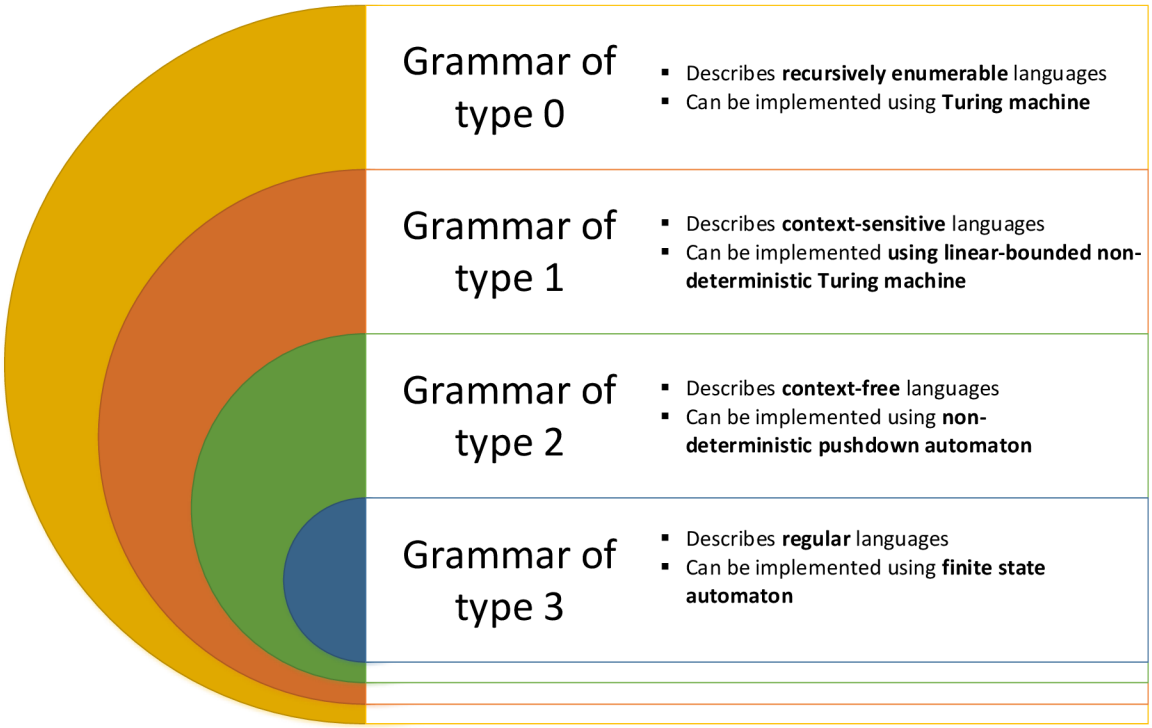

Figure B.1: Illustration of Chomsky hierarchy of formal grammar classes.

We will not go any deeper into rigorous aspects of formal grammars in this chapter. We will stay on a practical level and for each language class (family), we will focus on a real-world usage.

B.1.1 Regular Languages

Regular languages are the simplest languages generated by type 3 grammars in Chomsky hierarchy. These languages can be described using regular grammars (grammars of type 3), regular expressions or finite state automata. Each of these representations is equal in the sense that they can describe languages of the same complexity.

In practice, we often use regular languages and their equal representations for following tasks:

- **"grep" like functionality** – grep is a command-line program that takes as an input some regular expression and plain-text file, and as a result returns lines that match the regular expression. Such functionality can be often seen in advanced text editors like Notepad++ or Visual Studio Code.

- **User input validation** – programmers often have to deal with user inputs. Therefore, they have to make sure that a user input is like they expect and their program is able to process it. For example, checking whether a user wrote its email address or phone number correctly is one of the most common use cases, and many high-level programming languages have already libraries that allow programmers to easily match regular expressions and text inputs.

- **Tokenization** – is a process that takes a text stream as its input and splits it into tokes (words). The result sequence of tokens can be later used for parsing and building a parse tree and AST. In this thesis, a tokenization process is internally done by Xtext (and many other similar tools).

  - Example of token (terminal defined in Xtext grammar) terminal variableID: `('a'..'z') ('a'..'z'|'0'..'9')*`;

- **Describing states and transition between states in general** – a finite-state automaton represents very simple model that can describe states and transitions between them. It is useful in case when we want to describe behavior of some simple device (coffee machine) or "state dependent[1]" network communication protocol.

There are couple of other properties that can be derived from the theory of regular grammars and finite state automata.

The first finding is that every finite language is regular. The second one is that regular grammars and regular expression can be programmatically converted into non-deterministic finite-state automaton (NFA), and this model can be also programmatically transformed to deterministic finite-state automaton[2] (DFA). It is important to realize that such transformation (NFA to DFA) may lead to a DFA, which is in the number of states, exponentially larger than the original NFA. In order to reduce the number of states of DFA, we can use a process called DFA minimization. This process creates a new DFA with a minimum number of states. One of the fastest know algorithms of minimization is *Hopcroft's* algorithm.

But even with DFA minimization process, we cannot always reduce as many states as we would like to. The solution to this problem may be Quantum finite automata (QFA). Such automata can be, for certain languages, exponentially smaller than corresponding classical automata. Surprisingly, there are also languages for which QFA have exponentially more

[1] The number of states in such case has to be finite.

[2] NFA and DFA are equally powerful.

states than corresponding classical automata. So using QFA cannot solve every single problem we have with classical automata, but it may help a lot in some cases (assuming that we have a real quantum computer at our disposal).

B.1.2 Context-free Languages

Context-free languages are languages generated by type 2 grammars (context-free grammars, CFGs) in Chomsky hierarchy. The set of all these languages is superset of all regular languages, and it is equal to the set of languages accepted by pushdown automata[3]. This means that we can create some pushdown automaton for every single context-free language.

From a practical point of view, context-free grammars and their languages are heavily used for problems related to real, high-level programming languages. Here are some examples of what CFGs are capable of:

- **"Bracket matching"** – this may not sound important, but the ability to "match/-count brackets" is essential for context-free grammars. Practically, this is what makes CFGs different form regular grammars.
  - Let us have the following language: $L = \{a^n b^n | n \geq 1\}$. And now imagine that a stands for (and b stands for). Such language covers cases like () or ((())). This is something that regular grammars and finite state automata cannot do. The reason (simplified explanation) is that finite state automata have finite number of states (as the name already suggests) and they cannot remember unlimited number of brackets.
  - Pushdown automata have on the other hand an unlimited stack that can be used to remember how many brackets were already detected. So accepting languages like $L = \{a^n b^n | n \geq 1\}$ or even $L = \{a^n b^m c^m d^n | m, n \geq 1\}$ is possible.
  - Contradictory to the previous examples, CFGs/pushdown automata cannot deal with languages similar to $L = \{a^n b^n c^n | n \geq 1\}$. The non-formal explanation of this is that we count a characters on the stack (we push each a we read on the stack) and then for each b character, we remove one a from the stack. When the stack is empty, we know that the number of a characters was equal to the number of b characters. But at this point, we have lost the information of how many a and b characters were read before, so now we cannot count c characters. In other words, the memory of context-free languages is limited.
- **Programming languages parsing** – many modern, high-level programming languages use constructs that can be easily expressed by a deterministic CFG. For example, brackets in expressions: $x/(4 * (x - 3))$.
  - The advantage is that deterministic CFLs are accepted by deterministic pushdown automata, which are relatively fast on modern computers. The disadvantage is that pushdown automata are not capable of remembering the presence of a language construct over an input of arbitrary length. This essentially means that context-free grammars cannot be used for programming languages that need to have variables declared before they are used.

[3] A pushdown automaton is pretty much (non-formally) a finite-state automaton with an unlimited stack.

- In practice, we overcome such limitation by constructing a CFG that accepts a superset of our language (it accepts also source code that may not be valid) and then we use a semantic analysis to verify that the source is correct.

B.1.3 Context-sensitive Languages

Context-sensitive languages are languages generated by type 1 grammars (context-sensitive grammars, CSGs) in the Chomsky hierarchy. CSGs are more powerful (generic) than CFGs because they can generate more languages than CFGs (every context-free language can be generated by some context-sensitive grammar).

The computational model for context-sensitive languages is a linear bounded automaton (LBA). LBA is essentially a non-deterministic Turing machine with a limited length[4] of tape and it reflects more accurately the existing computers with limited memory size.

B.1.4 Recursively Enumerable Languages

The most powerful type of formal languages are recursively enumerable languages (abbreviated as REL; generated by grammars of type 0 in Chomsky hierarchy). There exist several formal definitions of what is a REL, but for the purpose of this thesis, we will stay on the practical level.

So practically, a REL is a formal language for which there exists a Turing machine. The Truing machine will always accept an arbitrary string from the language. Contrary to that, if the string is not from the language, the machine may loop/run forever (this is called the Halting problem).

In everyday life, we assume that Turing machine is an abstract mathematical model for modern computers. The only noticeable difference between Turing machine and computers is that Turing machine has an unlimited tape, whereas computers have real-world limitations and the amount of available memory is finite.

Another concept related to REL and Turing machine is so called Turing completeness. We say that a programming language is "Turing complete", if it can be used to simulate an arbitrary Turing machine with one tape. In real software development, probably every general-purpose programming language is Turing-complete, and it was even proven that very simple languages like *Brainfuck* and *Ook!* (with only 8 commands) are Turing-complete languages (there also exists *one instruction set computer*).

B.2 Classification of Real-World Programming Languages

IT is a very wide field that affects work and lives of billions of people, spreads across many industries and the development accelerates every year. Because of the diversity of IT usage and increased demand for IT solutions and their complexity, developers and programmers need better tools that allow them to express their thoughts and intentions easily and in a form that computers can understand. One set of such tools are programming languages. Languages, which are usually built on top of the formal bases mentioned in the previous chapter, and they use other concepts like lambda calculus or object-oriented design.

[4]The length of a LBA tape is $k * n$ where n represents the length of the input and k is some constant number associated with the LBA.

Currently, there are hundreds of noteworthy programming languages and thousands of programming languages that are not widely known[5]. In order to see, what are the main similarities and differences between these languages, we put them into groups (see Table B.1), depending on characteristics, features, level of abstraction, paradigms, etc. It is also common that a programming language is in multiple groups at the same time.

Table B.1: Types of programming languages.

Array lang.	Assembly lang.	Authoring lang.
Constraint programming	Command line interface lang.	Compiled lang.
Concurrent lang.	Curly-bracket lang.	Dataflow lang.
Data-oriented lang.	Data-structured lang.	Decision table lang.
Declarative lang.	Embeddable lang.	Educational lang.
Esoteric lang.	Extension lang.	Fourth-generation lang.
Functional lang.	Hardware description lang.	Imperative lang.
Interactive mode lang.	Interpreted lang.	Iterative lang.
List-based lang. – LISPs	Little lang.	Logic-based lang.
Machine lang.	Macro lang.	Metaprogramming lang.
Multi-paradigm lang.	Numerical analysis	Non-English-based lang.
Object-oriented class-based lang.	Object-oriented prototype-based lang.	Off-side rule lang.
Procedural lang.	Reflective lang.	Rule-based lang.
Scripting lang.	Stack-based lang.	Synchronous lang.
Syntax handling lang.	Transformation lang.	Visual lang.
Wirth lang.	XML-based lang.	

For example, C# can be put into following groups (the total number of such groups[6] is at least 50):

- **Compiled language** – C# is first compiled to Common Intermediate Language (aka bytecode), and then JIT[7] compiler compiles it into a native code that can be executed by an underlying hardware.
- **Curly-brackets language** – C# design is greatly inspired by C++ and Java and one of the things it takes from these languages is the "brackets style" (another commonly used way how to create blocks of code is indentation, which can be found in Python, Haskell and F#).
- **Imperative language** – statements in C# can change a program state (variables are mutable by default).

[5]The exact number of programming languages is unknown, because many of them are not public and some of the known languages are considered as dialects of other existing languages.

[6]The list of programming languages types was taken from: https://en.wikipedia.org/wiki/List_of-_programming_languages_by_type. Please note, that no official classification scheme for programming languages exists. This list is the most comprehensive source I was able to find.

[7]JIT is just-in-time compilation, a process of compilation into a native code that takes place during runtime. Such approach allows a VM to perform adaptive optimization (for example, dynamic recompilation) offers performance that may in some cases outmatch performance of C/C++ code.

- **Impure functional language** – as already mentioned, C# was originally inspired by C++ and Java, which are imperative languages, but the next major versions were later inspired by ML, Haskell and F# (functional languages).
- **Iterative language** – generators are in C# since version 2.0 (`yield` keyword).
- **Multi-paradigm language** – C# combines multiple paradigms (imperative, functional, object-oriented, declarative and generic).
- **Object-oriented class-based language** – OO paradigm is in C# built around classes that define the structure and behavior of objects.
- **Procedural language** – subroutines are in C# represented by methods.
- **Reflective language** – C# can examine and modify its high-level behavior and structures at runtime, using concept of so called reflection.

It is important to realize, that not every group/class is formally or clearly defined, which makes the classification less strict. Therefore, it may happen that the classification of a programming language is changing over time. It is due the fact that widely used languages are usually evolving and new major versions of languages can bring significant features (for example Lambda expressions) that can put the languages into new groups (following the previous example, Java is considered an *impure functional language* since version 8).

Another possibility of how a classification of a programming language can be altered is when the meaning of a certain group/class is changed. For example, scripting languages are traditionally considered to be languages, whose main purpose is to call external programs with some arguments (for example, Bash). However, the meaning was shifted, and nowadays even language like F# is by some programmers considered in a certain sense as a scripting language[8].

B.2.1 Classification of CTRL Language

Similarly, to the previous example of C# classification, we can classify the WinCC OA related languages that the IDE supports.

The following classification belongs to CTRL language (for more information about CTRL language, see Section 3.2.2).

- **Interpreted language** – a source code written in CTRL is executed by the interpreter included in WinCC OA.
- **Curly-brackets language** – the basic syntax of CTRL is very similar to C language, which uses brackets.
- **Imperative language** – variables in CTRL are mutable by default and big part of the CTRL code written at CERN relies on modifying a state.
- **Extension language** – because it extends the functionality of WinCC OA.

[8] Dave Fancher shows in his book, „The Book of F#“ (ISBN-13: 978-1-59327-552-5, chapter 2), how F# can serve as a scripting language.

- **Scripting language** – CTRL language is considered by some SCD members as a scripting language due to its way how it is executed, memory management, `eval` function and dynamic-type checking.
- **Multi-paradigm language** – imperative and partially object-oriented.
- **Object-oriented (partially)** – object-oriented paradigm is in the case of CTRL at least arguable. It defies most of OOP definitions (CTRL language does not provide programmers any means to create an object or define a class), and yet it allows programmers to treat certain variables as objects (if a variable refers to an UI element/*graphical object*, we can call its methods and use its properties).
- **Procedural** – since CTRL is derived from C language, it contains libraries and subroutines in a form of functions.

As already discussed, a programming language can evolve and its classification may change. This is the case of CTRL. It originally did not support "UI objects" (object-oriented paradigm) and it may happen that potential future versions will shift the language towards different paradigm or add significant new features[9].

[9]Note that the CTRL classification in this thesis applies for the version shipped with WinCC OA 3.14.

Appendix C

Manual

- *Online IDE documentation* :
  https://wikis.web.cern.ch/wikis/display/EN/WinCC+OA+IDE
- *CTRL Plug-in - dev. documentation* :
  WinCCEclipseIDE/Editor/ch.cern.winccoa.ctl/Ctl_plugin_developer_doc.docx
- *Configuration Plug-in - dev. documentation* :
  WinCCEclipseIDE/Extra/ch.cern.winccoa.config/doc/
  Config_file_plugin_developer_doc.docx
- *Configuration Plug-in - user documentation* :
  WinCCEclipseIDE/Extra/ch.cern.winccoa.config/doc/
  Config_file_plugin_user_doc.docx
- *Configuration Template Plug-in - language specification* :
  WinCCEclipseIDE/Extra/ch.cern.winccoa.configTemplate/doc/
  Config_template_language_description.docx
- *Configuration Template Plug-in - dev. documentation* :
  WinCCEclipseIDE/Extra/ch.cern.winccoa.configTemplate/doc/
  Config_template_plugin_developer_doc.docx

Appendix D

Notes

- EN-ICE (*Engineering Department–Industrial Controls & Engineering*) group at CERN was restructured and SCD section was moved to BE-ICS (*Beams Department–Industrial Controls and Safety*).