SCL-T
Template programming for Siemens SCL

Version of August 19, 2019

Jeffrey Goderie

SCL-T
Template programming for Siemens SCL

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jeffrey Goderie
born in Steenbergen, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

CERN
Geneva, Switzerland
www.cern.ch

SCL-T
Template programming for Siemens SCL

Author: Jeffrey Goderie
Student id: 4006232
Email: c.j.m.goderie@student.tudelft.nl

Abstract

Programmable Logic Controllers (PLCs) are used to control large scale systems with thousands of I/O devices. Writing and maintaining the logic for each of these is a cumbersome task, which is well suited to be abstracted through templating. For this purpose, CERN developed the Unicos Application Builder (UAB). While UAB is successful at templating, it provides no guarantees over the validity of the outcome, leading to erroneous generated code. This is where SCL-T comes in. It builds on the foundation of UAB to facilitate meta-programming for Siemens' SCL. Unlike its predecessor, it guarantees syntactic correctness and also draw conclusions regarding the semantic validity of the generated code. Its architecture has been designed in such a way that support for other PLC languages can be added using the same meta-language, reducing the cost of a having a meta-programming language tailored for a specific PLC language.

Thesis Committee:

Chair: Prof. Dr. Eelco Visser
Committee Member: Dr. Casper Bach Poulsen
Committee Member: Dr. Christoph Lofi

Preface

First and foremost, I would like to thank Eelco Visser for guiding me through the process of writing this thesis. Additionally, I would like to extend my gratitude to him, Guido Wachsmuth, and CERN (in particular Ivan Prieto Barreiro and the PLC section) for giving me the opportunity to take on this project on premise at CERN.

Jeffrey Goderie
Delft, the Netherlands
August 19, 2019

Contents

Preface iii

Contents v

List of Figures vii

1 Introduction 1

2 Analysis of UAB 3
- 2.1 Introduction to UAB . . . 3
- 2.2 UAB's limitations . . . 7
- 2.3 Evaluation . . . 18

3 Meta-programming approaches 21
- 3.1 Introduction to meta-programming . . . 21
- 3.2 Separation of languages . . . 23
- 3.3 Use time . . . 26
- 3.4 Separation of static and dynamic code . . . 27
- 3.5 Type of meta language . . . 28
- 3.6 Hygiene . . . 29
- 3.7 Type Validation in Meta-programming . . . 31
- 3.8 Evaluation . . . 32

4 Language Design 35
- 4.1 Introduction to SCL-T . . . 35
- 4.2 SCL-T by Example . . . 36
- 4.3 Data Types . . . 42
- 4.4 Functions . . . 49
- 4.5 Splicing . . . 50

5 Language Implementation 53

5.1 Architecture 53
5.2 Type System 55
5.3 Object language 56
5.4 Meta-Language 61
5.5 Mixin Language 71

6 Evaluation **77**
6.1 Evaluation of SCL-T 77
6.2 Prototype Reception at CERN 79
6.3 Discussion 80

7 Conclusions **83**

Bibliography **85**

A Glossary **91**

B Simple Template **93**

C Template in Old and New Syntax **95**

D Meta-Language Functions and Methods **99**

E Meta-Language Syntax **103**

List of Figures

2.1 UAB's architecture 6
2.2 UAB's operational flow 7
2.3 Example of a faulty but valid specification file 9
2.4 Example of UAB output 9
2.5 Source code 10
2.6 Output from running Figure 2.5 in UAB 10
2.7 Mixed indentation 11
2.8 Missing keyword 12
2.9 Variable lifting 12
2.10 Arbitrary code constructs 13
2.11 List comprehension 14
2.12 Segmentation 15
2.13 Representation of the ideal workflow. 17
2.14 More realistic representation of the workflow. 17

3.1 Homogeneous, string-based meta-programming 24
3.2 Heterogeneous, string-based meta-programming 24
3.3 Homogeneous meta-programming using Python3's internal mechanisms 25
3.4 Run-time evaluation 26
3.5 Compile-time evaluation 27
3.6 Hygiene violation 29
3.7 Precedence violation 30

4.1 An example SCL-T template, based on an actual production code file. 37
4.2 Excerpt of Figure 4.1: lines 12 to 15. 38
4.3 Excerpt of Figure 4.1: lines 16 to 23. 39
4.4 Excerpt of Figure 4.1: line 25. 40
4.5 Exploded version of the statement in Figure 4.4 41
4.6 Mutable objects and copy operations 43
4.7 Differences between lists and sets 44

4.8 Methods to iterate over dictionaries . . . 45
4.9 Nested pairs . . . 46
4.10 Usage of <Var(T)> . . . 47
4.11 Statement concatenation through splicing . . . 48
4.12 UNKNOWN type propagation . . . 49
4.13 Programmatic expression formation . . . 49
4.14 Valid and invalid function overloading . . . 50
4.15 Mappings from meta-level objects to object-level types . . . 51
4.16 Arbitrary code constructs . . . 51
4.17 Example of 'generateIfElse' function . . . 52
4.18 Output of 'generateIfElse' function in Figure 4.17 . . . 52

5.1 SCL-T's language composition. The arrows indicate the direction of composition. . . 54
5.2 Meta-language reuse among various PLC languages . . . 54
5.3 Language composition based on varying the meta-language or the mixin-language . 55
5.4 Format and meaning of the various typing rules used in this chapter. . . . 56
5.5 Type rules for variable declaration . . . 58
5.6 Type rules for units . . . 59
5.7 Type rules for function (block) calls . . . 61
5.8 Type rules related to Structs . . . 62
5.9 Noteworthy Meta-Language syntax rules . . . 63
5.10 Subtyping (<:) relationships for the meta-language . . . 64
5.11 Type rules to iterate over statements . . . 65
5.12 Type rule for control statements . . . 66
5.13 Type rules for variable declaration . . . 66
5.14 Type rules for variable assignment . . . 67
5.15 Type rule for function declaration . . . 67
5.16 Type rules for return statements . . . 68
5.17 Type rule for function calls . . . 68
5.18 Type rule for the 'len(e)' standard function . . . 68
5.19 Type rules for the '.add(...)' method . . . 69
5.20 Type rules for the '.add(...)' method . . . 70
5.21 Field Validation for Compound Devices. 'AutoOff' is not a field in T2. . . . 71
5.22 Mixin-Language syntax additions . . . 72
5.23 Type rules for quoting operations . . . 73
5.24 Type rules for splicing operations . . . 74
5.25 Type mappings from meta-level objects to object-level types . . . 74
5.26 Type rule for expression splicing . . . 75
5.27 Mapping from meta-variable to object-level AST . . . 76

C.1 New Syntax . . . 95
C.2 Old Syntax . . . 96
C.3 Missing keyword . . . 96
C.4 Missing Keyword . . . 97

E.1 Meta-Language: Start Symbols 103
E.2 Meta-Language: Types 103
E.3 Meta-Language: Statements 104
E.4 Meta-Language: Expressions 105
E.5 Meta-Language: Functions 105

Chapter 1

Introduction

Programmable logic controllers have obtained their foothold in the domain of industrial automation because of their reliability. These real-time systems have proven to be resilient against harsh conditions, like extreme temperatures and electric noise.

This resilience, combined with their overall ease of programming, is the reason why CERN (Conseil Europen pour la Recherche Nuclaire) utilises them in large quantities. While they are sometimes used in small setups, their main application is in immense control systems. These systems consist of various PLCs combined with thousands of I/O devices.

The PLCs are loaded with logic files that act on the input/output of each I/O device. Some systems have hundreds of logic files, which all need to be created and maintained. However, not all of these files are truly unique. Many control setups have various sets of identical I/O devices in various places, and therefore have logic files that are similar. These files vary mostly in identifiers, parameters or minor logic sections, but have a common ground. Despite the similarities, creating and maintaining the code proved to be a cumbersome and error-prone task. Developers started looking for ways to make their lives easier. Rather than writing each file by hand, they started to create them by duplicating similar implementations. This way of working proved to be problematic, as it was still easy to make a mistake, and validating cost a lot of time.

To effectively reduce the cost and effort associated with large PLC software projects at CERN, the Unicos Application Builder (UAB) was designed. UAB offered a templating functionality in Jython. Its purpose was to prevent developers from having to resort to the labour-intensive, error-prone 'copy-paste' workflow that they had adopted.

The templating facility allowed developers to write a single logic file, and combine it with a specification file, to generate many different instances of the same logic. This allowed entire systems to be reduced to a maintainable amount of logic templates, and a single specification file. This also meant that in case of an adaptation, the developer no longer needed to touch up numerous files, but instead simply had to change the template, and regenerate the code.

All in all, UAB proved rather successful in reducing the effort and cost of PLC programming. Throughout its existence, it gained more foothold at CERN and outside. While acknowledging its potential, there were users that refrained from using the tool due to its somewhat poorly designed meta-language and its lack of features. The suggestion that something could be improved was strengthened by the fact that source files and generated code started to diverge. Inquiring

about this revealed that users experienced issues with UAB. In particular, they felt it was more effective to fix errors in the generated code than it was to do so in the source.

This is where this document comes in. To allow the UAB framework to reach its full potential, SCL-T has been created. SCL-T is a new meta-programming language for UAB that provides meta capabilities for the Siemens SCL language. Rather than having the same meta-programming language for all PLC languages, SCL-T is tailored only for SCL. This opens up the possibility of providing PLC specific features.

SCL-T is not meant to completely overhaul the existing UAB framework. Instead it addresses some of the underlying meta-language's most apparent weaknesses in an as unobtrusive manner as possible. As such, the design for SCL-T combines the strengths of the original meta-language and the insights gained from a thorough analysis of its weaknesses.

Unlike its predecessor, SCL-T allows developers to focus on writing PLC code, rather than writing meta-code. Templates consist of PLC code enhanced with meta-level code. Developers can use concrete SCL syntax to generate object-level code. Due to the changed format of the templates, SCL-T can guarantee full syntactic correctness of the resulting SCL code. This alone removes a large number of commonly made errors.

Besides syntactic validity of the generated code, SCL-T also ensures syntactic and semantic validity of the template code. Since the meta-stage is sound, SCL-T is able to provide a significant degree of cross-stage type safety. As a result, the generated code is guaranteed to be free of certain semantic errors.

A large degree of the semantic guarantees on the resulting code come from the addition of type checking UAB's static information sources, e.g. its specification files. SCL-T does not just allow users to query device instances and their respective attributes, it also ensures that the queries and outcomes are well-typed.

Since UAB was made to support multiple PLC languages, reducing support to merely SCL would be a huge step backwards. To counter this, UAB's architecture has been addressed as well. Where the original architecture was designed with scalability and flexibility in mind, it's implementation was thin. The new architecture builds on the original design, implementing the new language based on a combination of a meta-language, an object language, and a mixin-language. The meta-language is reusable between different PLC languages. Adding support for additional languages only requires the object language and the mixin language to be designed. As a result, PLC developers do not have to learn a new meta-language for each PLC they want to program using UAB.

Before introducing the details of SCL-T, a thorough analysis of UAB's current approach to meta-programming will be detailed in Chapter 2. It looks into the how, what, and why of the UAB framework to determine where the current approach can be improved upon. Chapter 3 summarises relevant, existing literature concerning code generation. It deals with various types of meta-programming, as well as type safety. Using the obtained information it is possible to design SCL-T, whose features are discussed in Chapter 4. Chapter 5 covers the underlying implementation. In order to see how SCL-T holds it own, it is evaluated against the set goals. Additionally, SCL-T reception by the users is covered. Both are done in Chapter 6. This chapter is also used to identify potential follow-up research topics, and weak points in the study leading up to SCL-T. Lastly, Chapter 7 provides an overall conclusion to this study.

Chapter 2

Analysis of UAB

Unicos Application Builder (UAB) is a collection of tools for the programming of Programmable Logic Controllers (PLCs). UAB has been developed at CERN. It provides a templating facility for various PLC languages (e.g. Siemens SCL, Schneider ST) that allows users to easily generate many instances of the same logic based on a single template and some input parameters.

Having been used for several years, UAB's characteristics have become apparent. Using the insights in its strengths and weaknesses, it is possible to create an improved version of UAB meta programming features.

2.1 Introduction to UAB

Before diving into the internals of UAB's meta-programming language, it is important to understand what UAB is, and how it came to be. For this, one needs to look at what problems it is meant to solve, and what its architecture and process look like. This information is crucial in understanding exactly why the current approach is sub-optimal.

First, the origin story of UAB will be told, in which it becomes apparent what problem UAB is trying to solve. In this section it also will be made clear what design constraints UAB has, and what the sub-goals of the framework are. This information is important when redesigning any component of the framework, like the meta-programming language. Following the background of UAB is an analysis of the architecture of the framework. Having an understanding of what the internals look like is helpful in determining the impact of changes, but could also reveal additional issues with the implementation.

2.1.1 Origin of UAB

At CERN experiments are conducted at enormous scales. Behind these experiments are Programmable Logic Controllers. PLCs manage the processes to ensure the desired functionality. Their main selling point is that they are highly reliable, even in environments that are not exactly electronics-friendly, e.g. exposure to dust, radioactivity, or extreme temperatures. The controllers are provided with information originating from various sources, most of which are sensors and timers. Based on the information, they are in charge of controlling a number of

components, e.g. valves, switches. Each PLC can be in charge of managing hundreds of I/O devices[1]. While there are endless possibilities in which a controller can be combined with input and output devices, it is often the case that within a single system there are multiple identical device setups. They serve the same purpose, but they might be used at 2 different stages of the automation process, meaning that their precise workings might differ. However, since the general functionality of the components is similar, their corresponding code (i.e. device specification files, logic files) is most likely also similar.

Initially this meant that a developers had to create multiple nearly identical files when creating software for the complete chain. In most cases the files varied only in identifiers or parameter values, while the actual functionality was identical. However, since developers would often focus on a single PLC at a time, not taking into account the rest of the chain, the same logic was written multiple times, potentially by different developers. Writing each file by hand proved to be a tedious and error-prone task, regardless of whether it was done by a novice or an experienced person. As Wilson et al. [60] said, "sooner or later even the most careful person will lose focus and make a mistake". Besides the inevitable mistakes that might occur, having a fully manual chain brings along problems regarding maintainability, consistency, and comprehensibility. Since all files are rewritten in full, it is difficult to be consistent across the various files. As a result, changes to be made cannot be copy-pasted around, as the code might not be the same throughout all the files. Additionally, the inconsistencies make it harder to judge the functionality of the whole system, as it is not directly obvious that the different logic files provide the same functionality.

Realising that this approach was all but efficient, the developers adopted the basic mindset of making the computer work for them. They started creating the required files for single component, and duplicated it for each of the similar instances. Afterwards the varying identifiers and parameters were adapted for the derived instances, for example through find-and-replace. While this was a huge time saver, this approach also proved to be prone to mistakes and still required a significant amount of manual labour.

The main reason for this approach being erroneous was the developer being in charge of adapting each file with the correct identifiers and parameters. It sometimes happened that a single value was forgotten, or erroneously replaced, resulting in invalid code. Syntactic or semantic errors were often caught at compile time or immediately at run-time. However, wrongly replaced values and identifiers could result in logic files that were both syntactically and semantically valid, but could result in faulty behaviour, which could remain undiscovered until that specific faulty scenario was triggered in a physical setup.

Additionally, by duplicating a file, errors could be propagated. This meant that if the source file contained a mistake, it was often necessary to retouch all derived files, which is a costly action, which could even result in new errors and diverging logic. This is exactly why developers (should) adhere to the DRY (Don't Repeat Yourself) principle [27] whenever possible.

While copying and replacing code was no more error-prone and generally more efficient than writing each file by hand, it still turned out to be an inefficient approach to obtaining files that share common elements. Acknowledging that the human factor in the duplication process was a common source of mistakes, CERN decides to come up with an automated approach. The

[1] http://unicos.web.cern.ch/applications

initial attempt to code generation was based on Microsoft Access, but it did not meet the technical requirements [58]. Taking into account the lessons learned from the prototype, a second iteration, focused on scalability and flexibility, was designed. This version was coined Unicos Application Builder (UAB).

2.1.2 Goals of UAB

The goal of UAB is not to remove all human involvement with writing PLC code, but to facilitate the development of abstracted control system applications, effectively removing the repetitive nature of the task. The aim is to reduce the time involved in developing, configuring and commissioning control code, and at the same time produce well-structured code.

Besides its main goal, the creators of UAB came up with a list of design constraints that UAB has to meet. These requirements are:

- Flexibility
- Scalability
- User-friendly
- Versioning management

Even though UAB is meant for a niche market, namely PLCs, its diverse use-cases require it to be flexible. There is no golden solution which fills in all the needs for the various applications. Therefore the framework should support the addition of user-defined templates, functions, and semantic checks.

UAB is supposed to be scalable, meaning that new PLC languages, objects, and components can be added at any time. Scalability is important to guarantee the functionality of the tool in the future, where CERN might use different PLC manufacturers or the manufacturers might change their languages. In a way UAB's scalability also contributes to its desire to be flexible.

The user-friendliness criteria of UAB is related to its GUI. It should prompt the user for the mandatory data of the application, validate the user data and trigger the execution of the selected plug-ins with the specified parameters.

Version management is important to guarantee backwards compatibility, which is a crucial aspect of code generation for many different systems, some of which are legacy. Having a mechanism to manage the evolution of software, in particular the evolution of device interfaces, is crucial in being backwards compatible.

In general, these criteria either concern the whole UAB framework, or a component other than the meta-programming language. However, it makes sense to also project them onto the meta-programming language. This means that the language itself should be flexible, scalable, user-friendly, and support versioning, where the latter involves both the language itself, as well as the device definitions used within the language.

2.1.3 Architecture of UAB

Armed with a basic introduction to what problem(s) UAB is trying to solve, the architecture can be detailed and assessed. Aside from explaining what the architecture looks like, it will also be addressed how the framework operates.

The first publicly available mention of UAB's architecture originates from 2007 [16]. Its corresponding diagram is visible in Figure 2.1.

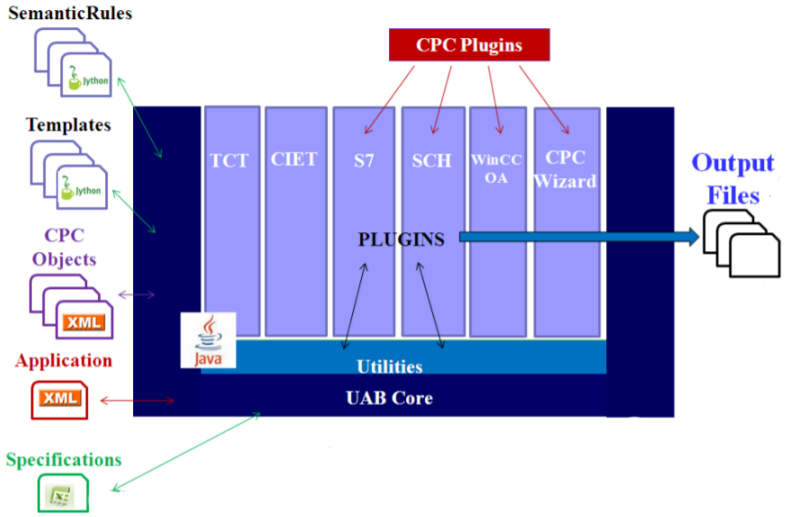

Figure 2.1: UAB's architecture

There are three main groups that can be identified. On the left there is the input, which consist of various different files, each with their own purpose. On the right are the output files, which are the result of running the UAB process. The largest group comprises the UAB application itself. Architecturally, this group is also the most interesting.

Moving top-down, the plug-ins are up first. These plug-ins each have their own set of functionalities and rules. They are independent of each other, and generally have only a single target language. Below this layer of language-specific components is a set of utilities which are plug-in independent. These provide methods and functions that are useful regardless of the target system. Surrounding each component of the application is the UAB Core. As the name suggests, this consists of the core elements of UAB, and is the backbone of the entire application.

The architecture shows how the different groups relate to each other, but provides no real indication of how they interact. This is also important to understand in order to be able to judge how well UAB realises the set goals. It also adds insight that an architectural overview alone cannot provide.

Much like the architecture, the operational flow can be divided in a series of sections. The flow is displayed in Figure 2.2, and consists of 'generation', 'validation', 'exportation', and 'importation, compilation and execution'. The latter can also be fittingly named 'external'.

Aside from these phases, there are also things that happen when UAB is started. The CPC Objects, the Application XML and the CPC Plugins are loaded on start-up through a bootstrap-

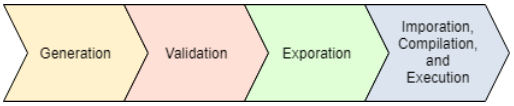

Figure 2.2: UAB's operational flow

ping mechanism. Bootstrapping allows UAB to support various device generations that might otherwise cause interference. Having loaded these components, UAB is started, and the operational flow can commence.

The first operation is the 'generation' phase. At the base of this phase are the user-created templates, and the static information files. These are used to generate the eventual PLC code. In general, most of the functionalities used at this stage are part of the core component. It is possible for the developer to have utilised some of the methods available in the language specific plugins, but this happens only sporadically.

The language-specific components become more apparent in what UAB considers the 'validation' phase. Part of this phase overlaps with the generation, as a scripting error (i.e. a Python error) will result in generation-time errors. After generation a set of validation and verification checks are performed. Some of these are PLC-agnostic, but most of the available checks are user-defined for a specific language. If an error occurs at generation-time, or during the validation phase, the generation cycle is flagged unsuccessful.

In case there are no errors, the code will then be exported to an user-specified location. This is where the user interaction with UAB stops. The user now to switch to the PLC manufacturer's own tooling, hence the name 'external'. The next step is to use this tool to import the project. The project is then compiled, during which the whole project is syntactically and semantically validated. If the compilation step is successful, the compiled programs are loaded onto the appropriate devices, and are ready to be put to use.

Having obtained insight in both the architecture and the operational flow, it is safe to say that these do not inhibit UAB in achieving its goals. The overall architecture is well suited for flexibility and scalability, but simply is not used to its fullest extent. The operational flow revealed only that validation of the generated code is largely put off till the end of the cycle. This does not necessarily have to be harmful, depending on what errors can be caught in the earlier validation stage. In order to determine how to better achieve its goals, one has to step away from the overviews, and dive into UAB. Or more specifically, into its weaknesses.

2.2 UAB's limitations

"Optimise Software Only after It Works" [60] is a great description of where UAB is currently at. While it has stood the test of time, and has proven itself to be useful in a full-scale production environment, its flaws have been exposed. The tool lacks some key features that would greatly increase its usability, and some of the ones that are present are suboptimal.

In order to see how the current situation can be improved upon, it is crucial to investigate exactly where the current approach runs short. In the following sections each of UAB's limitations are detailed. No additional attention will be given to its strengths, as the intention is to improve

UAB where-ever possible, without making unnecessary changes to the current implementation. This means that it is acceptable to adapt the tool if, and only if, it provides significant benefits, or if the impact on the final product is negligible.

The scope of the analysis will be limited to the meta-programming language used to write logic files and the information sources of UAB. These are the components the user has to interact with the most, and as such the impact of an issue in one of these is more significant. As a result, fixing an issue in these areas will yield the largest reward.

As detailed in section 2.1.3, UAB's meta-programming language is suited for multiple PLC languages. To concretise the issues with the meta-programming language, they will be illustrated using Siemens SCL as the target PLC. The reason for this is that Siemens is CERN's largest PLC manufacturer, but it could have been any of the supported PLC languages.

The current meta-programming facility will be evaluated on its syntax, its implementation, its validation, and its source of static information. The order in which these components are addressed is of no particular importance.

2.2.1 Static information

As described in section 2.1.3, the UAB framework uses a set of parameters which are fed to the generator as a pair of XML files. These are used to generate different logic files from the same template. The configuration file, which contains a definition for the various device types, is provided to the user. The other XML file contains the device instance specifications.

Using XML for parameter and specification files is not a novel approach. It is put into practice in distributed systems[20], industrial engineering[25], data processing[26], and other fields.

XML has quite some advantages[38]. One of these is its extensibility, as suggested by the name XML, which stands for 'Extensible Markup Language'. Since the structure of XML is not fixed, it can be adapted at any point in time without having to revise the associated parser. This also allows for the creation of hierarchical structures[29], making it easier to group components. Additionally, it is human and machine readable, and application-neutral, making it well-suited for a variety of use cases.

The main downside to XML is its verbosity [14]. Swarmed with tags, the format is difficult to read, and it is easy to make a mistake which invalidates the whole file. Expecting users to edit an XML file thousands of lines long is generally not acceptable. That's why the team behind UAB created an XML that is properly editable from within Microsoft Excel. The visual editor of Excel reduces the chances of users creating invalid specification files. Whereas XML requires them to write both the correct structure and values, in Excel only the values suffice.

While this makes the specification files more user-friendly, the underlying XML is not used to its fullest potential. The absence of an XML schema (e.g. XSD [44]) still allows the user to define invalid specification files. The validation that is currently performed on the files is rather simple. Some values are to be selected from a drop-down menu, and are therefore always valid, but generally the only enforced requirement on each cell is that its content needs to be textual (i.e. no formula, image). This check is very liberal in the sense that it accepts numeric values where the parameter clearly requires an alphabetic input, and vice versa.

DeviceIdentification	FEDeviceParameters	
Name	Range Min	Range Max
SENSOR_1	0	100
SENSOR_2	0	ABC
SENSOR_3	0	200

Figure 2.3: Example of a faulty but valid specification file

An example of this is given in Figure 2.3. Parameter 'Range Max' is supposed to be numerical, as the name suggests. With the current validation the user can assign value "ABC" to this field, even though this is not a valid numerical value. The outcome of an arbitrary use case of this specific example is displayed in Figure 2.4.

```
IF (SENSOR_1.IN1 < 0 OR SENSOR_1.IN1 > 100) THEN
    //trigger alarm for SENSOR_1
END_IF
IF (SENSOR_2.IN1 < 0 OR SENSOR_2.IN1 > ABC) THEN
    //trigger alarm for SENSOR_2
END_IF
IF (SENSOR_3.IN1 < 0 OR SENSOR_3.IN1 > 200) THEN
    //trigger alarm for SENSOR_3
END_IF
```

Figure 2.4: Example of UAB output

Such mistakes will generally not break the generation, meaning that it will be detected during compilation. As discussed earlier fixing errors at that stage is more costly. If variable ABC is non-existent, or of a non-numeric type, a compilation error will occur. If the variable exists and is of the right type, something unforeseen will most likely happen at run-time. No valid use case can be thought of where a specification file should refer to a variable that may or may not exist in the template. If a template developer needs to utilise such a reference, they can directly use the local variable, rather than referring to the specification file.

The developers behind UAB noticed that XSD has its limitations with respect to the intended purpose [4]. While their point is certainly justified, it can be debated whether an XSD that validates some elements, is better or worse than a complete absence of validation.

Aside from validation issues, there is very little negative to say about the static information sources. They fit their intended purposes, and have been purposefully made more user-friendly than their datastructure originally is.

2.2.2 Syntax

Most problems with UAB are not related to the static information, but with the actual meta-programming language. The biggest problem of which is its lack of an appropriate syntax. The templates are Python programs are used to generate PLC code. The code to be generated is passed as a String parameter to a custom Python function called 'writeSiemensLogic'. Each

String can be annotated with \$ < *variable* > \$, where < *variable* > is part of the current Python heap. When running the template, the value of variable at the stage of execution where the code is generated, will be spliced into the String, after which it is printed to the output file. A 'Hello-World' example of this is shown in Figures 2.5 and 2.6, which display the source code and the generated code, respectively.

```
username = "World"
plugin.writeSiemensLogic('''
Hello $username$!
''')
username = "John"
plugin.writeSiemensLogic('''
Hello $username$!
''')
```

Figure 2.5: Source code

```
Hello World!
Hello John!
```

Figure 2.6: Output from running Figure 2.5 in UAB

In case of the simplest template, where each inserted value is a constant, the String of SCL code can be inserted by a single call to the Python functions, where the parameter is the annotated SCL program. The logic is fixed, and the variables are replaced by a simple find/replace mechanism, which could be done using a simple text editor. An example of such a template can be found in Appendix B. As can be seen, the impact of the boilerplate code is minimal, but so is the template's functionality.

In practice, UAB templates are much more complex than this, with dynamic logic and varying values. In these cases the resulting code is governed by the input parameters or the control flow of the templates. Generally this involves templates that consist of multiple segments of SCL code. Each segment is directly surrounded by boilerplate Python code (i.e. 'plugin.writeSiemensLogic'), and is separated from another segment by one or more Python statements. To understand the templates the developers have to read past the boilerplate code in order to only consider the relevant elements. However, in the process of mentally filtering it is possible to skip over significant code, especially when it is short and surrounded by multiple sections of wrapper code.

Normally Python's indentation rules, which will be discussed in more detail in Section 2.2.3, should help in determining the control flow and logic of a template. However, the syntax of the SCL code segments counters this indentation. If the developer intends to add indentation to the resulting output code, this indentation has to be included in the String. However, this indentation is absolute, and not relative to the Python indentation level. This means that if a single tab (or its 4 space equivalent) is desired, the developer will type a single tab, even if the template's indentation level at that point is already 4 tabs. An example of this can be seen in Figure 2.7. This clearly shows that the readability of the templates suffers heavily from the mixed indentation rules. This effect becomes even worse if developers do not capitalise PLC keywords, which is allowed since the languages are case-insensitive. The degree to which this affects readability can be limited by using an editor that highlights the various code elements, but this can only do so much. Additionally, this is a work-around for a poor design decision.

```
...
if condition_1:
    if condition_2:
        plugin.writeSiemensLogic('''
IF $DB_Global$.Pulse_Ack THEN
    $name$.AuRStart := TRUE;
END_IF;
        ''')
    else:
        plugin.writeSiemensLogic('''
IF $DB_Global$.Pulse_Ack THEN
    $name$.AuRStart := FALSE;
END_IF;
        ''')
...
```

Figure 2.7: Mixed indentation

Aside from harming readability, the segmentation of code can facilitate template errors by means of missing keywords. Since SCL is a block-structured language, each block of code (e.g. type declarations, statement blocks, control statements) is clearly indicated by a keyword at the beginning and at the end of each block. In case of segmented code, where the segmentation occurs inside a block, it is very easy to miss the closing keyword of the surrounding block. Especially when dealing with nested control statements one can forget to close the top level statement, as displayed in Figure 2.8, which is missing a final 'END_IF'. The more significant code that separates the opening keyword of a control statement from its closing one, the greater the chances of it being forgotten, especially if the closing keyword is utilised in one of the nested segments.

There is yet another issue with segmentation, which is its impact on validation. Section 2.2.4 will cover UAB's validation mechanics, but the bottom line is that due to the segmentation of PLC code it hardly possible to draw any conclusions on the validity of the template, both semantically and syntactically. This significantly increases the chances of the templates or generated PLC code being faulty.

Most of the syntax-related problems are caused by code segmentation, but the developers have expressed other grievances as well. It is only allowed to insert variables, which means that in order to splice in the result of a simple expression, the developer has to lift the expression into a variable, and splice that in, rather than being able to directly splice in the expression. An example of this can be seen in Figure 2.9.

Another syntax-related issue that came to light when investigating the existing templates, is the absence of any control on the format of the produced PLC code. There are no rules regarding what constructs can be used to generate PLC code, allowing developers to come up with any way to generate code that they feel is right. This can further harm the readability and re-usability of the templates, as indicated in Figure 2.10.

```
plugin.writeSiemensLogic('''
IF CONDITION_1 THEN
    # Common statements
    A := 1;
    B := 2;
''')
if condition_1:
    plugin.writeSiemensLogic('''
    IF CONDITION_2 THEN
        $name$.AuRStart := TRUE;
    END_IF;
    ''')
else:
    plugin.writeSiemensLogic('''
    IF CONDITION_3 THEN
        $name$.AuRStop := TRUE;
    END_IF;
    ''')
```

Figure 2.8: Missing keyword

```
# Current situation
for i in range(10):
    temp = i * 2
    plugin.writeSiemensLogic('''A_$i$ :=
$temp$;''')

# Desirable situation
for i in range(10):
    plugin.writeSiemensLogic('''A_$i$ :=
$i * 2$;''')
```

Figure 2.9: Variable lifting

All in all, the issues detailed above indicate that the current syntax is counter-intuitive, error-prone, and results in templates that are difficult to comprehend due to boilerplate code and poor formatting. In order to improve upon this, the focus of the templates should shift from writing Python code to writing SCL code, both for validation purposes and general comprehensibility.

In order to ease the transition between the old and the new situation, the decision was made to stay as close to the current approach as possible. The reason for this is that large adaptations to an established entity often fail as people are resistant to change. Through the process of mimicry, keeping a strong link between the old and the new, changes can be hidden, and the overall difficulty of introducing a new facility is reduced greatly [33]. For the new meta-language

```
# Wrong scenario (actual code example)
plugin.writeSiemensLogic('''BOOLEAN_VAR := ''')
for i in values:
    plugin.writeSiemensLogic('''$i$ OR ''')
plugin.writeSiemensLogic(''0;''')

# Better scenario
condition = " OR ".join(values.append(0));
plugin.writeSiemensLogic('''BOOLEAN_VAR := $condition$;''')
```

Figure 2.10: Arbitrary code constructs

this means that in order to get rid of the boilerplate code, the meta-language cannot be adapted drastically, so Python should not be replaced by a scripting language with a completely different look and feel.

2.2.3 Implementation

UAB is a Java program that interprets Python-based templates to generate PLC code. In case of SCL, the generated code is then imported into a native PLC editor, e.g. Siemens SCL's Step7. After importing the code, it can be compiled and loaded onto the PLC devices.

Python is well suited as a meta language due to its dynamic typing system, its terse syntax, and the ease of creating prototypes. In order to use these advantages with the existing UAB backend, the decision was made to implement the template facility using Jython. Jython is a Python implementation that runs on the JVM, making it possible to combine Java and Python code. This enabled the full use of Python's meta language capabilities, while at the same time removing the need to refactor the UAB backend.

At first glance it seems there are no significant downsides to the implementation of the templating facility. However, through extensive usage several problems have surfaced, indicating that improvements can be made in this area when creating a new templating language for SCL.

The implicit mixing of Java and Python code, though straightforward, resulted in an unforeseen issue. Since there is no direct mapping from Java classes to the corresponding Python classes, it was up to the developer to map one to the other. The most common example of this was mixing Java lists and Python lists, both of type List. Some of UAB's default functions returned a list in Java, that could not directly be used as a list in Jython. Helper classes were created to perform the mapping of the common cases, but to a novice user the reason why they should be used was experienced as confusing.

Since there are only a few Java classes which are utilised from within the templates, the occurrences of the aforementioned issue were scarce, limited to once or twice in each template. The majority of each template consisted of pure Python code, which revealed another potential flaw of the current approach. 'Potential' is an important nuance in this case, as opinions varied among developers, mostly governed by their proficiency in Python. Using a general purpose language as a meta language opens up the possibility to write powerful and complex code templates,

often in multiple ways. The possibility to create virtually any piece of code was unanimously accepted as an advantage, but novice developers indicated that they struggled to understand the templates created by their more experienced peers. Further investigation of the 'complex' templates revealed that they were not necessarily more complex than their counterparts, they simply contained some of Python's more advanced language constructs, like list comprehension and lambda functions.

In most cases the incomprehensible template code could be replaced by a slightly more verbose, but easier to understand version, as is displayed in Figure 2.11. This is a good indication that Python is too extensive for the intended purpose. Normally requiring a certain level of familiarity with a language is acceptable, but since the object language originates from a different domain (mechanical engineering), one in which general programming experience is not required, it is defendable that the templates should be as simple as possible.

```
#List comprehension
a = [f.strip() for f in feature.split('=',1)[-1].split(',')]

#Extended example
a = list()
for f in feature.split('=',1)[-1].split(',')
    a.append(f.strip())
```

Figure 2.11: List comprehension

Aside from the problems with having to compensate for varying data types between Java and Python, and Python's extensive capabilities, there is another downside using Python as a meta language for UAB. This issue is related to the readability of the templates, and is caused by a significant difference between the structure of Python and PLC languages.

SCL code is syntactically based on Pascal, and is characterised by its clear block-structure. Additionally, each statement is delimited by a semicolon, making multi-line code a commonly used language feature. On the other hand, Python enforces a strict indentation-based formatting, in which new lines are used to delimit statements. The idea behind Python's syntactically significant whitespace is to improve readability [56]. Whether it is successful at this, is an area of debate. Nonetheless, many of UAB's users indicated that coming from a block-bounded language in which whitespace is optional, using a language that enforces indentation required some additional effort. Sheard [45] also discovered that "templates that 'look like' object language programs are a great boon to the programmer", which is in line with the experience that the users had. The developers also indicated that for large template files, with many nested statements, readability did in fact suffer from the lack of explicit block bounds.

It seems that Jython is technically well suited for UAB. However, disconnections caused by lack of experience, differences in languages, and an imperfect mapping from the UAB backend to the templates, harm its overall potential for this particular use case.

2.2.4 Validation

As pointed out in 2.1.3, errors are not detected till a stage later in the generation and compilation process. This is because UAB does not natively provide PLC code validation.

The validation of the UAB generation cycle is performed in 3 distinct phases. Template files are validated upon execution, as any Python program. Each generated code file is then run against a user-defined set of Python functions, which flag the generation as successful or unsuccessful. Lastly, the generated code is imported into Siemens' Step7 environment, where it is validated upon compilation, which needs to be triggered manually.

As became clear from the sections on Syntax and Implementation, SCL templates are essentially Python programs, and are run, and therefore validated as such. Since Python's type system is dynamic, SCL templates are validated at runtime.

Since the SCL programs are formed from String parameters passed to Python code, the default validation of Python code provides no detail about the quality of the generated code, not even whether it is valid SCL syntax. A successful generation only means that the template was executed without problems, in other words, the template was valid. In fact, it does not even guarantee that the whole template is valid, rather that the executed path is valid. It might be possible to write a regular expression-based validator for the SCL blocks. However, segmentation of the code blocks, and its dependence on the control flow of the surrounding logic, make it difficult to draw conclusions on the code that will be generated. The degree of syntax and semantic validation would generally be limited to an approximation at best. An example of this can be seen in Figure 2.12.

Validating this specific use case comprises the parsing of the first block of SCL code, detecting this is missing a segment, and validating that at some point the missing segment is provided. This is actually a relatively easy example, which can be solved, despite the segmentation, but it shows that segmentation greatly hinder the validation capabilities.

```
plugin.writeSiemensLogic('''
IF CONDITION_1 THEN
''')
if condition_1:
    plugin.writeSiemensLogic('''
    // arbitrary code
    ''')
else:
    plugin.writeSiemensLogic('''
    // arbitrary code
    ''')
plugin.writeSiemensLogic('''
END_IF
''')
```

Figure 2.12: Segmentation

Since UAB does not feature any additional validation compared to regular Python, the possibility exists that a valid template always results in an invalid SCL code file. This shows that a clear gap exists between the state of a template and its output. This gap contributes further to the aforementioned disconnect between the goal of the templates and the way they are formed, suggesting that the current approach is not desirable.

To provide some sort of validation, UAB allows template developers to write a set of Python functions which can be used to validate the generated code. However, since no parser is present, the validation largely needs to be performed based on regular expressions. While this is easier to do on generated code than it is based on templates, it is still tedious. Based on the projects that were analysed, this form of validation was rarely used. If it was utilised, it was mostly to validate whether all declared parameters were actually set in the code.

After generation and importing the output SCL code, it can be validated at compile time. In case of PLCs, runtime validation could prove to be disastrous, with critical errors not being caught until the facility is up and running, so having compile time checking is a great feat. However, in Step7 the manual trigger enforces a full compilation of the program, which can sometimes take several minutes. This is a long time if the developer edits something inside the editor and wants to see whether it results in valid code. While some errors might be costly to detect on every code adaptation (e.g. array dimensions, cross-file interaction), there are many that could be detected and reported to the user on every iteration (e.g. syntax validity, type limitations, variable issues). This can be considered a limitation to the Step7 environment. However, it should generally be avoided that the developers fix the generated code, rather than the template.

An error in the generated code is often an indication that the template is wrong. Syntax errors are most common, but semantic errors can also be the result of an erroneous template. In other cases the error is caused by faulty data in the specification file, which will always lead to semantic errors. As discussed in section 2.2.1, the specification file is very liberal in what is considered valid, increasing chances of errors occurring. Each erroneous entry in the specification file results in a single faulty output file, whereas an error in a template will propagate into all output files originating from that template. This means that an error in a template is generally more costly than a specification file containing a faulty entry. An ideal workflow would resemble the one in Figure 2.13.

Regardless of the source of the error, developers often consider it faster to fix the errors inside the SCL editor. The alternative of fixing the template/specification file, triggering a regeneration cycle, and importing the project, is slow. As a result, the realistic workflow diverges from the ideal one, and can be seen in Figure 2.14. The operational steps, introduced in Chapter 2.1.3, have been mapped onto the workflow. This mapping leads to a similar observation with respect to the costliness of errors: The impact an error makes, increases the further the developer progresses within the workflow.

So, even though making manual adaptations to the generated code is the most time-efficient, it is unfavourable. Fixing code manually can cause new errors, especially when considering code generated from a faulty template. In these cases the fix would have to be performed in several generated code files, making it enticing to use the error-prone copy-paste method. Additionally, with every change made in the generated code file, the result and the source diverge more. If the

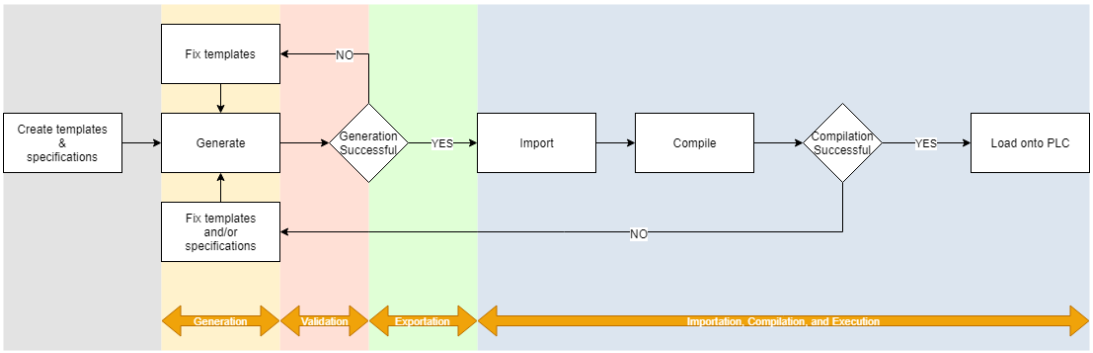

Figure 2.13: Representation of the ideal workflow.

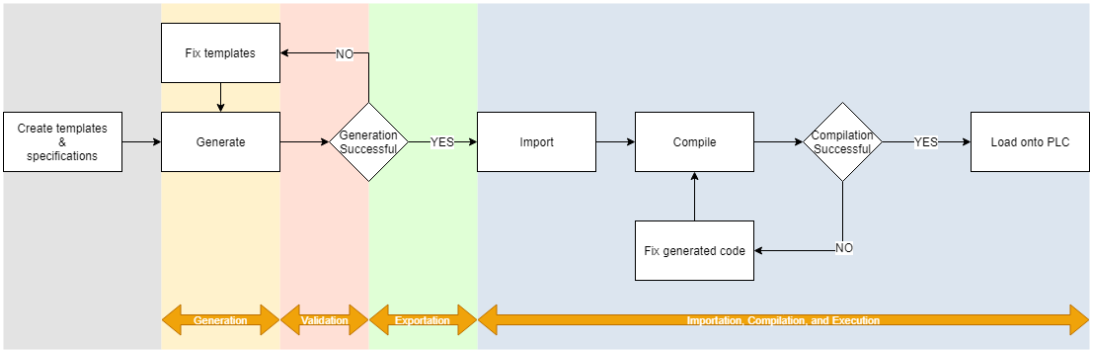

Figure 2.14: More realistic representation of the workflow.

cause of the error is not fixed, the code loaded onto a PLC is no longer in line with the template it originated from. This could cause issues when code has to be regenerated at a later stage as the error must be fixed again, and there is no guarantee that it is fixed in the same way that time around. Lastly, it would be much more effective if the developer of the current template fixes the error in the template as they have intrinsic knowledge of it, and therefore are able to pinpoint the error more easily. If the template is passed on to a new developer it will most likely take more time for them to figure out the origin of the error, unnecessarily wasting time.

For these reasons, having a disconnect between the source files and the generated code is a situation that must be avoided. CERN is aware of this, and as a result the UAB developers were specifically asked to make sure that the templates were always kept up-to-date. However, coercing people into performing additional tasks is often met with resistance. Additionally, it has no guarantee of succeeding, at least not without (enforcing) serious consequences [10], which is undesirable. For this reason it proved to be a gamble to leave it up to developer to ensure that any change in the generated code was also made in the source files.

An alternative to the manual approach, might be to find a way to automatically infer changes from the changed generated code, and propagate this back to the template or specification file. In reality, this might be feasible, but could prove extremely difficult. Aside from the technical

implications, dealing with multiple sources, and changes whose form is unknown beforehand, back-porting changes is generally non-deterministic. As a result, the developer would still have to make sure that the inferred changes are correct, which comes with the same difficulties as the current approach. An approach without an additional manual phase is to detect errors earlier. Structurally improving the templating facility in such a way that errors are detected before importing and compiling, preferably even before generation, reduces the chances of errors occurring at compile time. Furthermore, it would reduce the labour involved in the overall development cycle, which could aid in enticing the developers to fix the source rather than the output.

2.3 Evaluation

Now that the 'dirty' laundry has been aired, it becomes possible to judge the framework's implementation against the goals set by the creators. It is obvious that UAB succeeds at effectively reducing the time and effort involved in developing a new control system application. It features a rich set of functionality that makes it possible to create abstractions of PLC code, which can then be used to generate specific implementations. Therefore it is safe to say that UAB meets its intended purpose.

Claiming that the generated code is well-structured is a different story. The developer is in full control of the structure of the resulting PLC code. In fact, in Sections 2.2.2 and 2.2.4 it was demonstrated that they can even generate something which is not PLC code at all. Based on other documentation on the UNICOS framework (of which UAB is a component), the aim might not have been to create well-structured code, but to have a uniform way of defining devices and device instances across different manufacturers [23][39]. This they did succeed in, as the device instance files are all well-structured and normalised. They can even be validated using custom semantic checks written in Python.

The scalability criteria is hard to value. UAB's architecture is designed in such a way that new languages, object, and components can be added at any time. However,this is only because the impact of each element is minimal, barely touching the core functionality of the tool. The actual architecture, as presented in Section 2.1.3, shows that the only language-specific components consist of a few Python functions. The plugins do not affect the meta-language in any way, as this is fixated in the core component of UAB. An important note is that because the plugin layer is so thin, the level of language-specific functionality is as well. Something that could be improved in this area is to increase the depth of these elements, but overall it is a good approach to have a plugin per object language.

Another requirement of UAB is to be user-friendly. The description focuses mainly on the User Interface (UI), and barely on the templating language. This is weird as the user generally spends significantly more time working with the meta-programming language compared to the UI. As became obvious from the analysis in Section 2.2.2, the meta-programming language is all but user-friendly. It is verbose, counter-intuitive, and features no real validation. On the other hand, the UI seems to meet the standard, being both simple and intuitive. All in all, the criteria is met based on the description, but the scope is too narrow, especially taking into account the level of user interaction each component of UAB has. If the description were to include the

user-friendliness of the language, UAB would not live up to its design criteria, as it is not as user-friendly as it should be.

The requirement for version-management is met. This is just because the requirement is only meant to hold for the UAB framework, and not for the templates themselves. This has been an explicit choice, which is justifiable. New device specifications are released to support newer device instances, which are not bound to be backwards-compatible. For that reason it makes sense to ensure that the tool can support older device generations, but that templates are generally written for a single generation only. Using the same logic for a new device instance will most likely require a set of adaptations to the template. UAB supports different device specification versions through bootstrapping mechanics.

If one was to step away from the UAB definition of version-management, and simply look at storing code and documenting its evolution, there is generally not that much that needs to be done. Utilising any versioning tool like SVN or Git would allow this to be done. However, it is important to note that in the current form, developers choose to fix errors after generation rather than fixing the template. This creates a discrepancy between the actual code files, and the ones that can be generated from the template. So while this is not in the actual goals of UAB, it is a strong advocate for adding validation to the templates.

In short, evaluating the criteria against the implementation revealed that the most gain is to be made in the syntax and the validation aspects of the meta-programming language. This directly correlates to the large degree of user-unfriendliness and lack of structure. The scalability requirement is currently met only because of the liberal nature of the programming language. This means that when a new meta-programming language is introduced, it is important to re-evaluate the scalability criteria, and how it is represented. The user-friendliness criteria will also be tested against the new language implementation. The other criteria are either met because of a different component of UAB, or because they are part of the intrinsic properties of a meta-programming language. Therefore they can be assumed to remain in place when redesigning the language.

Having this information makes it possible to define a list of explicit requirements that the new meta-programming language needs to meet:

- The tool should support statically loaded information, i.e. configuration and specification files, in all of its functionalities.
- The object and meta syntax should be distinguishable.
- The overall syntax should be readable.
- The user should feel like they are using an extended version of the object language.
- The boilerplate code should be kept to a minimum.
- Design one way to do something, and do it well.
- The meta-programs need to be validated, syntactically and semantically.
- The quality of the generated code should be safeguarded.

- Meta-expressions should be eligible for splicing without having to be lifted into variables.
- The language design needs to be portable to other PLC languages, preferably by reusing components.

These requirements can be seen as the goals for the new version of the UAB language facility. They embody both the original goals for UAB, as well as the improvements to be made to the current situation. Aside from the criteria that apply to the language design, the actual implementation of the language should be implementable on top of the JVM to be able to incorporate it into the existing architecture of the greater Unicos framework.

Chapter 3

Meta-programming approaches

In the previous section the flaws in the current approach were identified, which resulted in a list of points to improve upon. One of the largest issues that was discovered was the counter-intuitive way code is currently generated, i.e. writing sections of PLC code in Python strings. This suggests that a different approach to templating is required to obtain a more usable template facility.

In order to determine a better approach, the concept of meta-programming was explored in detail, as well as various meta-programming solutions that were created in the past. The field of meta-programming will be introduced in short, after which the gained insights are detailed, resulting in a set of constraints for the new meta-programming language.

Afterwards various type system approaches as used in existing meta-languages will be compared, leading to a general set of ideas on how to provision typing within a meta-programming context.

3.1 Introduction to meta-programming

Meta-programming started off with the ability to manipulate programs at runtime, by implementing the Von Neumann architecture [59], putting data and programs in the same space. Greenwald et al. [24] expanded on this by defining a macro facility that made it possible to re-use common code elements, increasing the efficiency of programming, and reducing the chances of mistakes being made. The application of meta-programming has long since been extended significantly, and proved useful in a variety of fields. e.g. shader programming [36] and industrial automation [30]. Many of these applications originated from the academic world. However, meta-programming is supported by a significant amount of general purpose languages in one way or another, making it available to a larger public.

Besides being applied in different fields, meta-programming can be used for various purposes. One of its prime uses is to improve developer productivity, and increase code reusability and maintainability [50]. It is also used to improve performance, for example through partial evaluation [12] or specialised code generation [28]. This specialised code generation also facilitates DSLs and program transformers, like compiler-compilers [21]. Besides generation, meta-programming also makes it possible to reason about object programs. These analysers can

validate behaviour [9], can be used to construct theorem provers [5], or can perform introspection [19][22]. Lastly, Sheard [45] also recognises an educational element in meta-programming, as complex algorithms can be dissected into intermediate stages.

The variation in its application is also represented in the definition of meta-programming. In fact, there's no single definition for meta-programming. Depending on the point of view, multiple descriptions of meta-programming exist. Some focus on meta-programming as a concept [57], others define it from a generator point of view [6][43][34][13], and some address its analytic capabilities [7]. Sheard's definition [45] is probably the most complete, as it addresses both the generative and analytic nature of meta-programs:

"A meta-program may construct object-programs, combine object-program fragments into larger object-programs, observe the structure and other properties of object-programs."

Based on the definitions, it is up for discussion whether UAB's approach to templating is to be considered meta-programming. Its meta-capabilities are limited to generation only. Additionally, according to Štuikys and Damaševičius' definition a meta-program handles a program as data [48]. While UAB is indeed capable of generating a program from its data, it is also capable of producing output which is not a syntactically valid PLC program. In that sense it is simply handling String data which might result in a program, rather than a program itself. On the other hand, Mainland [35] considers this "printf purgatory" a form of code generation, which means that the Python based templates are to be considered real meta-programs.

Regardless of whether UAB is to be considered meta-programming, chapter 2 revealed that the current approach requires some reconsideration. In order to do so, it is important to look at the various aspects of meta-programming, and decide what a more suited approach would be.

Simply looking at all available literature is likely to result in a collection of approaches from which it is going to be impossible to distill a suitable one. Rather than blindly comparing all the different meta-programming solutions, it is much more sensible to classify them based on some clear characteristics, and make a decision based on those. To do so a taxonomy on meta-programming can be used, of which several exist. There's Sheard's [45] and Pasalic's [41] versions, which cover similar aspects. In comparison to Štuikys and Damaševičius' taxonomy [15], these taxonomies are less detailed, and focus on meta-programming systems rather than meta-programming as a concept.

Examining meta-programming from a conceptual point of view can yield precious insights. However, given the context it makes more sense to compare meta-programming approaches at a higher level. Additionally, the higher level separation is more clearly represented in the literature on meta-programming, resulting in a better classification. Even though Sheard and Pasalic consider many of the same aspects of a meta-programming system, there are a few characteristics that are unique in either of them. Whether this puts one ahead of the other is not important in classifying the different meta-programming approaches. In fact, it might be beneficial to cover the complete set of characteristics as depicted in both taxonomies in order to obtain a clear classification.

The list of characteristics of meta-programming languages obtained from merging both taxonomies is listed below. Each will be discussed separately in more detail:

- Separation of languages (homogeneous and heterogeneous)

- Use time (static and run time)
- Separation of static and dynamic code (manual and automatic)
- Type of meta-language (open and closed)

Aside from the mentioned characteristics, both taxonomies also differentiate on the type of meta-program. Both advocate that there are 2 distinct types: program generators, and program analysers. Since the focus is to generate code, this distinction is omitted. This does not mean that meta-languages used for analysis cannot provide useful insights, but they are simply out of scope.

Beyond the aspects of meta-programming that are covered in the taxonomies, there are some additional subjects which are worthwhile to address:

- Hygiene and variable capturing
- Type validation in meta-programming

3.2 Separation of languages

Aside from the distinction between generative and analytical meta-programs, both Sheard and Pasalic categorise meta-languages based on what Štuikys and Damaševičius refer to as: 'the separation of languages'. The name is somewhat poorly chosen, as their definition does not concern how languages are separated. Instead, they distinguish languages based on whether the meta-language is equal to the object-language, and whether the same compiler/interpreter can be utilised to execute both meta-programs and object programs. Herein they define 2 possibilities, namely homogeneous and heterogeneous meta-languages. As the prefix might suggest, homogeneous meta-programming is characterised by the ability to use the same tool to compile/interpret the meta program as can be used for the object program. The execution patterns for a two-stage, homogeneous meta-program look like this:

$$exec_x(meta\text{–}program) \Rightarrow object\text{–}program$$
$$exec_x(object\text{–}program) \Rightarrow result$$
$$exec_x(exec_x(meta\text{–}program)) \Rightarrow result$$

Examples of languages that follow this execution pattern are: MetaML [51], Template-Haskell [46], and Converge [53].

For heterogeneous meta-programming this does not hold. In that case the meta-program is executed in a different manner than the object program. The respective patterns for a two-stage, heterogeneous meta-program are:

$$exec_x(meta\text{–}program) \Rightarrow object\text{–}program$$
$$exec_y(object\text{–}program) \Rightarrow result$$
$$exec_y(exec_x(meta\text{–}program)) \Rightarrow result$$

F# [49], the C preprocessor [47], and MetaHaskell [35] are examples of heterogeneous meta-programming languages.

Figures 3.1 and 3.2 show Hello World examples of both. Pasalic mentions that while it might be meta-programming, it is hard to classify Figure 3.1 as being homogeneous meta-programming. The reason for this is that it does not depend on Python's internal mechanisms. Instead, it creates a new file, based on a String, which is then executed. A Hello World example which would meet Pasalic's requirements, is given in Figure 3.3. This too is written in Python, and makes use of the capability to define classes at runtime to go from a meta-program to a result. It is obvious that this form of meta-programming is much more powerful than a string-based approach, but it is also more complex.

```
execfile('meta_program.py')
execfile('object_program.py')

with open("object_program.py", "w") as text_file:
    text_file.write("print(\"Hello␣World\")")
```

Figure 3.1: Homogeneous, string-based meta-programming

```
execfile('meta_program.py', r)

with open("object_program.sh", "w") as text_file:
    text_file.write("#!/bin/bash")
    text_file.write("echo␣\"hello␣world\"")

### from within cmd ###
./object_program.sh
```

Figure 3.2: Heterogeneous, string-based meta-programming

Even though Figure 3.2 is only a few lines long, it shows one of the biggest downsides to heterogeneous meta-programming. This downside is not the product of the string-based approach used in that example; it is characteristic of heterogeneity. It is the requirement to know multiple languages. In the other examples only Python is used, meaning that a Python developer can at least read them, and most likely understand them. In the heterogeneous example the developer needs to know both Python and Bash.

Aside from the pedagogic advantage of only needing to master a single language, there are benefits that are largely unique to homogeneous meta-programming. When the meta-language is identical to the object language, the same type system can be used to validate each stage. This not only facilitates maintainability, it also guarantees type safety across stages. Additionally, because the meta-language is specialised for a single object language, it is possible to support object-level evaluation. This allows sections of code to be executed and replaced by their respective values, potentially improving performance.

```
# Define the meta class
class HelloMeta(type):
    def hello(cls, a):
        print("Hello %s" % (a))

    def __call__(self, *args, **kwargs):
        cls = type.__call__(self, *args)
        setattr(cls, "hello", self.hello)
        return cls

# Define an implementation of the meta-class
class HelloWorld(object, metaclass=HelloMeta):
    def greet(self):
        self.hello("World")

# Define another implementation of the meta-class
class HelloJohn(object, metaclass=HelloMeta):
    def greet(self):
        self.hello("John")

# Call the object code
hello = HelloWorld()
hello.greet()
hello = HelloJohn()
hello.greet()
```

Figure 3.3: Homogeneous meta-programming using Python3's internal mechanisms

While not all homogeneous languages make use of these advantages, it is obvious that these are powerful mechanisms. Heterogeneous language might lack these properties, but have their own set of strong characteristics. The most prominent ones are their reusability and the fact that they can be used without any restrictions on the host language [54].

In homogeneous situations, the meta-language and the object language are identical, and therefore are only suited for that specific object language. This does not hold when using a meta-language that is different from the object language. There is nothing that prevents the use of the same meta-language with a different object language. As a result, a single meta-language can provision many different use cases. It is important to keep in mind that creating a reusable language is not without concessions. The language should effectively only support the minimal unionised subset of features of the object languages it is meant to facilitate. Any feature outside of this set might have no meaning for a certain object language, and therefore should not be part of the meta-language. To avoid losing crucial meta-functionality, support for such features needs to be achieved in an alternative manner, e.g. through extension of the meta-language.

The two-language composition is not all bad. Because of it, heterogeneous languages arguably enjoy a higher degree of comprehensibility. In case of different languages it is much easier to differentiate between the two stages. As such, a developer can first try to understand what goes on in the meta-phase, before moving on to the object code.

Additionally, the type safety that homogeneous meta-languages offer, can also be achieved when considering two different languages. It would require a significant section of the object language's semantics to be encoded. Tratt [54] describes this as unrealistic for relatively large DSLs, but not impossible. So while there are many benefits to homogeneity, there's also potency in heterogeneous meta-programming.

With a clear overview of the strengths and weaknesses it is possible to make a calculated choice between the two, but it is crucial to also take the object language's capabilities into account. A language that does not have meta functionalities, and cannot be extended with them, is not suited for homogeneous meta-programming. In that case, a heterogeneous meta-language is generally the only choice. Nonetheless, even for such cases, it is possible to mimic a homogeneous setup. Assuming the object language is powerful enough, one can always create a new program for the same language, e.g. through string concatenation (see Figure 3.1) or AST building. Alternatively, if the language is being reimplemented, mechanisms (e.g. annotations) can be added that use the existing syntax to facilitate meta-programming. In this approach it is important to stay as close to the original language as possible. Diverging too much from the object language will limit the advantages that a homogeneous meta-language offers. Additionally, it may cause confusion if a similar syntax is used, but methods and statements are added. For example, a developer might try to use a meta-stage operation in an object stage. This obviously would be caught by the interpreter or compiler, but the developer might not be able to tell why.

3.3 Use time

Sheard's taxonomy recognises the split in the use time of various meta-programming approaches. On the one hand there are static code generators, which produce object programs that are to be processed by a different compiler or interpreter during a separately triggered execution phase. On the other hand, there is run-time generation, where the generated code is executed within the same execution phase. Berger et al. [8] describe run-time evaluation as 'conceptually simple', as evaluation is done anew every single time. Evaluation occurs like with any other function, making the impact of side effects predictable. Compile-time evaluation can have unexpected side effects. Figures 3.4 and 3.5 are examples taken directly from Berger et al [8]. The former is evaluated at run-time, and prints '1 3 6', whereas the later is compiled and will print '3 1 6' on execution. This shows that compile-time evaluation can change the output of a system in such a way that it is no longer in line with the order it was defined in.

```
print(1);
print(2 + eval(print(3); ASTInt(4)))
```

Figure 3.4: Run-time evaluation

```
print(1);
print(2 + ↓{print(3); ASTInt(4)})
```

Figure 3.5: Compile-time evaluation

Compile-time has the advantage that it has access to the internal of the object language. It can enforce hygiene, and it can be exposed to the same optimalisation cycles that regular code is put through. The only thing it cannot do, is reference variables, as these are not available until run time. Run-time evaluation has this ability, which allows code to be specialised based on dynamic values. For example, the code in Figure 3.4 would have worked identically if the value '4' would have been passed to it via stdin. On the other hand, run-time evaluation cannot easily make use of the internal of the language.

One big advantange to compile-time meta-programming is that it incurs no run-time performance penalty. Using meta-programming at run-time has an impact on the overall performance, but if this can be moved to the compilation phase, this impact moves with it. As a result, the compilation might take longer, but the run-time remains unaffected. An additional advantage to run-time meta-programming is that it can be applied in various stages. This is called multi-stage meta-programming, and it means that the outcome of stage n is a program which is a meta-program itself. This makes it possible to define layered abstractions, which can further increase code re-use and specialisation at runtime. Technically multi-stage is also possible for compile-time evaluation, but since all values are then known statically, it provides no gains over a single-stage meta-program.

Examples of compile-time meta-languages are Template Haskell, Yacc, C++ and Lisp. Lisp is special, as it also supports run-time evaluation. This also puts it in the list of run-time meta-languages, together with JavaScript, Python and MetaMl.

3.4 Separation of static and dynamic code

Another aspect of meta-programming languages that Sheard covers in his taxonomy is the manner of how meta-code is separated from object-code. In case of heterogeneous languages, this can be done automatically by the interpreter, based on differences in the syntax between the meta-language and the object language, assuming total disambiguity. In case of heterogeneous languages this can be done through a process called Binding Time Analysis (BTA). BTA allows the compiler to determine which values are static, and which are only available at run-time.

An alternative approach is to incorporate manual, or explicit, annotations. In those cases, the developer is in charge of marking the static elements. Any unmarked element is assumed to be dynamic. Template Haskell and MetaML use this form of annotations.

Automated separation of static and dynamic code is user-friendly in the sense that they do not have to do anything. However, there are situations where it gets it wrong. BTA makes assumptions in order to determine execution order, which might not be in line with what the user intended. This becomes more likely in multi-stage setups, where BTA is performing poorly. Additionally, BTA requires the syntaxes of both languages to be disambiguate. For manual

annotation this is not required, assuming the parser is instructed how to treat annotated and unannotated language components. Manual annotations gives the user more control over the staging, but also requires them to write the annotations themselves.

While the statement that manual quotations are cumbersome has largely been debunked, there is still something to say for cases where it is redundant or unimportant. If there is no chance for BTA to get it wrong, or if the user does not need to know what goes on, requiring manual annotations actually is cumbersome. In these cases a hybrid model might be implemented which provides the best of both worlds.

Idris [11] uses such a hybrid model. Annotations are only required on elements where the compiler cannot faultlessly determine whether it is static or not. This means that type classes, which are always static, are automatically marked as such. Function calls on the other hand require manual marking using the '%static' annotation. Lisp does something similar, as macros need to be explicitly defined, but their use is inferred, so that the user does not need to concern themselves with that.

Hybrid models do require an additional level of clarity. From a user's standpoint, there should be no confusion on whether an annotation should be added or not. This requires a well-defined collection of rules when and when not to add explicit annotations.

3.5 Type of meta language

Unlike Sheard, Pasalic also differentiates meta-languages by whether they can be extended or not. A closed meta-language cannot be extended upon. Decisions regarding typing, syntax, and variable capturing are made by the meta-language designer, rather than making it the users responsibility. Additionally, pre-defining these aspects of a meta-programming language makes it possible to guarantee certain meta-properties, e.g. type-safety and hygiene.

MetaML [51] is an example of a closed meta-language. It provides the user strong typing guarantees and capture-avoiding mechanisms, putting these beyond the developer's concern. Because of its specialisation for a single language, it is powerful, but it lacks extensibility.

Closed meta-programming is the best choice when the object-language is fixed. However, when the meta-programmer needs to be able to use varying object-languages, a closed approach simply is not viable. It might be possible to support a few object-languages from a closed point of view, but if it has to be truly dynamic, an open meta-language is the only option.

Open meta-languages can be tailored to the object language that the developer wants to use. The downside is that the implementation of most of the language features is their responsibility. The best the language designer can do is provide abstractions in the meta-language. These allow programmers to easily adapt it to their intended object-language. Most open meta-language provide 'interfaces' for certain language aspects, like syntax or typing, but generally these are incomplete.

For example, Standard ML [40] allows the user to define the syntax of the object-language, guaranteeing that the generated code is syntactically correct. However, it does not allow the user to define the semantics in a way that guarantees type safety.

There is a strong relation between the extensibility of the language, and whether the meta-language is heterogeneous or homogeneous. As mentioned earlier, heterogeneous languages can

be reused with various object languages. However, for this to be able, the meta-language cannot contain object-language specific knowledge. This specialisation has to be performed by the user at a later stage. As such, the meta-language needs to be extendable.

Since extensibility in this case is a must, a closed, reusable, heterogeneous meta-language does not make sense. It is either going to be so general that it lacks any feature, or it is only specialised for a set of near-identical languages, and therefore not truly reusable. However, closed, non-reusable, heterogeneous meta-language are a viable option, for example when the host language doesn't allow for homogeneity.

Something similar, but less definitive holds for open, homogeneous meta-programming. As homogeinity applies for a single, well-defined object language, extensibility is often a wasted feature of the accompanying meta-language. The only viable use case is when an object language allows for extension of its own syntax or semantic rules. In that case the meta-language should allow the same form of extension, however, no example of this was found.

3.6 Hygiene

Accidental variable capturing is a well-known problem in macro-based code generation. Hygiene, coined by Kohlbecker et al. [32], is the principal of avoiding this accidental capture upon macro expansion. Unhygienic macros are a liability, as they will function correctly in all situtations with the exception of at least $2^n - 1$, where n is the number of macro parameters. This number is equal to all minimum set of parameter configurations in which at least one variable is in danger of being captured. In short, hygiene violations can occur when the macro code is scoped within the lexical context of the calling site.

```
#define INC1(i) do {int x=0;\
                    ++i;\
                    printf("%d\n", i);\
                } while(0)
int main() {
    int x = 4, y = 4;
    INC1(x); // prints 1
    INC1(y); // prints 5
    return 0;
}
```

Figure 3.6: Hygiene violation

An example of hygiene violation in C macros is shown in Figure 3.6. Since both inputs have the same values, it is sensible to assume that their outcome would be identical as well, which is clearly not the case. Since the macro is evaluated based on the caller's lexical context, the arguments do not receive a new binding. As a result variable 'x' in the calling site ends up being shadowed within the expanded macro-code. If the lexical contexts were kept separate, the

assigned binding for variable ‘i’ could never end up being shadowed by the macro’s variable ‘x’, guaranteeing hygiene.

Macro expansion as done by the C-preprocessor [47] is based on string replacement, and is not just vulnerable to accidental variable capturing. It also is susceptible to precedence issues. The processor takes the macro parameter and splices it into the macro as is, which can lead to unexpected results, as displayed in Figure 3.7.

```
#define EVEN(x) (x % 2 == 0)
int main() {
    printf("%d\n", EVEN(2 + 2)); // prints 0
    printf("%d\n", EVEN(4)); // prints 1
    return 0;
}
```

Figure 3.7: Precedence violation

Once again, the arguments that are passed to the macro are semantically equivalent, while their outcomes are not. As can be seen from the provided examples, non-hygienic macros can severely influence the functionality. Including hygienic macros in the design of a new language can prevent these issues from occurring.

A macro is referred to as a hygienic macro if the macro expander itself takes care of solving the hygiene problem. Languages like Rust [31], Scheme [17], and Racket [18] have such macro systems in place. Some languages, like Elixir [37], allow the user to explicitly override hygiene for a macro, giving them more control. In languages that do not have hygienic macros (e.g. Common Lisp) the user is burdened with the responsibility to avoid accidental variable capturing.

Manual solutions embody the use of obscure names which are unlikely to occur in practice, using methods to generate unique names, or using a special syntactical element which is only available within a macro [1]. Which approach works depends on the language. If no special format is supported, or if no ‘gensym’-like function is available, developers might have to prevent variable capture themselves by coming up with random or obscure names.

In general hygienic macros are desirable, but the underlying issue is not easy to solve. The solution also greatly depends on the languge’s features, which hinders a standardised approach. If no hygienic system is put into place, the responsibility and the complexity moves towards the user, and away from the implementation. This increases the chances of the hygiene problem manifesting itself, as a good understanding of the issue is required to avoid it. Both these scenarios indicate that hygiene is a difficult subject.

As for PLC languages, this complexity is no different. An additional problem is that none of the manual approaches work for languages adhering to the IEC standard [52]. These languages require all variables to be defined before the logic sections. As such, if a hygiene violation would occur within the logic sections, no new variable can be introduced. Similarly, if the macro uses obscure names, these names will have to be declared in the template. This requires the developer to have intrinsic knowledge of each macro that is used. It is also not robust as the obscure names might be used within the template itself, or in another macro.

For lack of a good solution, there are some potential workarounds. The first one is to identify which variables are statically introduced in the macro. Upon expansion the name of each of these can be adapted (to avoid conflict), after which they can be added to the temporary variable declaration at the top of the output file. However, this can severely impact the code's functionality, and since PLCs are physical machines, it also might unintentionally affect block allocation. A different option is to accept macro expansion as is, and only invalidate generation when a case of hygiene violation occurs. Since these situations are likely to be detected at runtime, it can result in unexpected errors.

3.7 Type Validation in Meta-programming

Type systems in meta-programming know various forms. Similarly to regular type systems, there is the distinction between static and dynamic type systems. Another distinction, which solely applies to meta-programming is the one between isolated stages and cross-stage validation. In isolated validation, each stage is validated individually.

Having a type system that can validate both the meta-stage and the generated object stage is obviously more powerful than one that can validate only a single one. However, given the fact that both stages are validated separately, it is still very likely to generate invalid object code. Not being able to detect errors beforehand will lead to code regeneration which would otherwise be preventable.

Validating the generated code from the meta-stage, that is where a new type system can really prove its worth. In order to do this, it needs to be able to do cross-stage type checking. MetaML is a meta-programming language with such a type system. It does not just ensure that the resulting code is syntactically valid, but also semantically sound. Achieving such levels of type safety is difficult. For homogeneous meta-languages it is possible to wrap the object language's types in a code object. The language's type system can then be used to validate the meta-stage. It might require some small additions to do so, but it's generally less effort than creating a new type system.

Sheard [45] recognises the difficulties of such an approach for heterogeneous meta-programming language. For starters, a different type system might be required for each object language that should be supported. Secondly, the type system might not be suited to be modelled using a code data type. Such a type can still be used to lift object language elements into the meta-domain. However, for objects anti-quoted into the object language it will not always suffice. Where a homogeneous language has a code equivalent for each data type, this might not apply to a heterogeneous language. When dealing with two different languages, a type mapping has to be created from the meta-language to the object language. This mapping might be obvious, but it could also be a daunting task. This greatly depends on whether the type system from the object language is compatible with that of the meta-language. MetaHaskell manages to guarantee well-typedness of generated object language terms through the use of a quasiquotation mechanism [35], proving that it is possible (to an extent). This is even more impressive considering MetaHaskell is an open meta-language.

For closed heterogeneous meta-languages it might be easier to achieve cross-staged type-safety. The same restrictions apply to closed languages and open languages. The main difference

is that the object language is fixed. Therefore the meta-language does not have to support ways to define or infer a type system for the target language. The language designers can concern themselves with achieving type safety.

Besides depending on quasiquotation mechanism, it is also possible to use dependent types, or even the less powerful Algebraic Data Types (ADTs) [41]. Dependent types are generally more powerful than the code data type, as they represent types indexed by terms [42], rather than types indexed by types. The problem with dependent types is that they are harder to debug, and less transparent to the user.

While cross-stage type safety is something to be desired, it is not the case that validating each stage separately is worthless. If a type system is sound for each individual stage, it still provides a series of strong guarantees. *Given the absence of errors when statically validating stage n, it is ensured that no type errors will be encountered during the execution of stage n. Additionally, any type error in the code generated during the execution of stage n will be detected when validating stage n+1.*

Extending the type system with fully sound cross-stage validation will add another guarantee: *If no errors are detected at stage n, the object code of stage n+1 will be free of type errors as well.* Whether the absence of type errors in stage n dictates anything about stage n+m (with $m > 1$) depends on the information that is available at stage n.

3.8 Evaluation

The obtained theoretical insights can be mapped onto UAB's use case. This should yield a set of characteristics with which the new meta-programming facility can form an improvement over the current installation.

Preliminary research revealed that the languages cannot be extended natively to support meta-programming. As such, a true homogeneous approach is not feasible. Mimicking homogeneity is undesirable as the unique layers around the standard mean that there would be multiple meta-languages to create and maintain, none of which are truly reusable. Additionally, since the requirements for PLCs are focused on I/O operations, the languages' capabilities are lacking for the purpose of meta-programming. This means that creating a 'homogeneous' meta-language would require additions (e.g. methods, annotations, statements) to the object language. Each one would make the outcome more heterogeneous. At some point the languages will differ to such an extent, that the advantages of having a homogeneous language no longer apply. Additionally, for the PLC languages, this approach is not desirable. As the language is directly linked to the physical domain, it might cause confusion when it is lifted into a software domain. These reasons, combined with the re-usability that heterogeneity brings, vow for a heterogeneous approach, like its predecessor.

As for the type of language, taking into account the expertise of the users, combined with UAB's purpose, a closed language is desirable. That way, users do not have to concern themselves with the internals of the language. However, closed, heterogeneous meta-languages are do not offer reusability. In case of the PLC languages, the meta-language cannot be uniformly characterised as open or closed. To allow the meta-facility to support language specific elements, while at the same time reusing the same meta-language, it is important that the language is open.

In other words, it can be extended upon based on the use case. However, after specialising the meta-language for a single object language, the result should appear closed to the end-user.

A similar hybrid model is applicable in the separation of static and dynamic code. All PLC code is static, whereas all meta-code is dynamic. Whenever it is deemed clear enough, the interpreter infers whether a statement is dynamic or static. In the other situation, e.g. variable splicing, the user is responsible of annotating the dynamic element. As mentioned, this requires a clear definition of where it is and is not required.

Since PLC code is compiled within a different domain, the 'use time' is static in respect to the whole process. However, looking only at the templating stage, it's definitely a run-time process, as not all information is available statically. This also means that there might be a use for multi-stage meta-programming. However, currently a single layer of abstraction suffices since the meta-stage is mostly limited to values and identifiers, and occasionally a segment of logic.

Given the absence of multiple meta-stages, adding cross-stage type validation might seem redundant. This is not entirely true, since the principals of such a type system also hold between the meta- and object-stage. As explained in section 2.2.4, the absence of cross-stage validation drives the developers towards fixing the generated code, rather than the templates. Adding cross-stage type checking can prevent this, though the dynamic nature of the templates will result in these checks being soundy at best. Therefore it is key to ensure that the type system is also capable of validating each stage individually. The latter might result in errors being fixed in the templates, as the code no longer needs to be transferred to be validated, but it cannot prevent unnecessary code regeneration.

To facilitate the cross-stage type checking, a code data type will be introduced. This allows the users to write object-level code in its concrete syntax within the meta-domain. This improves the overall readability, and allow predictions on the validity of the generated code. Dependent typing might have allowed for a more complete type system, but also would be much more complex to implement. Additionally, much of the template parameters come from external information sources, meaning that the terms are not known, just their types, limiting the capabilities of dependent typing.

As discussed in section 3.6, writing PLC code has some implications when it comes to hygiene. Based on the existing templates and their corresponding user-defined functions, solving the hygienic problem can be put off (for now). In case of code-generating functions, the required identifiers were passed to the function, or were part of the global variables defined in the symbols table. In line with this, hard-coded identifiers will be invalidated within code-generating functions. This avoids hygienic issues from occurring altogether, but might turn out to be a draconic measure. If this is indeed the case, an alternative solution needs to be thought of.

Chapter 4

Language Design

The previous chapter covered a series of different language characteristics for meta-programming languages. The insights obtained during the analysis were used to determine a different approach to meta-programming for UAB. This new approach is SCL-Templates, or SCL-T in short. As the name suggests, it is a language that facilitates templating specifically for the Siemens SCL language. SCL-T offers the developer an intuitive syntax to write templates for SCL, while at the same time also providing static and semantic guarantees of the meta- and the resulting object-code.

The details of SCL-T will be covered below. Firstly, an overall introduction of the language's features and its templating mechanism will be given. This is followed by a in-depth example of how to use the language. After that the language's data types and statements are covered. Finally, some attention will be given to SCL-T's interaction with the static information sources.

4.1 Introduction to SCL-T

SCL-T is a heterogeneous, closed, 'run-time' meta-programming language that uses explicit annotations to separate the meta-stage and the object-stage. Unlike UAB's original meta-facility, this language is only suited for a single PLC language. In this case, the target language for the templates is Siemens SCL.

One of SCL-T strongest features is the way the meta-code is embedded into the object-level code. Instead of writing meta-level statements that generate object-level code, the user now writes SCL code enhanced with templating parameters. What this looks like is covered in section 4.2. Of course, the template parameters have to be assigned at one point or another, which is why meta-level statements are allowed alongside the SCL code. The locations within the SCL code where meta-programming is allowed are fixed to outside of SCL Unit blocks and within the logic sections of each block. Inserting code into the object level domain is also restricted. Standardising the use-sites not only opens the door to a more powerful language regarding validation, it also adds a degree of conformity over the templates. This ensures that the overall look and feel of a template resembles the outcome. Formatting the templates like this not only makes them easier to read, it also removes much of the boilerplate code that UAB forced onto the developers.

Another characteristic of SCL-T which is noteworthy is its validation. UAB had run-time validation of its meta-stage, and no incorporated validation of the object-code (pre- or post-generation). SCL-T offers compile-time guarantees on the validity of the meta-stage which help in ensuring that no type errors will occur when executing the template. Even though guarantees regarding sound meta-execution are important to have, SCL-T's real strength is in the validation of the (to-be) generated code. SCL-T ensures syntactic validity of the resulting SCL code before the code is generated. Because of this it is no longer possible to write a template which produces syntactically incorrect SCL code. Besides syntactic checks, SCL-T also support semantic verification at meta-level. This means that it can detect whether a type violation will be present in the generated code. Unfortunately, this check is not fully sound, meaning that the absence of errors in the template does not guarantee generation of valid SCL code. Nonetheless, SCL-T's type system can ensure that certain type of errors will always be detected before generation. After generation most of the holes in the code have been filled, allowing the type system to eliminate errors that were undetected while validating the template. Validating the outcome from within the generation tool can prevent unnecessary migration to the Step7 program.

Besides the obvious changes to the language, SCL-T offers some quality-of-life improvements over its predecessor. One of these is the removal of meaningful indentation. This makes it possible to write templates using an indentation that promotes readability rather than functionality. Additionally, since whitespace is no longer meaningful (outside of strings), it also becomes possible to write multi-line meta-programming statements. Semi-colons are used to indicate the end of a statement. Control statements are now bounded by explicit symbols ('{...}'). As a result, it now is easier to see where a block begins and ends, even for heavily nested blocks.

4.2 SCL-T by Example

Figure 4.1 shows an example of an SCL-T template. This is based on an actual logic file used at CERN. Part of the body has been omitted to reduce the overall size of the code block. The template is not overly complex, but a lot is happening in the template. This example will be used to introduce some of SCL-T's more prominent language elements.

Starting from the top, the first thing that is encountered is a series of 'supplied' declarations. These variables are passed to the template as a parameter, and have to be explicitly typed. If these variables would not be declared, the type system would not be able to fill in the gaps created by these external parameters. If a template has no 'supplied' variables every execution of the template will result in the exact same output (assuming the specifications haven't changed in between runs), as all information is present statically. Template parameters are constant in their declaration, which means that they cannot be reassigned within the template. However, their values can be mutable objects, a concept which is discussed in more detail in section 4.3.1.

Moving on to the first line of SCL code, an explicit annotation is encountered, '$...$'. This annotation is a recurring element within SCL-T. It indicates splicing, which is the insertion of a meta-level object into the object-level code. This is what gives SCL-T its meta-programming functionality. SCL-T recognises 3 different types of splices: regular splices, suffixed splices, and statement splices. The different types of splicing are covered within the example and will be addressed when they are encountered.

```
1  supplied <Var(UNKNOWN)> name;
2  supplied bool master;
3
4  FUNCTION $name$_GL : VOID
5
6  VAR_TEMP
7      old_status : DWORD;
8  END_VAR
9  BEGIN
10
11 list(<BOOLEAN>) conds = [];
12 var AllAI = findMatchingInstances(
13                  "AnalogInput",
14                  "'#FEDeviceIOConfig:FE_Encoding_Type#'='1'"
15              );
16 for var i in AllAI {
17     string AIDesc = i.getAttributeData ("DeviceDoc:Descr")
18                       .lower()
19                       .strip();
20     if AIDesc != 'spare'{
21         conds.add(<$i$.IOErrorW>);
22     }
23 }
24 if master {
25     $<DB_Global.AI_Sensor_Alarm := $orBuilder(conds,<0>)$;>$;
26 }
27
28     ...
29
30     IF UNICOS_LiveCounter - DB_Global.T_PulseAck > 1.0 THEN
31         $name$.Pulse_Ack := FALSE;
32     END_IF;
33 END_FUNCTION
```

Figure 4.1: An example SCL-T template, based on an actual production code file.

This particular case is a suffixed splice. It is a splice that appends a suffix to an SCL identifier. In this relationship the suffix is constant, but the base identifier is not. Suffixed splicing are particularly useful when iterating over SCL units or variables that have a common suffix based on a dynamic value. They are allowed at any point where SCL accepts identifiers. Suffixed splices cannot be typed, as the name of the newly created identifier is dynamic.

Executing the suffixed splice with 'name' being equal to 'PCO_10' would result in 'PCO_10_GL'. Valid suffixed splices have the guarantee to produce a valid SCL identifier. Any splice that vio-

lates this guarantee will result in a type error.

Skipping over the SCL declaration section, line 11 contains the next SCL-T language component: regular variables. Unlike template variables, regular variables have to be initialised upon declaration. It is not allowed to declare a variable without directly assigning a value to it. For collections it is possible to declare an empty collection. The initial declaration site of a regular variable has to be explicitly marked. This can be done with the 'var' keyword, which indicates that the type has to be inferred. This is possible because variable declarations are required to be initialised. The type of the variable can then be inferred based on the assigned value. If required, the type can be explicitly overridden, given that the assigned value is a subtype of the desired type. An example of this can be seen in line 17. For collections, type inference is only possible when the initialisation is not empty. In case of empty declarations, the type has to be explicitly specified, so replacing line 11 by 'var conditions = [];' would be invalid.

Explicit variable declaration is required whenever a new variable is declared. As a result it is also required in the header of a for-loop, inside control-statements, and inside user-defined functions. Without the explicit annotation, the type system will assume the variable to have been declared at an earlier point. If this is not the case, an error will be reported.

After they have been declared, variables can be reassigned given that the new value is a subtype of the declared type. Reassignment sites should not have the 'var' keyword, or an explicit type declaration. If the keyword or type is present, the type system will see it as an initial declaration site, resulting in a duplicate declaration of the variable. Variable shadowing is not allowed in any way. This means that if a variable has been declared, it cannot be redeclared within the same or any subsequent scope. Variables introduced within a child scope will not exist in the surrounding scope, and thus can be reintroduced within other sub-scopes.

Besides being the first encounter with regular variables, there is another interesting this about the declaration of the 'conds' variable. Its type, and in particular the '<BOOLEAN>' part of the type is new in SCL-T. '<BOOLEAN>' should be read as 'code of BOOLEAN'. This type is assigned to any meta-level object whose value represents a valid SCL expression of type 'BOOLEAN'. The 'code of ...' construct can be used for any SCL data type. Objects of these types play an important role in splicing.

Lines 11 to 26 consist of several meta-statements which contribute to a single purpose. The block queries information from the specification files to build a list of '<BOOLEAN>' objects, which are then used to construct an SCL statement. Even though the various statements are closely related, they will be covered individually to properly cover their functionality and that of the underlying SCL features.

```
12 var AllAI = findMatchingInstances(
13                  "AnalogInput,DigitalInput",
14                  "'#FEDeviceIOConfig:FE␣Encoding␣Type#'='1'"
15              );
```

Figure 4.2: Excerpt of Figure 4.1: lines 12 to 15.

Figure 4.2 is an excerpt of lines 12 to 15, which is a good example of SCL-T's interaction with the specification file. Anyone who worked with UAB before will recognise the syntax.

The ‘findMatchingInstances’ function is what is used to extract the device information from the specs. It takes a comma-separated list of device types, an optional Master object, and one or more constraints. It queries the specification file for a list of device instances of the requested type(s) that meet the condition(s). In this case the requested device types are ‘AnalogInput’ and ‘DigitalInput’. As a result the return type of the function will be ‘list(Device(AnalogInput, DigitalInput))’. This will also be the inferred type of ‘AllAI’ since it has been declared with the ‘var’ keyword.

The ‘Device(_)’ data type is a new addition in SCL-T and is used to qualify information obtained from the specification files and device definitions. This allows their information to be used for static and semantic validation of the templates. Besides creating a dedicated output type for the ‘findMatchingInstances’ function, SCL-T also offers validation on the query. The device types are validated against the known list of devices, and each device type’s metadata is used to determine the supported attributes. Querying an unknown device, or writing a condition based on an attribute which does not exist will result in a type error. Because of this it is no longer possible to write a non-sensical query.

One thing that is important to keep in mind is that attributes are validated against the intersection of the queried device types. This means that when dealing with multiple device types, the attribute needs to be existent in each of them. For this concrete example it boils down to the fact that the ‘findMatchingInstances’ call is valid if and only if attribute ‘FEDeviceIOConfig:FE Encoding Type’ exists in both ‘AnalogInput’ and ‘DigitalInput’.

```
16  for var i in AllAI {
17      string AIDesc = i.getAttributeData ("DeviceDoc:Descr")
18                        .lower()
19                        .strip();
20      if AIDesc != 'spare'{
21          conds.add(<$i$.IOErrorW>);
22      }
23  }
```

Figure 4.3: Excerpt of Figure 4.1: lines 16 to 23.

The next statement is the for-loop ranging from line 16 to 23, which is displayed once more in Figure 4.3. The purpose of the loop is to iterate over the device instances that were obtained in the previous statement, and build a list of conditions. The header of the for-loop will be discussed first, after which the contained statements will be detailed.

As mentioned earlier, new variables introduced in the header of a for-loop also have to be explicitly declared using either a ‘var’ keyword or one of the supported types. Since variable ‘i’ has not been introduced in the surrounding scopes, it has to be declared here. Its type is inferred based on the type of ‘AllAI’, which is ‘list(Device(AnalogInput, DigitalInput))’ as specified earlier. The type of ‘i’ will therefore be ‘Device(AnalogInput, DigitalInput)’.

The first child-statement of the loop takes the device instance ‘i’ and queries the parameter file for its description (“DeviceDoc:Desc”). Like ‘findMatchingInstances’, the ‘getAttribute-Data’ method is a remnant of UAB. Its purpose is to obtain the value of a single attribute for a

specific device instance. The type of the returned value depends on the definition of the attribute. In this particular case the queried attribute is of type ‘string’. The string is then turned into lower case characters (‘.lower()’), after which the surrounding whitespace is removed (‘.strip()’). The type system could easily infer the resulting type of ‘AIDesc’ based on this chain of expressions. However, it has been declared explicitly as a ‘string’ to illustrate manual typing.

The validation rule for attributes for the ‘findMatchingInstances’ function also applies to ‘.getAttributeData’. If an attribute is not present in one of the device types that make up the type of the instance the method is called upon, the type system will raise an exception.

The next statement checks whether the cleaned-up string in ‘AIDesc’ is equal to ‘spare’. If it is not an element is added to the ‘conds’ list. The object that is added to the list is formed by ‘<i.IOErrorW>’ which is a construction that has not been encountered before. The ‘<...>’ construction represents an operation called quoting, and as the annotations might suggest, it is related to the ‘code of ...’ data type. Quoting takes a valid SCL element, in this case an expression, and lifts it into the meta-domain, turning it into an object. The type of the quoted element depends on the type of the SCL element. If it is an ‘INT’, the quoted type is ‘<INT>’. If it is a ‘STRING’, the outcome will be ‘<STRING>’. And if it concerns a statement rather than an expression or identifier, quoting it will always result in an object of type ‘<Statement>’.

The expression that is being quoted is ‘i.IOErrorW’, which is a structured variable access. The splice of ‘i’ is an example of a regular splice. Regular splices are used to insert an expression or an identifier, as is the case here. Unlike suffixed splices, regular splices can be typed. Regular splices always replace an entire identifier or expression, so the type after splice can be inferred from the type of the spliced object. For example, splicing an object of type ‘<BOOLEAN>’ will always result in an SCL expression of type ‘BOOLEAN’.

Splicing an object of type ‘Device(_)’ requires some additional attention. Evaluating such a splice always results in the instance’s name. This removes the need to explicitly query and splice the name attribute for that device. Type-wise, the SCL equivalent of a ‘Device(_)’ is a ‘STRUCT(_)’, where the fields in the struct depend on the field declared in the device types. As is the case with the attributes, the fields need to exist in each of the device types that compose the ‘Device(_)’ type.

Given that variable ‘i’ is type ‘Device(AnalogInput, DigitalInput)’, the expression ‘i. IOErrorW’ is invalid if ‘IOErrorW’ is not a field in both ‘AnalogInput’ and ‘DigitalInput’, or if their types don’t match. In this particular example, both device types have a field called ‘IOErrorW’ of type ‘BOOLEAN’, so the expression is a valid SCL expression. Given this information, the type of the expression resolves ‘BOOLEAN’. Since it is quoted, the type of the object that is added to the list is ‘<BOOLEAN>’, which is in line with the type of the list. If the type of the list and the added object were incompatible, a type error would occur.

```
$<DB_Global.AI_Sensor_Alarm := $orBuilder(conds,<0>)$;>$;
```

Figure 4.4: Excerpt of Figure 4.1: line 25.

It is now time to look at the use site for the constructed list of ‘<BOOLEAN>’ objects (Lines 24 to 26). The if-statement condition is assumed to be self-explanatory, and is therefore skipped. What happens within the statement is less clear. Due to the quoting and splicing

annotations being used in close proximity to each other it is hard to see, but this is a statement splice. Determining this from the format used in Figure 4.4 can be challenging when one is not familiar with the syntax. Figure 4.5 shows the same logic in an exploded form. The individual components should be easier to comprehend than the compacted version.

```
<BOOLEAN> base = <0>;
<BOOLEAN> orExp = orBuilder(conds,base);
<Statement> stmt = <DB_Global.AI_Sensor_Alarm := $orExp$;>;
$stmt$;
```

Figure 4.5: Exploded version of the statement in Figure 4.4

The first 2 lines of the exploded logic are used to build one large OR-expression.'orBuilder' is a user-defined function. Its implementation will be discussed in section 4.4, but it's function signature is:

$$list(\text{<BOOLEAN>}) \rightarrow \text{<BOOLEAN>} \rightarrow \text{<BOOLEAN>}$$

It takes a list of quoted 'BOOLEAN' objects, combines it with a '<BOOLEAN>' base condition, and turns it into a single '<BOOLEAN>' expression. In this case the base condition is an explicitly typed quoted of 0, which is a valid SCL 'BOOLEAN'. The function is called on the list that was filled earlier, and this base condition. This resulting expression is then spliced into the right-hand side of a statement.

Once the splice has been evaluated, the result is a valid SCL statement, assuming that the type of 'DB_Global.AI_Sensor_Alarm' is a supertype of '<BOOLEAN>'. However, since this code is part of the meta-domain (being inside a meta-level if-statement), it is not allowed to write SCL code directly. Because of this just having a valid SCL statement won't suffice. It has to be turned into a meta-level object so that is can be handled by a meta-level operation. For SCL statements, the only valid option is an object of type '<Statement>'. Such an object is obtained by quoting the SCL statement.

The last line of the exploded statement is used to perform a statement splice. Suffixed splices and regular splices deal with expressions and identifiers. Statement splices can be used to insert quoted SCL statements, i.e. objects of type '<Statement>'. This can be done from within the object-domain, or from within meta-level control statements. In both cases it is only allowed within an SCL Unit's logic section. Splicing from within control statements makes it possible to generate SCL statements based on the data flow in the meta-phase. Without it, the statements would have to be collected in a variable which could then be spliced outside of the control statement.

This covers the dynamic section of the body. The remainder is pure SCL code with the exception of the 'name' splice. This too is a regular splice. Unlike the regular splice encountered earlier, this one does not provide semantic guarantees. This is because the type of 'name' is '<Var(UNKNOWN)>' whereas the type before was 'Device(_)'. Splices have varying outputs and capabilities depending on the object that is spliced in. Splicing '<Var(_)>' ensures that the result of the splice is a syntactically valid identifier. However, since the contained type is 'UNKNOWN' it is impossible to draw such conclusions on the semantics of the splice. Because of

these (un)certainties it can be concluded that '$name$.Pulse_Ack := FALSE;' will always produce syntactically valid code, but the outcome might be semantically invalid. The type system deals with this uncertainty by accepting everything involving an object of type 'UNKNOWN'. As a result, the absence of errors when dealing with objects of type 'UNKNOWN' does not guarantee that there will be no errors in the output.

The Function, and in this case the template as well, is closed with the 'END_FUNCTION' keyword. SCL-T only deals with semantically correct blocks of SCL code. As a result a syntax/-parse error will be reported if this keyword is missing. The same holds for the SCL IF-statement encountered earlier in the example.

This example does not cover the entire SCL-T language. This is not viable in a single template. Since they are crucial for SCL-T, data types and user-defined functions will be discussed in their own sections. Additionally, some more advanced information on splicing is given in section 4.5.

4.3 Data Types

Most data types received no real attention in section 4.2 simply because there is a lot to say about them. SCL-T consists of a variety of data types. The most basic group are the primitive data types. These consist of strings, booleans, integers and doubles. Then there are collections, which group objects of the same type together, and pairs, which can couple objects of different types. Devices and code data types are used to bridge the gap between the meta-domain and the object-level domain.

Each of the supported data types can be used when explicitly defining a variable's type. The allowed values for explicit type declaration are shown below. $\mathbf{DT_n}$ is a declared DeviceType, and **EXP** is any data type supported by SCL, or UNKNOWN. $\mathbf{T_n}$ is a nested data type, which can also be any type from the list.

- bool
- string
- int
- real
- Device($\mathbf{DT_1}$(,$\mathbf{DT_m}$)*)
- list($\mathbf{T_1}$)
- set($\mathbf{T_1}$)
- dict($\mathbf{T_1}$,$\mathbf{T_2}$)
- pair($\mathbf{T_1}$,$\mathbf{T_2}$)
- <**EXP**>
- <Var(**EXP**)>
- <Statement>

The various data types will now be discussed in more detail. The primitive data types are considered to be straightforward, and therefore no additional explanation will be provided. Devices and their functionality has been covered in the previous section. The remaining data types will be covered below. Throughout the sections some of the operations available to each data type will be covered, but this list is incomplete. For the complete list, see Appendix D.

4.3.1 Collections

As the name suggests, collections are objects that group other objects together. The 3 types of collections that are available are: lists, sets, and dictionaries.

Each collection has its own characteristics, which will be discussed individually. One thing they have in common is that they can be iterated over, and that their types are fixed. Since collections are of a fixed type, it is no longer possible to add different types of objects to a single collection. This might feel like a limitation, but it enforces the single responsibility principal, and it improves the overall comprehensibility of the template.

An important thing to realise about collections is that they are mutable objects. This means that elements can be added to or remove from the collection, while the collection object stays the same. This also makes it possible to pass a collection around by reference. Assigning the collection to a different context will be done by a memory pointer, meaning that the underlying representation is the same. Any adaptation made to the collection will affect every variable referencing the same collection. If a copy of a collection is required, the .copy() method can be used. Keep in mind that this will only copy the collection itself, not its children. So any lower level reference is still shared. This can be overcome by calling .deepcopy(). This will recursively copy all elements, creating a full stand-alone copy of the collection. Figure 4.6 represents this.

```
list(list(int) ls1 = [[1]];
list(list(int) ls2 = ls1.copy();
list(list(int) ls3 = ls1.deepcopy();
list(list(int) ls4 = ls1;

print(ls1); // [[1]]
print(ls2); // [[1]]
print(ls3); // [[1]]
print(ls4); // [[1]]

ls1.add([2]);
print(ls1); // [[1],[2]]
print(ls2); // [[1]]
print(ls3); // [[1]]
print(ls4); // [[1],[2]]

ls1[0].add(3);
print(ls1); // [[1,3],[2]]
print(ls2); // [[1,3]]
print(ls3); // [[1]]
print(ls4); // [[1,3],[2]]
```

Figure 4.6: Mutable objects and copy operations

Lists

Lists represent a finite sequence of objects of the same type. They can be initialised empty using '[]' or non-empty using '[obj_1, ...,obj_n]'. Values can be added to the end of a list with the .add(x)

method. Items exist in a list in the order they are added. Values in the list can be accessed by iterating over the list. An example of how to do this is shown in Figure 4.7.

Besides iterating over a list, it is possible to access a specific value in a list. This can be done by obtaining a value by its index. Here it is important to avoid accessing an index which is not part of the list. Doing so will result in an exception. Using the len(l) function it is possible to determine the maximum index available in a list. Index-based access can also be used to override values at a specific index, given that this index is already assigned. Removing an object at a given index will not leave a gap in the indices. Instead all higher indices are shifted left by 1, releasing the last index.

Sets

Sets are like lists, with 2 main exceptions. Adhering to their mathematical definition, sets do not contain duplicate entries. This means that any cannot be 2 entries for which 'e1 is e2' holds. Secondly, sets do not maintain order. Maintaining order would require consensus on what to do with duplicate entries (i.e. discard the new or the old version), and foregoes the idea behind a set. A set is meant to aggregate unique elements, not filter duplicate entries from a list. Given that users cannot depend on the object's order, sets cannot be accessed by index. Figure 4.7 shows the effect of sets on order, and also contains an example of invalid access of a set.

Sets can only be constructed from a list, using 'set(list)'. This can be a new list, or a pre-existing one. The list can be empty or filled. Creating a set from a list creates a shallow copy of the list. As explained earlier, this means that any contained mutable object is shared between the list and the set. Since sets do not allow index-based reference, the only way to change its content is through the .add(obj) and .remove(obj) methods. Besides methods to manipulate a set, and ones to analyse the content of a set, there are also some methods that facilitate set operations: .union(set), .intersection(set), .difference(set), and .symmetric_difference(set). These operations return a new set, so they do not alter the original object. Like with lists, the contained objects of the new set will be shallow copies of their original counterparts.

```
var l = [1,4,2,3,1];
var s = set(l);
for (var e in l) {
    print(e); // prints 1,4,2,3,1
}
for (var e in s) {
    print(e); // prints 1,2,3,4
}
var e = l[1] + s[2]; // Invalid set access
```

Figure 4.7: Differences between lists and sets

Dictionaries

Where lists and sets contain only values, a dictionary consists of key-value pairs. The type of the key and the value can be different. The key is limited to primitive data types. The value can take on any type. Each key is unique, and adding the same key again with a different value will override the old value. Accessing a key that does not exist will result in a runtime exception, similar to trying to obtain a list item at an out-of-bound index. In order to make sure that these exceptions do not occur at runtime, one can check map keys by calling 'obj in dict'.

This weak coercion towards defensive programming adds some additional steps for the developer. However, it ensures that the optional nature of an object has been considered, and that the templates are designed to handle these scenarios accordingly. If the object should be present, and the template should fail upon its absence, the checks can be omitted. If the goal is not to access a specific element, but go over all the entries of the dictionary, it is possible to iterate over it.

Dictionaries are initialised using '{}'. None-empty dictionaries can be constructed with '{key_1 : $value_1$, ..., key_n : $value_n$}'. To add a value to a dictionary there are 2 approaches. The first one is to call the '.add(key,value)' method. The alternative way is to use key-based assignment, i.e. 'dict[key] = value'. Both approaches can also be utilised to override an existing value in the map. '.remove(key)' can be used to remove an entry from the dictionary. Execution will fail if the key does not exist.

There are various ways to go over all the entries in a dictionary. The first one is by calling the .keys() method, and accessing the map based on each found key. If the key is not important, it is also possible to obtain a list of all values by calling .values() method. The preferred method to iterate over each key-value pair is to call the .items() method. This returns a list of pairs. Each pair consists of the key and the value. An example of each of these approaches is shown in Figure 4.8.

```
var dict = {"a":1, "b":2};
for (var key in dict.keys()) {
    var value = dict[key];
    ...
}
for (var value in dict.values()) {
    ...
}
for (var entry in dict.items()) {
    var key = entry.left();
    var value = entry.right();
    ...
}
```

Figure 4.8: Methods to iterate over dictionaries

4.3.2 Pairs

Unlike collections, pairs are of a fixed length, and can contain objects of varying types. Additionally, pairs are immutable objects. The main function of pairs is to create compound objects of elements that are closely related. Pairs can be nested to create larger tree-like structures, but each pair will always consist of a left and a right element. Figure 4.9 shows an example of a nested pair. While verbose, restricting the size to two elements, with explicit access methods allows the type system to accurately resolve the type of each access.

An example of a usage for pairs is the generateIfElse function. This standard function is covered in more detail in section 4.5, but its purpose is to generate a syntactically correct SCL IF-statement. In order to do this, it takes a list of pairs of conditions and lists of statements, i.e. 'list(pair(<BOOLEAN>, <Statement>))'. The reason to go with a pair is that each condition is linked to a list of a statements. Technically, it would be possible to model the function using two separate lists: one containing objects of type '<BOOLEAN>', and the other containing lists of '<Statement>'. However, the developer would have to make sure that elements of both lists are in the correct indices, and the function would have to validate that both lists are equally long. Using a list of pairs solves both issues.

On the one hand, pairs can be useful in function parameters to ensure that elements that belong together are passed to the function together. On the other hand, they can be utilised to return 'multiple' objects. An example of this is shown in Figure 4.9.

```
def pair(string, pair(bool,int)) foo() {
    return ("bar",(true, 1));
}
```

Figure 4.9: Nested pairs

Elements of a pair can be accessed by calling .left(), and .right() on the object. As their names suggest, they return the left and right element of the pair, respectively. There are no additional operations that can be applied to pairs, limiting their usability outside of function-related contexts.

4.3.3 Code Data Type

The purpose of the code data type is to make it possible to pass syntactically correct object-level code around as a meta-level object. Where the other data types have a purpose at meta-level, even Devices, an object of the code data type can only be utilised in an object-level context. In that sense, it is not like the other types, even though it is part of the meta-domain. The code data type is crucial for SCL-T to achieve its level of static guarantees. The quoting and splicing mechanisms that were be discussed in section 4.2 have a very strong relation to the code data type.

SCL-T differentiates between 3 types of code objects. The first one is '<Var(T)>', used to enforce the notion that it concerns a quoted identifier. Secondly, there is '<T>', which is the

quoted equivalent of an SCL expression of type T. The last one is '<Statement>'. These objects can contain valid SCL statements.

Code type: '<Var(T)>' and '<T>'

The reason why '<Var(T)>' and '<T>' are mentioned together is that they have a lot in common. When used in the context of an expression, objects of these types are interchangeable. Expressions work on values and are not concerned with whether this is a variable, a compound expression, or only a value. Instead it is only important what the type is. This is different for assignment sites and explicit variable references (e.g. function names and output parameters in SCL function calls). In those cases the code can only be valid if the spliced in element is certain to be an identifier. This is what the '<Var(T)>' type is really for. It ensures that the object in question contains a valid SCL identifier. Figure 4.10 shows what this means in practice.

```
var id = <A>;                  //id : <Var(BOOLEAN)>
var exp = <TRUE OR FALSE>; //exp : <BOOLEAN>

$id$  := $id$;
$id$  := $exp$;
$exp$ := $id$;  //invalid
$exp$ := $exp$; //invalid
```

Figure 4.10: Usage of <Var(T)>

When an identifier is quoted, its SCL type T is inferred from the current SCL context, and it is transformed into a code data type of the form '<Var(T)>'. If the identifier is unknown within the local context, a typing error will be reported. '<Var(T)>' is a subtype of '<T>' and therefore are allowed to be assigned to value of type '<T>', but not vice versa.

Nearly all operations performed on variables of type '<Var(T)>' will not return a value of type '<Var(_)>'. The main reason is that the return value is no longer an identifier, or that it cannot be guaranteed to be an identifier. In case of the latter, type safety has been chosen over freedom, but it could lead to unexpected type errors. The one operation for which this does not hold is a suffixed splice. Suffixing a splice of an object of type '<Var(_)>' allows a new variable to formed. The suffix has to adhere to the following regular expression '([_]?[a-zA-Z0-9])+' and there can't be any whitespace between the splice and the suffix. Suffixed splicing is only allowed on objects of type '<Var(_)>', and will always generate an object of type 'Var(UNKNOWN)'.

If contextually there is no reason to use an identifier, declaring a parameter as '<Var(T)>' will only form a self-imposed restriction as these variables can't be assigned values of type '<T>'. In those cases there is no reason to use '<Var(T)>' over '<T>', but the user is free to use the '<Var(T)>' nonetheless.

Code type: '<Statement>'

There might be scenarios in which splicing an expression or variable is not sufficient, for example if entirely different logic should be inserted. A statement code type, '<Statement>', was added to support these abstractions.

SCL does not have a block which takes exactly one statement. This means that whenever a statement can be spliced in, it is always syntactically valid to splice in multiple statements as well. As a result, the '<Statement>' can be an arbitrary number of statements, even zero.

Initialising '<Statement>' variables can be done using an explicit statement, or simply '<>', which is equivalent to zero statements. An empty statement list allows statements to be constructed based on a for-loop without having to write special constructs for the first element. '<Statement>' objects can be appended to each other using the following format '<*s*1; *s*2;>'. It will return an object of type '<Statement>', as shown in Figure 4.11.

```
var stmts = <>; //stmts : <Statement>
for (var i : [1,2]) {
    stmts = <$stmts$; A_$i$ := i;>;
}
```

Figure 4.11: Statement concatenation through splicing

Since a '<Statement>' object is equal to zero or more SCL statements, the decision was made to disallow lists of '<Statement>' objects (i.e. 'list(<Statement>)'). Splicing a list of '<Statement>' objects or splicing a single compound '<Statement>' would be semantically equivalent, so there is no technical reason behind this decision. However, one of the goals of the new language is to prevent ambiguous approaches to the same problem, so to standardise the way '<Statement>' objects are spliced into the object-level code, specifying lists of these objects is disallowed.

4.3.4 UNKNOWN Data Type

The 'UNKNOWN' data type was already encountered in the example in section 4.2. It is a self-defined object-level data type that indicates that the type system can't draw any conclusions on the semantic validity. The best example of this is an identifier passed to the template from the calling site. It can be guaranteed that it is an identifier based on the specifications file's metadata, but it is impossible to determine the corresponding type without knowing the content of the specs. Since the content of the variable can change the outcome, there are scenarios in which the resulting code is semantically invalid. However, syntactic correctness is still guaranteed.

Any object-level operation performed with an object of type 'UNKNOWN' is also 'UNKNOWN'. For example, an assignment to a value of type 'UNKNOWN' is always valid. Building an expression where one of the terms is 'UNKNOWN' will have the return type 'UNKNOWN'. Lastly, calling a function of which either the function name or any of the parameters is 'UNKNOWN', will turn the entire call into an 'UNKNOWN' entity. Figure 4.12 shows the

propagating effect of an ‘UNKNOWN’ data type. The orange underline is put into place to indicate that the referenced variable is of type ‘UNKNOWN’.

```
var a = <A>; //a : <Var(BOOLEAN)>
var _a = <$a$_SFXD>; //_a : <Var(UNKNOWN)>
var o1 = <$a$ AND TRUE>; //o1 : <BOOLEAN>
var o2 = <$_a$ AND TRUE>; //o2 : <UNKNOWN>
```

Figure 4.12: UNKNOWN type propagation

4.4 Functions

Function calls have been touched upon in the example of section 4.2. However, since the example only covers the template format and features, function declarations and their features/rules were not covered. Functions can be used to specify commonly used functionality. They can be called from within the template, or from within another function. By using functions it is possible to reuse code, reducing the size of templates, and improving the overall maintainability. The ‘orBuilder’ used in section 4.2 is a good example of this. Its implementation is shown in Figure 4.13.

```
def <BOOLEAN> orBuilder(list(<BOOLEAN>) conditions,
                                    <BOOLEAN> base) {
    var cnd = base;
    for(var condition in conditions) {
        cnd = <$condition$ OR $cnd$>;
    }
    return cnd;
}
```

Figure 4.13: Programmatic expression formation

Functions need to have a return type. This type can be any from the list in section 4.3, or type ‘void’. Void indicates that there is no return value. Since the only mutable types are collections, the void data type only makes sense when the intention is to modify an object of these types. In all other cases the function will not have any side-effects, meaning that the function does nothing.

Functions can take zero or more parameters. These parameters are meant to pass information from the calling site to the function. Like template parameters, function parameters are constant. While they cannot be reassigned in the context of the function, their corresponding values might be mutable. This means that the mutable nature of collections also applies within functions. Any changes made in a function to a list which is passed as a parameter will affect the calling site.

The function header needs to be explicitly typed, both in parameters and return type. The body of a function follows the same rules as statements do within the template itself. Functions

consist of pure meta-level code. It is possible to define code type objects, but it is not allowed to splice the code, or write SCL code altogether.

Functions need to be defined in different files than the templates, which is a measure to enforce separation of concerns. Functions are available across the entire project. It is no longer needed to import a package to get access to a function. Function overloading is only allowed based on varying the number of parameters. Overloading based on parameter types or the return type is not supported. Since SCL-T scans the entire project, this means that the combination of function-name and amount of parameters has to be unique. An example of acceptable and invalid function overloading is visible in Figure 4.14.

```
def int min(int i1, int i2) {
    if (i1 < i2) {
        return i1;
    }
    return i2;
}

def int min(int i1, int i2, int i3) {
    return min(min(i1,i2),i3);
}

def real min(real r1, real r2) { //duplicate min_2
    if (r1 < r2) {
        return r1;
    }
    return r2;
}
```

Figure 4.14: Valid and invalid function overloading

4.5 Splicing

Much of the detail of splicing has been covered in section 4.2 and in the section on code data types. However, there are some aspects that have not been addressed. In general lines, splicing can be divided into 2 groups: Statements and Expressions/Identifiers.

Objects of type '<Statement>' are the only ones that are eligible for statement splicing. Statement splicing can be done anywhere when it concerns statement concatenation (as introduced in section 4.3.3). Splicing statements into the object-level domain is only allowed inside SCL Units' logic sections. Splicing any type other than <Statement> is considered a Expression/Identifier splice. As discussed in section 4.2, this is rather straightforward for 'code of ...' data types. If the type of a code object is '< T>' then the spliced type is 'T'. For the other data

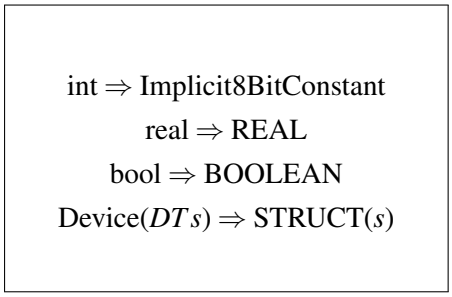

Figure 4.15: Mappings from meta-level objects to object-level types

types, it requires a more elaborate mapping as was the case for 'Device(_)'. This mapping is displayed in Figure 4.15.

In the mapping for devices DTs is a list of device types, and s is a scope containing declarations for the intersection of the fields in each of the device types in DTs. Collections, strings and pairs are missing from the mapping, as no sensible mapping exists. For this reason, these data types are not eligible for splicing.

Precendence-wise, splices are always isolated, meaning that they are not affected by the context they are spliced into. An example of this is '$< 1 + 2 >$ * 3 '. If the surrounding precedence rules were applied post-splicing, the result would be equivalent to '(1 + (2 * 3))'. This would imply that the outcome of a splice depends on the context where it is inserted into. This could lead to unexpected results (as discussed in section 3.6), and for that reason, the decision was made to isolate splices by giving them the highest level of precedence. The above example is therefore equal to '((1 + 2) * 3)'.

Through quoting and splicing, arbitrary code concatenation is a thing of the past. The new approach provides syntactic security, and, to a large extent, semantic certainty. The 'orBuilder' function introduced in Figure 4.13 is the SCL-T equivalent of the UAB code in Figure 4.16. The new function does not just ensure a correct expression syntactically. It also provides that guarantee that the resulting expression is a 'BOOLEAN' expression.

```
for i in values:
    plugin.writeSiemensLogic('''$i$ OR ''')
plugin.writeSiemensLogic('''0;''')
```

Figure 4.16: Arbitrary code constructs

Even though it prevents a lot of erroneous constructs, eliminating arbitrary code concatenation has made some things more difficult. Since statements are meant to be syntactically correct, dynamically formed IF-statements and CASE-statements became a problem. Since the amount of substatements could vary based on the input, the syntax could as well. Validating the dynamically formed substatements would be difficult to do, especially without implementing some form of data-flow analysis. On the other hand removing the possibility to generate such statements

dynamically would be unacceptable. SCL-T offers a middle-ground.

Fixed format IF-statements and CASE-statements can still be written in the native syntax. For the other situations, there now is a generateIfElse function that returns an IF-ELSIF-ELSE statement and a generateIf that returns an IF-ELSIF statement. The functions take a parameter of type 'list(pair(<BOOLEAN>,<Statement>))', and in case of the formIfElse, another parameter of '<Statement>' containing the statements for the ELSE block. Similarly, there is a case creation statement with and without a default value.

Figure 4.17 shows an real-life example of how these helper functions can be put to use. Assuming variable 'name' has a value of 'PCO_10', and 'dev' contains 2 devices called 'Slave1' and 'Slave2', the outcome of this function is visible in Figure 4.18. Since the amount of ELSIF blocks is dynamic, the would be impossible to construct this in any other way.

```
var cond = <DB_Global.RealPowerCut_temp
            AND NOT DB_Global.Pulse_Ack>;
var stmt = <$name$.AuAlAck := FALSE;
            $name$.AuRStart := FALSE;>;

list(pair(<BOOLEAN>,<Statement>)) l = [(cond,stmt)];
for var inst in dev {
    cond = <$inst$.Pulse_Ack>;
    stmt = <$inst$.AuAlAck := FALSE;
            $inst$.AuRStart := FALSE;>;
    l.add((cond,stmt));
}

generateIf(l);
```

Figure 4.17: Example of 'generateIfElse' function

```
IF DB_Global.RealPowerCut_temp AND NOT DB_Global.Pulse_Ack THEN
    PCO_10.AuAlAck := FALSE;
    PCO_10.AuRStart := FALSE;
ELSIF Slave1.Pulse_Ack THEN
    Slave1.AuAlAck := FALSE;
    Slave1.AuRStart := FALSE;
ELSIF Slave2.Pulse_Ack THEN
    Slave2.AuAlAck := FALSE;
    Slave2.AuRStart := FALSE;
END_IF
```

Figure 4.18: Output of 'generateIfElse' function in Figure 4.17

Chapter 5

Language Implementation

The features of the SCL-T language were introduced in chapter 4. While this focuses on SCL-T from the developers' point of view, it does not provide any insight on the internals of the language. This chapter goes over how SCL-T is implemented, and is therefore more interesting from a language designer's perspective.

SCL-T is composed of various languages, a concept which is discussed in section 5.1. The languages that make up SCL-T will be discussed individually. Both the syntax and the type rules of the components will be covered. However, before zooming in on the languages, the underlying type system also deserves some attention.

5.1 Architecture

The architecture behind SCL-T is similar to UAB's original one. There's the UAB native back-end (UAB Core), the CPC library and specification files, and a series of components related to the supported languages. The new architecture differs mainly on the last part. The original architecture had a single language for all supported PLCs. Language-specific plugins added specialised functionality by means of functions. The new architecture has a single dedicated language per supported PLC language. This tailored language is a composition of the object language, i.e. the PLC language, and a meta-language. The composition for SCL-T is shown in Figure 5.1. The composition happens in a third language, the mixin-language, which dictates how the languages are intertwined. Additionally, it can add meta-programming features based on the target language, and vice versa.

Having a language that is tailored to a specific target PLC makes it easier to support highly specialised features like semantic validation. Furthermore, splitting the meta-language from the object-language in a strict manner allows the languages to be maintained separately. The meta-language is much more likely to evolve over time compared to the PLC language. Having a hard cut between the meta-language and the object language allows them to evolve separately. In case a change is made to one of the languages, at least one, and at most two components have to be adapted. For example, a change made to the meta-language can result in the SLC-T language needing to be adapted. However, it will never result in a change having to be made in

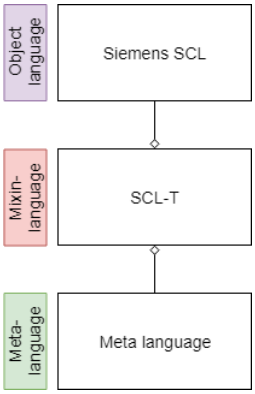

Figure 5.1: SCL-T's language composition. The arrows indicate the direction of composition.

the coupled object languages. Similarly, a change made in SCL will at most result in a change in the associated mixin-language, but it will never influence the meta-language.

The new architecture might seem like a step backwards from UAB's original architecture due to the fact that each target language now has to have its own dedicated meta-programming language. In reality this is not the case. As mentioned before, a dedicated language allows for more specialised features to be supported, making the language more powerful. Additionally, the meta-language is completely language-agnostic. This means that it can be reused among the various PLC languages, as displayed in Figure 5.2. Reusing the same meta-language not only makes it easier on the developers who have familiarised themselves with its capabilities, it can also lower the hurdle of adding a new PLC language.

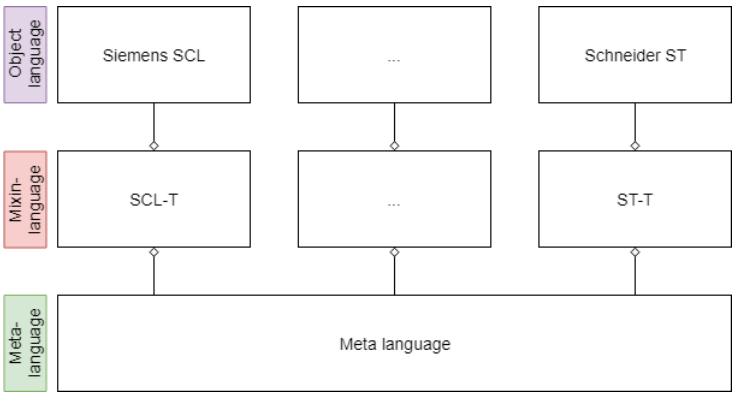

Figure 5.2: Meta-language reuse among various PLC languages

Admittedly, there are also downsides to a split approach. Having the meta-language heavily embedded in the object language generally leads to less complex language definitions and more freedom in meta capabilities per language. However, there is also a significant cost associated with this design. For every PLC language the meta-language, and its embedding, will have to be designed prior to implementing the language. In the new setup the object language can

be designed without concerning oneself with the implementation of the meta-language at all. How and where the meta-language is embedded can be decided and implemented later on in the mixin-language. Additionally, since the meta-language has been pre-designed, this decision is generally limited to defining the interfaces between the object and the meta-language, and how to map meta-types to object types.

Furthermore, it is debatable whether the decreased freedom in language-specific meta-features still applies to the suggested architecture. Specialised meta-level features can be added in the mixin-language. Additionally, since components are interchangeable, it would also be possible to use an alternative, more specialised meta-language. This goes against the idea of having a shared meta-language, but the language composition makes it possible. An example of such a composition can be seen in Figure 5.3. This level of flexibility, while at the same time ensuring code reuse, would be difficult to achieve in a scenario where the meta-language is embedded.

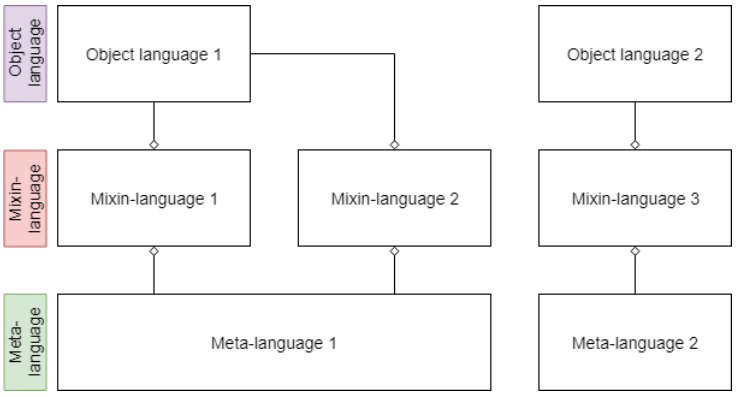

Figure 5.3: Language composition based on varying the meta-language or the mixin-language

5.2 Type System

The type system is modelled based on scope graphs [3]. Scope graphs provide a tangible idea of a type system, and make it possible to define a representation without having the actual source. This allows UAB to inject scopes for library components (e.g. baseline objects, standard functions) to facilitate type checking, without having to include the actual objects. Since the implementations are not exposed to the developers, they cannot alter them, adding to the level of standardisation.

Additionally, the languages consists of various distinct contexts (e.g. devices, symbols, functions). Scope graphs make it possible to handle these separately. This isolates the resources, which makes the rules easier to comprehend. Whenever contexts do depend on each other, scope graphs make it easy to relate them. By creating an edge from one to the other, scopes can be linked, allowing definitions to be resolved even though they are part of a different scope.

One recurring element in the type rules is the interaction with the context (ctx). Some rules have constraints based on the context where they are used. For completeness they have been

$$C \vdash t : T \iff \text{Returns type } T \text{ of term } t \text{ given the context } C$$

$$C \vdash t :: \text{OK} \,||\, \text{NOK} \iff \text{Indicates whether typing term } t \text{ in context } C \text{ is successful or not}$$

$$C \vdash t \rightarrow S \iff \text{Typing term } t \text{ in context } C \text{ returns scope } S$$

$$\nabla S \iff \text{Returns a new scope } S$$

$$S_1 \overrightarrow{E} S_2 \iff \text{Create edge } E \text{ from scope } S_1 \text{ to scope } S_2$$

$$T_1 <: T_2 \iff \text{Verifies that } T_1 \text{ is a subtype of } T_2$$

$$S \overrightarrow{:} NS(t) : T \iff \text{Declares term } t \text{ as a member of namespace } NS \text{ with type } T \text{ in scope } S$$

$$S \Leftarrow NS(t) : T \iff \text{Resolve term } t \text{ within namespace } NS \text{ in scope } S \text{ to type } T$$

Figure 5.4: Format and meaning of the various typing rules used in this chapter.

added to the type rules, but in practice it may be more sensible to perform such checks at a different stage in the validation process. Since none of the context-based rules has any interaction with the other elements from the type system they could be performed in a standalone manner. Executing them separately is also more desirable, as making them part of the type system requires the context information to either be available or be passed down. In practice this is a cumbersome process. Since the checks do not need any other typing information, they can be performed during a stage where ASTs can freely be traversed. This makes the implementation of these checks much simpler, without losing any functionality.

The format for the rules is explained in Figure 5.4. Whichever rule is applied is inferred based on the context and the term that is evaluated. However, there are rules that are ambiguous in the terms they are applied to. Their ambiguity is resolved within the corresponding premise. Rule 1 can only succeed if rule 2 fails, and vice versa. There is also a scenario in which both rules fail, which is the only scenario in which the entire rule call fails. If either of the ambiguous rules succeeds, the call itself is also successful. If no rule has been specified for a specific term, it will also default to failure.

5.3 Object language

The object language is the core component of SCL-T. It specifies the target language's syntax and type rules. In case of SCL-T, the object language is a direct implementation of Siemens' SCL language.

5.3.1 Syntax

Since the syntax for SCL is documented in the official documentation[1], it will not be covered in great detail. During the implementation and testing phase of SCL-T it became apparent that the documentation is not entirely correct. Since there is no documentation on SCL that is up-to-date, the decision was made to model SCL after the most recently documented version. The encountered errata are shown below.

- The documentation misses any mention of Block Attributes being allowed before the Declaration Section for each of the SCL Units.
- The documentation states that Declaration Sections and Statement Sections cannot be empty whereas the compiler accepts it. Furthermore each of the Subsections of the Declaration Sections can be empty.
- The documentation fails to mention that Block attributes can be terminated with a semicolon.

These errors only concern non-functional differences, limiting their impact. However, the inconsistency between the documented syntax and the implemented syntax can lead to template files being flagged as incorrect while they might compile just fine. The problem is that without information on what the implemented syntax looks like, one can only guess how to fix the rules.

5.3.2 Type Rules

While the syntax of SCL has its own chapter in the documentation, the semantics of the language are scattered across the entire document. Most of the rules are simple and easy to transform into a variant suited for scope graphs. However, there are a few linguistic elements that are either complex to model, or for which a naive approach is undesirable. These elements will be covered in detail below. The remainder of the rules can be easily conjured based on the original documentation and following the examples for the more complex rules.

Symbols table

The symbols table deserves a dedicated section when it comes to typing. While it is not particularly difficult, its behaviour is mostly implicit and can be difficult to understand without some explanation. Syntactically, each line in the symbols table is 'identical'. There is no differentiation among the lines. Semantically this is not the case. Obviously each declaration in the table has its own type, but this can easily be covered within a single type rule. The interesting semantic difference is the one between global variables and unit declarations. Global variable declarations are mappings to a memory location. On the other hand, unit declarations provide symbolic names to the absolute block identifiers, and indicate the type of the unit. The rules for both types are included in Figure 5.5.

[1] https://cache.industry.siemens.com/dl/files/793/5581793/att_66783/v1/SCL_e.pdf

$$(STMT_{meta})\ \frac{\text{isMemoryLocation}(a) \qquad S_g \overset{?}{\rightarrow} VAR(s)\ :\ T}{S_g, S_u \vdash s,\ a,\ T\ ::\ \text{OK}}$$

$$(STMT_{meta})\ \frac{\text{isUnit}(a) \qquad S_u \overset{?}{\rightarrow} SYMBOL(s)\ :\ (s, a, T)}{S_g, S_u \vdash s,\ a,\ T\ ::\ \text{OK}}$$

Figure 5.5: Type rules for variable declaration

For global variables, the symbols table is the declaration site, whereas for units, it is merely a declaration of the symbolic name. The differences in semantics also impacts how they are used in the type system. Since variables have a declarative meaning, they should be included in SCL's global resolution scope, S_g. This scope spans the entire SCL project. Since the variables are part of the global scope the type system can discover them when resolving variables. Contrary, unit blocks only use the symbols table to lookup their own alternative name, and validate their own types. They provide no additional information to the type system directly. Similarly, references to unit blocks are also not concerned with what was declared in the symbols table. Therefore, the unit declarations from the symbols table can be included as a stand-alone scope, S_u.

The reason to disregard the declaration in the symbols table as a (preliminary) declaration site is that it would give the users a false sense of security. If no actual implementation of a unit is found, no validation could be performed on any reference to that block. Additionally, since the existence of a declared unit is required for successful compilation, it is justified to expect the file to exist.

Units

Units are the builing blocks for SCL programs. A program can consist of various files, and each file can have zero or more SCL program units. Upon compilation, the whole project is compiled, regardless of the underlying file structure. From a type-system point of view, units are the highest level objects, together with global variables, which were discussed earlier.

Units are declared within the global scope, which is also the parent scope used to validate a unit's local context. Since unit declarations can be done using both absolute and symbolic names, the scope rules for units require access to the S_u scope as well. Organization Blocks are only called by the operating system, and therefore cannot be referred to. The only reason to store the declaration is to detect/prevent duplicate entries. This is also why the type of the declaration is empty. Data Blocks and User Defined Datatypes (UDTs) are both Struct-based. Functions and Function Blocks follow the pattern of the Organization Block with the exception that the I/O information for these blocks is required to validate calls. These function calls will be discussed in section 5.3.2.

$$\text{(SCL\_PROGRAM)}\ \frac{S_g, S_u \vdash \mathit{unit} :: \mathit{OK} \qquad S_g, S_u \vdash \mathit{units} :: \text{OK}}{S_g, S_u \vdash [\mathit{unit}|\mathit{units}] :: \text{OK}}$$

$$\text{(SCL\_PROGRAM)}\ \frac{}{S_g, S_u \vdash [] :: \text{OK}}$$

$$\text{(SCL\_UNIT)}\ \frac{\begin{array}{c}\nabla S_l \qquad S_l \overrightarrow{T} S_g \qquad S_u \Leftarrow \text{SYMBOL}(f) : (sn, an, t) \qquad S_l \vdash ds \rightarrow S_l' \\ S_u, S_l' \vdash \mathit{stmts} :: \text{OK} \qquad S_g \overrightarrow{:} \text{UNIT}(sn),\text{UNIT}(an) : \text{OrgBlock}()\end{array}}{S_g, S_u \vdash \text{ORG\_BLOCK } f\ ds \text{ BEGIN } \mathit{stmts} \text{ END\_OBLOCK} :: \text{OK}}$$

$$\text{(SCL\_UNIT)}\ \frac{\begin{array}{c}\nabla S_l \\ \nabla S_{IOs} \qquad S_l \overrightarrow{T} S_g \qquad S_u \Leftarrow \text{SYMBOL}(f) : (sn, an, t) \qquad S_g \vdash t : T \\ S_l \overrightarrow{:} \text{VAR}(sn),\text{VAR}(an) : T \qquad S_l \vdash ds \rightarrow S_l' \qquad \text{extractIOs}(ds, S_{IOs}) \\ S_u, S_l' \vdash \mathit{stmts} :: \text{OK} \qquad S_g \overrightarrow{:} \text{UNIT}(sn),\text{UNIT}(an) : \text{Function}(S_{IOs}, T)\end{array}}{S_g, S_u \vdash \text{FUNCTION } f : t\ ds \text{ BEGIN } \mathit{stmts} \text{ END\_FUNCTION} :: \text{OK}}$$

$$\text{(SCL\_UNIT)}\ \frac{\begin{array}{c}\nabla S_l \qquad \nabla S_{IOs} \qquad S_l \overrightarrow{T} S_g \qquad S_u \Leftarrow \text{SYMBOL}(f) : (sn, an, t) \\ S_l \vdash ds \rightarrow S_l' \qquad \text{extractIOs}(ds, S_{IOs}) \\ S_u, S_l' \vdash \mathit{stmts} :: \text{OK} \qquad S_g \overrightarrow{:} \text{UNIT}(sn),\text{UNIT}(an) : \text{FuncBlock}(S_{IOs})\end{array}}{S_g, S_u \vdash \text{FUNCTION\_BLOCK } f\ ds \text{ BEGIN } \mathit{stmts} \text{ END\_FBLOCK} :: \text{OK}}$$

$$\text{(SCL\_UNIT)}\ \frac{\begin{array}{c}S_u \Leftarrow \text{SYMBOL}(t) : (sn, an, t) \\ S_g \vdash s : T \qquad S_g \overrightarrow{:} \text{UNIT}(sn),\text{UNIT}(an) : \text{Type}(T)\end{array}}{S_g, S_u \vdash \text{TYPE } t\ s \text{ END\_TYPE} :: \text{OK}}$$

$$\text{(SCL\_UNIT)}\ \frac{\begin{array}{c}S_g \vdash ds : \text{STRUCT}(tm, S_s) \qquad S_u, S_s \vdash \mathit{stmts} :: \text{OK} \\ S_g \overrightarrow{:} \text{UNIT}(sn),\text{UNIT}(an) : \text{DataBlock}(\text{STRUCT}(tm, S_s))\end{array}}{S_g, S_s \vdash \text{DATA\_BLOCK } d\ ds \text{ BEGIN } as \text{ END\_DATA\_BLOCK} :: \text{OK}}$$

Figure 5.6: Type rules for units

One element which is not covered in these scope rules is the fact that SCL enforces Unit order within a program, but not between programs. The reason why this is not covered is because there is no graph that resolves this constraint. SCL programs are anonymous within the global scope, meaning that units are all declared at the same level, regardless of the file they are in. As a result order will be enforced for all, or for none. Given the tendency to define a single unit per template, the decision was made not to enforce order. This will result in false negatives if the user defines multiple units that reference each other within the same SCL program. The alternative is that the evaluation order would impact the outcome of the type system, which is undesirable.

Function and Function Block Calls

As spoilered when discussing the rules for units, the rules for Functions and Function Blocks are different. This is related to the effect of the declaration on the validation approach of a call to these units. If all declarations regardless of usage were declared within the same scope, extracting only the input and output variables would have required some effort. The information would be present in the units' scopes, but littered with declarations that are of no concern to a function call. A more elaborate distinction in the variable namespace could solve this easily, e.g. adding 'INPUT(x)' and 'OUTPUT(x)', but this would make the variable validation rules unnecessarily complex, and they have to resolve against multiple namespaces.

Instead, the decision was made to build up a second scope which contains only the input and output variables. The new scope is included in the type of a stored unit declaration, so that it is available whenever the unit is referenced. Besides being declared in their own scope, the variables are also still declared within the Function's local scope for internal validation. Unlike the local scope, the IO scope is unordered, as the order in which variables are declared has no meaning outside of the local context.

Now that the non-standard rules regarding the declaration site have been addressed, it is time to look at the validation of a Function (Block) call. Validating the calling site requires an approach from multiple directions. This is caused by the fact that parameters are not passed by order, but by name. Rather than simply having to compare a list of types on both sides, it now is required to validate that all referenced names are valid (checking for correctness), but also that all declared names have been referenced (checking for completeness). This means that in one stage an iteration has to be done over all referenced names, and checking whether they exist in the declaration and that their types match. In a different stage the IO scope of the callee's declaration is traversed, where each declaration is checked for presence in the function call. These rules are displayed in Figure 5.7. The details of 'validateCompleteness' and 'validateCorrectness' have been left out for terseness. These functions consist mostly of iterating over and comparing entries.

Structs

In general SCL-T's data types require no explanation. Except for Arrays and Structs they are terminals, making their declaration simple. Arrays are composed of other data types, so require

$$(STMT_{SCL})\ \frac{S_l \Leftarrow \text{VAR}(x) : \text{FuncBlock}(S_{IOs}) \quad \text{validateCompleteness}(S_{IOs}, params) \quad \text{validateCorrectness}(params, S_{IOs})}{S_u, S_l \vdash x(params);\ ::\ \text{OK}}$$

$$(EXP_{SCL})\ \frac{S_l \Leftarrow \text{UNIT}(fc) : \text{Function}(S_{IOs}, T) \quad \text{validateCompleteness}(S_{IOs}, params) \quad \text{validateCorrectness}(params, S_{IOs})}{S_l \vdash fc(params) : T}$$

Figure 5.7: Type rules for function (block) calls

a recursive call which is also straight-forward. Structs are a different story, as they represent a complex nested object.

Structs are data types which can most easily be described using a sub-scope. This sub-scope contains the underlying declarations of the struct. When resolving struct access, this is done by looking for the accessed field within this sub-scope. Using scope graphs, structs become relatively easy to implement in the type system. One thing that has to be taken into account is the fact that initial assignments to fields in a struct are validated against the outermost scope. This means that the outermost scope has to be passed along with the declaration of the struct and any lower-level struct. Since this differs from how declarations are normally validated, struct declarations require their own set of rules. The scope rules for structs are shown in Figure 5.8.

While scope graphs make it easy to model the nested nature of structs, they affect how to validate complete structures assignment. SCL allows one struct to be assigned to the other if their fields match in names, type, and order. It is possible to verify this using the scope graphs from both structs, but this would require the scopes to maintain order, and they would have to be traversed in entirety. Rather than doing this, the decision was made to also include a term-based representation of the struct in the computed data type. This term can easily be compared to the term of the other struct to check for equality. The declaration of a struct is therefore typed 'STRUCT(tm,s_s)', where tm is the term representation of the struct, and s_s is the struct's scope representation. The 'structToTerm' function takes the struct's AST and desugars every initialised declaration into an uninitiliased declaration. By removing the initially assigned values, a simple equality check between the terms suffices, as displayed in the last rule of Figure 5.8.

5.4 Meta-Language

Unlike Siemens' SCL language, the meta-language is a newly designed language. It provides the scripting functionality to SCL-T. The language defines the data types that can be used to manipulate templates and blocks of object-level code.

Since the meta-language is meant to be reusable between various target languages, it is

$$(DT_{SCL})\ \frac{\nabla S_s \qquad S_c, S_s \vdash decls \rightarrow S'_s \qquad \text{structToTerm}(decls) \rightarrow tm}{S_l \vdash \text{STRUCT } decls \text{ END\_STRUCT} : \text{STRUCT}(tm, s'_s)}$$

$$(STRUCT_{SCL})\ \frac{S_l, S_s \vdash decl \rightarrow S'_s \qquad S_l, S'_s \vdash decls \rightarrow S''_s}{S_l, S_s \vdash [decl|decls] \rightarrow S''_s}$$

$$(STRUCT_{SCL})\ \frac{}{S_l, S_s \vdash [] \rightarrow S_l}$$

$$(SDECL_{SCL})\ \frac{\begin{array}{ccc}\nabla S'_s & S'_s \overrightarrow{P} S_s & S_l \vdash t : T_1 \\ S_l \vdash v : T_2 & T_2 <: T_1 & S_s \overrightarrow{:} \text{VAR}(x) : T_1\end{array}}{S_l, S_s \vdash x : t = v; \rightarrow S'_l}$$

$$(SDECL_{SCL})\ \frac{\nabla S'_s \qquad S'_s \overrightarrow{P} S_s \qquad S_l \vdash t : T \qquad S_s \overrightarrow{:} \text{VAR}(x) : T}{S_l, S_s \vdash x : t; \rightarrow S'_l}$$

$$(TYPE_{SCL})\ \frac{tm_1 == tm_2}{\text{STRUCT}(tm_{1,}) <: \text{STRUCT}(tm_2)}$$

Figure 5.8: Type rules related to Structs

important to make sure that it does not contain any language-specific elements for the object language.

5.4.1 Syntax

Most of the interesting aspects of the meta-language's syntax have already been covered in chapter 4. There are 3 rules that require some explanation, the rest has been deemed straight-forward and uninteresting. For this reason the full syntax description has been moved to Appendix E.

The syntax rules worth addressing are repeated in Figure 5.9. The <Program> rule allows the meta-language to be used in a stand-alone context. The decision was made not to allow function definitions to be inlined with statements. While the value of a stand-alone meta-language is small, it helps in ensuring that there is no relation to any object language.

Allowing the <Call> non-terminal at statement level makes it possible to perform function and method calls to be done without assigning the output to a variable. This makes sense when the return type is void, but can also make sense when the call has side effects that the caller is interested in. Allowing any of the other expressions at statement-level would not be logical. They either handle immutable objects, or they resemble pure operations. Because of this they have no added value if their outcome is discarded.

⟨*Program*⟩ ::= ⟨*Statement+*⟩
| ⟨*FunctionDef+*⟩

⟨*Statement*⟩ ::= ⟨*Call*⟩ ‘;’

⟨*Exp*⟩ ::= ⟨*Exp*⟩ ‘is not’ ⟨*Exp*⟩
| ‘not’ ⟨*Exp*⟩ {**avoid**}

Figure 5.9: Noteworthy Meta-Language syntax rules

The reason why the <Exp> rule is highlighted, is the **avoid** annotation on the negation. This avoid is a disambiguating measure. It is required as $e1$ *is not* $e2$ would be ambiguous under those rules without the annotation. Semantically this could check that $e1$ is not equal to $e2$, or that $e1$ is equal to the negation of $e2$. The first one is valid for any combination of data types. Contrary, $e2$ in the second scenario must be a subtype of ‘bool’. The ambiguity was resolved in favour of the most applicable one, so $e1$ *is not* $e2$ checks the inequality of 2 objects. Besides being usable for more data types, the variant with the negation can still be triggered by enclosing the expression in brackets, i.e. $e1$ *is* $(not\ e2)$. This would once again require $e2$ to be a subtype of ‘bool’.

5.4.2 Type Rules

The type rules for the meta language generally take 3 different scopes. These scopes are the local meta scope S_m, the global function scope S_f, and the global device scope S_d. S_m contains all variables that are available at the current level of the local scope. Their definition looks like $\text{VAR}(x) : T$, where x is the variable name and T is its type. Scope S_f contains all declared functions and their required parameter types and count. Declarations of functions within that scope are formed like $\text{FUNC}(fn_n) : (T_r, Ts_d)$. Here fn is the function name and n is the amount of paramters. T_r is the function’s return type, and Ts_d is a list of expected parameter types, in the order they are expected.

The devices scope contains two levels of definitions per device type. The first level contains the attributes that each device can have, and therefore can be queried about. The other definition defines which fields are available in each device. The former is important for the meta-level type rules, whereas the latter is unimportant till the mixin-language gets introduced. Both levels of the definition are modelled as scopes themselves. As such, a declaration for a device type looks like $\text{DEV}(dn) : (S_a, S_{df})$. Herein dn is the identifier of the device type, S_a is the scope containing all the attributes, and S_{df} contains all the device fields. Entries of S_a are $\text{ATTR}(an) : T$ with an being the attribute name, and T being its meta-level type. For S_{df} this is $\text{VAR}(fn) : T$ with fn

$$\frac{}{\varepsilon <: \_} \qquad \frac{}{\text{int} <: \text{real}} \qquad \frac{T_1 <: T_2}{\text{list}(T_1) <: \text{list}(T_2)} \qquad \frac{T_1 <: T_2}{\text{set}(T_1) <: \text{set}(T_2)}$$

$$\frac{T_1 <: T_3 \qquad T_2 <: T_4}{\text{dict}(T_1, T_2) <: \text{dict}(T_3, T_4)}$$

Figure 5.10: Subtyping (<:) relationships for the meta-language

as the fieldname, and T their PLC type [2].

The reason to separate these scopes is that they are all isolated. There isn't a single scenario in which a meta-language elements affects two of these scopes at once. When considering a context in which one scope could potentially be changed, the others are either read-only, or do not even exist yet. If desired, they could be modelled as a single scope, but this would require the entire scope to be traversed even though one is explicitly looking for a function which is declared outside the local context.

Many type rules reference the concept of subtyping (<:). It is therefore crucial to specify properties of a subtype relationship, and define what relationships exists in the context of the meta-language. The subtyping relationship is reflexive, meaning that $T <: T$ always holds, and it is transitive. Transitivity results in the following: If $T_1 <: T_2$ and $T_2 <: T_3$, then $T_1 <: T_3$. The relationships for the meta-language are shown in Figure 5.10.

While the ε in the first rule is not truly a type, it plays an important role in indicating that the type of a collection is unspecified. The user cannot reference the ε, and therefore is only meant as an internal representation.

Below are the typing details for the more interesting elements of the meta-language. Expressions have been omitted as it is assumed that their rules can easily be deduced from the typing table in Appendix D.

Statements

Statements are a core component of the meta-language. They make up entire programs, but also are a crucial element within control-statements. In most cases where a block supports statements, it actually concerns a list of statements. This list has to be iterated over. A map function could be used if the scopes did not change between each iterating element. It would be acceptable if the scopes were mutated, as long as they always pointed to the same object. However, since declaration-before-use is a must, a layering of scopes is required. This means that each statement should return its local scope, which may not be equal to the input scope. This returned scope is

[2] Notice that this concerns a PLC type. This will be addressed in section 5.5.2

$$(STMTS_{meta})\ \frac{S_m, S_f, S_d \vdash stmt \rightarrow S'_m \qquad S'_m, S_f, S_d \vdash ss :: \text{OK}}{S_m, S_f, S_d \vdash [stmt|ss] :: \text{OK}}$$

$$(STMTS_{meta})\ \frac{}{S_m, S_f, S_d \vdash [] :: \text{OK}}$$

Figure 5.11: Type rules to iterate over statements

the input for the next statement. The equivalent rules are shown in Figure 5.11. The statements themselves are in charge of which scope they return, this is not managed by the iterating rule.

Variable assignments and function calls are covered in their own sections, so the control statements are the only ones left to cover. These are displayed in Figure 5.12. As can be seen from these rules, none of the control statements make adaptations to their surrounding scope. The for-loop introduces a new sub-scope, but this only lives within the bounds of the control statement.

Variable Declarations and Assignments

Figure 5.13 shows the scope rules for the initial declaration of variables. It covers both the inferred and explicitly type variants. As visible, these rules are very similar. The difference is that in case of implicit typing it has to be ensured that the type of the assigned expression is not equal to an ε collection or to the type void. For explicitly typed declarations this is enforced through the subtyping relationship between T_1 and T.

Besides initial declarations, there are statements for the reassignment of variables. The corresponding rules are shown in Figure 5.14. Reassignments can change the value, not the type, therefore the type of the assigned value has to be a subtype of the already declared type. Unlike declarative assignment, it is also possible to do index-based reassignment. For lists these follow the same constraint as regular reassignment. For dictionaries it means that besides the values, the declared key and the provided key also must be compatible. Reassignment doesn't change the local scope, and as such the rules for these statements return the input scope.

Functions and Function calls

Unlike functions in SCL, the functions in the meta-language have a fixed parameter order. This greatly eases the declaration of the function, as it is only necessary to keep track of the expected types. The declared type of a function is a pair of its return type, and the types of its parameters. The scope rule for function declaration can be found in Figure 5.15. Defining a function changes the global function scope, but since the declaration order is irrelevant for functions, no new scope has to be introduced. Because of this it is also not necessary to return a scope.

$$(STMT_{meta})\ \frac{S_m, S_f, S_d \vdash e : \text{bool} \qquad S_m, S_f, S_d \vdash ss :: \text{OK}}{S_m, S_f, S_d \vdash \text{while}(e)\{ss\} \rightarrow S_m}$$

$$(STMT_{meta})\ \frac{\begin{array}{c} S_m, S_f, S_d \vdash e1 : \text{bool} \\ S_m, S_f, S_d \vdash e2 : \text{bool} \qquad S_m, S_f, S_d \vdash ss1 :: \text{OK} \\ S_m, S_f, S_d \vdash ss2 :: \text{OK} \qquad S_m, S_f, S_d \vdash ss3 :: \text{OK} \end{array}}{S_m, S_f, S_d \vdash \text{if}(e1)\{ss1\}\ \text{elif}(e2)\{ss2\}\ \text{else}\{ss3\} \rightarrow S_m}$$

$$(STMT_{meta})\ \frac{\begin{array}{c} \nabla S'_m \qquad S'_m \overrightarrow{P} S_m \qquad S_m, S_f, S_d \vdash e : \text{list}(T) \,||\, \text{set}(T) \\ T\ != \varepsilon \qquad S'_m \overset{\cdot}{\rightarrow} \text{VAR}(v) : T \qquad S'_m, S_f, S_d \vdash ss :: \text{OK} \end{array}}{S_m, S_f, S_d \vdash \text{for}(\text{var } v \text{ in } e)\{ss\} \rightarrow S_m}$$

$$(STMT_{meta})\ \frac{\begin{array}{c} \nabla S'_m \\ S'_m \overrightarrow{P} S_m \qquad S_m, S_f, S_d \vdash e : \text{list}(T_1) \,||\, \text{set}(T_1) \qquad T_1\ != \varepsilon \\ S_m, S_f, S_d \vdash v : T_2 \qquad T_1 <: T_2 \qquad S'_m, S_f, S_d \vdash ss :: \text{OK} \end{array}}{S_m, S_f, S_d \vdash \text{for}(\ v \text{ in } e)\{ss\} \rightarrow S_m}$$

$$(STMT_{meta})\ \frac{\text{breakAllowed}(ctx) :: \text{OK} \qquad \text{isLastStmt}(ctx) :: \text{OK}}{S_m, S_f, S_d \vdash \text{continue;} \rightarrow S_m}$$

$$(STMT_{meta})\ \frac{\text{breakAllowed}(ctx) :: \text{OK} \qquad \text{isLastStmt}(ctx) :: \text{OK}}{S_m, S_f, S_d \vdash \text{break;} \rightarrow S_m}$$

Figure 5.12: Type rule for control statements

$$(STMT_{meta})\ \frac{\begin{array}{c} \nabla S'_m \\ S'_m \overrightarrow{P} S_m \qquad S_m, S_f, S_d \vdash e : T_1 \qquad T_1 <: T \qquad S'_m \overset{\cdot}{\rightarrow} \text{VAR}(v) : T \end{array}}{S_m, S_f, S_d \vdash T\ v\ =\ e; \rightarrow S'_m}$$

$$(STMT_{meta})\ \frac{\begin{array}{c} \nabla S'_m \qquad S'_m \overrightarrow{P} S_m \qquad S_m, S_f, S_d \vdash e : T \\ T\ != \text{list}(\varepsilon) \,||\, \text{set}(\varepsilon) \,||\, \text{dict}(\varepsilon, \varepsilon) \,||\, \text{void} \qquad S'_m \overset{\cdot}{\rightarrow} \text{VAR}(v) : T \end{array}}{S_m, S_f, S_d \vdash \text{var } v\ =\ e; \rightarrow S'_m}$$

Figure 5.13: Type rules for variable declaration

$$(STMT_{meta})\ \frac{S_m, S_f, S_d \vdash e : T_1 \qquad S_m \Leftarrow \text{VAR}(v) : T_2 \qquad T_1 <: T_2}{S_m, S_f, S_d \vdash v = e; \rightarrow S_m}$$

$$(STMT_{meta})\ \frac{\begin{array}{c} S_m, S_f, S_d \vdash e : T_1 \qquad S_m, S_f, S_d \vdash i : T_2 \\ S_m \Leftarrow \text{VAR}(v) : \text{dict}(T_3, T_4) \qquad T_2 <: T_3 \qquad T_1 <: T_4 \end{array}}{S_m, S_f, S_d \vdash v[i] = e; \rightarrow S_m}$$

$$(STMT_{meta})\ \frac{\begin{array}{c} S_m, S_f, S_d \vdash e : T_1 \\ S_m, S_f, S_d \vdash i : \text{int} \qquad S_m \Leftarrow \text{VAR}(v) : \text{list}(T_2) \qquad T_1 <: T_2 \end{array}}{S_m, S_f, S_d \vdash v[i] = e; \rightarrow S_m}$$

Figure 5.14: Type rules for variable assignment

$$(FUNC_{meta})\ \frac{\begin{array}{c} \nabla S_m \qquad \text{len}(params) = n \qquad \text{extractTypes}(params) = Ts_d \\ \text{verifyFunction}(ctx, ss) \qquad S_m \overset{+}{\rightarrow} \text{VAR}(\text{"@rt"}) : T \\ S_m, S_f, S_d \vdash ss :: \text{OK} \qquad S_f \overset{+}{\rightarrow} \text{FUNC}(fn_n) : (T, Ts_d) \qquad Ts_c <: Ts_d \end{array}}{S_f, S_d \vdash \text{def}\ T\ fn(params)\{ss\} :: \text{OK}}$$

Figure 5.15: Type rule for function declaration

As functions can be overloaded based on parameter count it is important that the parameter count is part of the declaration and the reference. In the rules referred to above, the decision was made to include this in the function name, but there are other possibilities.

One interesting aspect of the function definition rule is the declaration of VAR("@rt"). This variable stores the return type. This allow the type system to resolve any return statement it encounters, without having the pass around the expected return type in the type rules. Figure 5.16 shows how return statements are verified.

The fixed parameter order also simplifies the validation of a function call, as displayed in Figure 5.17. Rather than having to validate the calling site against the callee, and vice versa, it now suffices to just address it from the calling site. When calling a function, the function scope is queried for the function that matches the function name and the parameter length. The types of the provided arguments are then determined (i.e. Ts_c) and compared to the expected types, Ts_d. Each provided argument should be a subtype ($<:$) of the expected parameter. The type of the function call is the return type of the function.

$$(STMT_{meta})\ \frac{\begin{array}{c}\text{returnAllowed}(ctx) :: \text{OK} \\ \text{isLastStmt}(ctx) :: \text{OK} \qquad S_m \Leftarrow \text{VAR}(\text{"@rt"}) \ : \ \text{void}\end{array}}{S_m, S_f, S_d \vdash \text{return;} \rightarrow S_m}$$

$$(STMT_{meta})\ \frac{\begin{array}{c}\text{returnAllowed}(ctx) :: \text{OK} \qquad \text{isLastStmt}(ctx) :: \text{OK} \\ S_m, S_f, S_d \vdash e : T_1 \qquad S_m \Leftarrow \text{VAR}(\text{"@rt"}) \ : \ T_2 \qquad T_1 <: T_2\end{array}}{S_m, S_f, S_d \vdash \text{return } e; \rightarrow S_m}$$

Figure 5.16: Type rules for return statements

$$(EXP_{meta})\ \frac{\begin{array}{c}\text{len}(params) = m \\ S_m, S_f, S_d \vdash params \ : \ Ts_c \qquad S_f \Leftarrow \text{FUNC}(fn_m) \ : \ (T, Ts_d) \qquad Ts_c <: Ts_d\end{array}}{S_m, S_f, S_d \vdash fn(params) \ : \ T}$$

Figure 5.17: Type rule for function calls

$$(EXP_{meta})\ \frac{S_m, S_f, S_d \vdash e \ : \ \text{string} \ || \ \text{list(\_)} \ || \ \text{set(\_)} \ || \ \text{dict(\_, \_)}}{S_m, S_f, S_d \vdash \text{len}(e); \ : \ \text{int}}$$

Figure 5.18: Type rule for the 'len(e)' standard function

The rules introduced above are applicable to user-defined function. Standard functions and their respective calls are explicitly modelled in the type system. A good example of this is the len(obj) function, displayed in Figure 5.18. The reason for this is that these do not necessarily adhere to the overloading constraint that applies to the user-defined functions. Because of this they cannot fall back upon the default ruleset. However, since these are functions intrinsic to the meta-language they can easily be incorporated in the type system.

There is a way for standard functions to fall back onto the rules for the user defined functions. This can be done by creating an internal type T for which $\{string, list, set, dict\} <: T$ holds. However, the introduction of such a type only fixes the issue for functions that take exactly that

$$(EXP_{meta})\ \frac{S_m, S_f, S_d \vdash e1 : \text{set}(T_1) \,||\, \text{list}(T_1) \qquad S_m, S_f, S_d \vdash e2 : T_2 \qquad T_2 <: T_1}{S_m, S_f, S_d \vdash e1.\text{add}(e2) : \text{bool}}$$

$$(EXP_{meta})\ \frac{\begin{array}{c} S_m, S_f, S_d \vdash e1 : \text{dict}(T_1, T_2) \\ S_m, S_f, S_d \vdash e2 : T_3 \qquad S_m, S_f, S_d \vdash e3 : T_4 \qquad T_3 <: T_1 \qquad T_4 <: T_2 \end{array}}{S_m, S_f, S_d \vdash e1.\text{add}(e2, e3) : \text{bool}}$$

Figure 5.19: Type rules for the '.add(...)' method

set of types. There might be other functions that require a different combination of types. Once each unique combination is given a supertype, the standard functions can be modelled as user-defined functions. To do so they simply have to be added to the function scope using the same declaration format as the other functions. Assuming the supertype of string, list, set, and dict is called 'sequence', the declaration for the len(obj) would be 'FUNC(len_1) : $(int, [sequence])$'.

While it is possible for functions to fall back on the default rules, methods cannot do that. Their explicit and varying nature makes it difficult to design a scope rule that is universally applicable. It is much easier to declare each method in its own rule. This is acceptable, especially since methods are fixed entities and users cannot add new ones. An example of a rule for a method call is shown in Figure 5.19.

Devices

Devices are a key factor in the parameterisation of the templates. They also function as a strong linking-pin between the object domain and the meta-domain. Since they are born out of a requirement related to the object-domain, it is not surprise that their meaning in the meta-language is limited. Most of the interaction with objects of type Device happens when splicing them into the object-level domain, which is of no concern to the meta-language. It will be covered in the definition of the mixin-language, 5.5.2. However, there are a meta-level function and a method that deals with them as well. These are 'findMatchingInstances' and 'getAttributeData'. Their typing rules are visible in Figure 5.20.

'findMatchingInstances' returns a list of Devices, but this is not all that interesting. However, it also requires the Device type when validating its second parameter, the condition. The condition is used to filter out results based on a criteria. This criteria always concerns one or more attributes declared within the device definition. The type rule has to check whether this attribute is valid for the queried device types. If the attribute exists, the type of the returned value is equal to the type of the attribute. If it does not exist, a type error will be returned. The rule for 'getAttributeData' is more or less identical to that of 'findMatchingInstances'. It also validates the parameters against the attributes in the Device's attributes scope. Where it differs is that the return type of the method depends on the attribute that is queried.

$$(EXP_{meta})\ \frac{S_d \Leftarrow \text{DEV}(DT_s) : (S_a, \_) \qquad \text{validateConditions}(cond, S_a)}{S_m, S_f, S_d \vdash \text{findMatchingInstances}(DT_s, cond) : \text{list}(\text{Device}(DT_s))}$$

$$(EXP_{meta})\ \frac{\begin{array}{c} S_m, S_f, S_d \vdash d : \text{Device}(DT_s) \\ S_d \Leftarrow \text{DEV}(DT_s) : (S_a, \_) \qquad S_a \Leftarrow \text{ATTR}(attr) : T \end{array}}{S_m, S_f, S_d \vdash d.\text{getAttributeData}(attr) : T}$$

Figure 5.20: Type rules for the '.add(...)' method

In the rules above, the assumption is made that the Device is a singular type with 2 scopes. In reality, the Device type can represent various types of devices (e.g. Device<DigitalAlarm, AnalogAlarm>), and is therefore composed of more than 2 scopes. However, these compounded types can be reduced to just 2 scopes, which make the introduced rules applicable to these scenarios as well. This reduction is done by calculating the intersection between the types of devices.

Assuming that there are n types of devices, and that the combinations are always sorted, there are $2^n - 1$ possible combinations. Computing the intersection for each of these combinations is an expensive operation, and generally unnecessary. In most projects the required compositions of device types will be limited. Given that most of the $2^n - 1$ combinations will remain unused, computing them preemptively is a waste of resources. Instead it is much more sensible to determine the intersection on a need-to-have basis. The calculation is done by recursively determining the intersection for both the attribute scopes and the field scopes of 2 devices types. The low-level mechanics of the scope-intersection function have been omitted from the type rules, but are based on iterating over all declarations in the scopes of one device type and checking their presence in the scopes of the other device type. Since this operation is idempotent, the results can be cached to further reduce computational effort.

One might wonder why it is that the intersection of the device types is used. Working with the intersection of the different device types ensures validity regardless of the content of the specification files, rather than providing guarantees based on assumptions/chance. If the type rules would depend on the union instead of the intersection, the excerpt in Figure 5.21 would be valid even though the field AutoOff does not exist in devices of type T2. There are scenarios in which this could lead to valid generated code, e.g. when no device instance of type T2 is present in the specification file. However, this success would depend on the content of the specification file. If someone was to change the specification file without re-evaluating the template, invalid code would be generated. Since the scenario for errors is more likely than those of false positives (i.e. reported errors when there are none), the decision was made to base validation on the intersection of the device types.

```
var devices = findMatchingInstances("T1,T2", \_, \_);
for (var d in devices) {
    $<$d$.AutoOff := TRUE;>$;
}
```

Figure 5.21: Field Validation for Compound Devices. 'AutoOff' is not a field in T2.

5.5 Mixin Language

The mixin-language is the glue between the object language and the meta-language. It is not really a language in the sense that it needs a 'host' language to latch onto.

If required, the mixin-language can introduce meta-programming elements specifically for the target object language. This makes it possible to add functionality which is not desirable to incorporate in the meta-language. If no additional functionality is required beyond what the meta-language offers, the additive element of the mixin-language is greatly reduced. Beyond adding laguage specific elements, the mixin-language governs where and how the meta-language can be used within the object-level code.

5.5.1 Syntax

Syntax-wise, the mixin language is insignificant in size. It imports the syntax from both the object and the meta-language. Importing the code stills result in two distinct languages. For that reason the mixin language must define a set of connection rules. These rules define where meta-level statements are allowed within the object language. Additionally, they specify how and where quoting and splicing can be used. Besides these limited additions, the Mixin language's syntax is unchanged compared to those it builds upon. However, these supplements make it significant in its functionality.

The BNF grammar specification in Figure 5.22 only shows the additions to the other syntaxes. For any predefined non-terminal on the left-hand side the rule must be read as another choice for that non-terminal.

As can be seen from the grammar, meta-level statements are allowed alongside SCL Units. This is achieved by making them an alternative for the existing <Unit> non-terminal. Furthermore, the statements can be used as object level statements and db-statements. This makes them usable within the blocks' logic sections. Within these logic sections it is also allowed to splice meta-objects of type <Statement> (not to be confused with the non-terminal <Statement>). This is why the "$<meta-expression>$;" rules are defined. Keep in mind that these rules don't enforce the type, this will be handled in the type system.

Identifiers can be formed through splicing. The fact that the <Identifiers> non-terminal is used as a placeholder does not mean that only identifiers can be spliced, any meta-object can. The reason for choosing identifiers as a placeholder is the fact that identifiers are allowed in many different locations where splicing would be beneficial to have. Using identifiers allows splicing in the left-hand side of an assignment, in the function name of a function call, in the header of a unit block, and in any expression context. This rules also allows SCL variables to

⟨*Unit*⟩ ::= ⟨*STMT-meta*⟩

⟨*Statement*⟩ ::= ⟨*STMT-meta*⟩
| '$'⟨*EXP-meta*⟩'$;'

⟨*Identifiers*⟩ ::= '$'⟨*EXP-meta*⟩'$'
| '$'⟨*EXP-meta*⟩'$' ('_'? (⟨*Letter*⟩ | ⟨*Digit*⟩))+

⟨*EXP-meta*⟩ ::= '<'⟨*Statement*⟩'>'
| '<'⟨*Expression*⟩'>'

⟨*STMT-meta*⟩ ::= '$'⟨*EXP-meta*⟩'$;' {**avoid**}

Figure 5.22: Mixin-Language syntax additions

be declared dynamically, but only in a pre-specified quantity. This is the direct effect of not allowing meta-level statements to be used within the Unit blocks' declaration sections.

The second rule for the identifiers might seem somewhat obscure. It allows for suffixed splicing. It is possible to achieve the same result through meta-level operations, but this syntax is more readable than string concatenation.

Since the decision was made to make the Device-related functions part of the reusable meta-language, the only addition that has to be made to the meta-level expressions is the addition of a quotation mechanism. Quotes can be applied to object-level statements and expressions.

The last rule specifies specifies that splicing can be done directly from the meta-domain. The 'avoid' is required. Without the annotation it would lead to ambiguities with the new rules in <Statement>. Adding it will solve the ambiguity in favor of the object language. Semantically the outcome would be identical if the meta-language was preferred, and therefore the annotation could just as well have been placed on the other violators instead.

One thing that still needs to be enforced is that splicing directly from the meta-context is only allowed within a control-statement within an SCL Unit's logic section. This could technically be captured as part of the syntax, but it would add an unnecessary amount of complexity to the grammar description. Instead, it will be incorporated in the type rules.

5.5.2 Type system

Contrary to the syntax, the type system requires some adaptations to be made to the rules from the object- and meta-language. Since the contexts are now intertwined, the domains need to be

$$(EXP_{meta})\ \frac{S_g, S_s, S_m, S_f \vdash e_{SCL} : \text{Var}(T)}{S_g, S_s, S_m, S_f \vdash <e_{SCL}> : <\text{Var}(T)>}$$

$$(EXP_{meta})\ \frac{S_g, S_s, S_m, S_f \vdash e : T}{S_g, S_s, S_m, S_f \vdash <e> : <T>}$$

$$(EXP_{meta})\ \frac{S_g, S_s, S_m, S_f \vdash stmt :: \text{OK}}{S_g, S_s, S_m, S_f \vdash <stmt> : <\text{Statement}>}$$

Figure 5.23: Type rules for quoting operations

aware of each other. In practice this means that the scopes for the meta-language need to be passed to all SCL rules, since either splicing is allowed, or meta-level statements are allowed, both of which require those scopes. Similarly, the current local SCL scope has to be available to the meta-domain as quoting is possible anywhere where meta-code is allowed. Additionally, object-level statements have to return the updated version of the meta-scope.

Besides these adaptations, the rules can be ported without changing a thing. The rest of the typing rules for mixin-language are additions to the pre-defined rules. Obviously, the new rules are related to the functionality that the mixin-language adds, which is quoting and splicing, but it also covers the type mappings from the meta-level objects to the object domain. The new rules are introduced in the following sections.

Quoting

The quotation rules from Figure 5.23 are rather simple. When quoting an expression, the current SCL context is used to determine the type of the expression. Since this expression itself may be formed through a series of meta-level operations, it is important to pass along the current meta-scopes as well. Once the type T has been determined, an implicit check is performed to see whether it concerns an SCL identifier. If it does, the quoted type is <Var(T)>. If it is not an identifier, the quoted type is simply <T>. For statements it must hold that the quoted SCL statement is a valid statement given the current scoping information. If it is, the quoted object is of type <Statement>.

Splicing

Splicing statements is only valid for objects of type <Statement>. Splicing expressions depends on a type mapping. Any type that is not part of this list is not eligible for splicing. The supported mappings for SCL-T are displayed in Figure 5.25.

$$(STMT_{SCL})\ \frac{S_g, S_s, S_m, S_f \vdash stmt : < \text{Statement} >}{S_g, S_s, S_m, S_f \vdash \$stmt\$; :: \text{OK}}$$

$$(EXP_{SCL})\ \frac{S_g, S_s, S_m, S_f \vdash e : T1 \qquad \text{type\_map}(T1) = T2}{S_g, S_s, S_m, S_f \vdash \$e\$: T2}$$

$$(EXP_{SCL})\ \frac{S_g, S_s, S_m, S_f \vdash e : < Var(\_) >}{S_g, S_s, S_m, S_f \vdash \$e\$_{suffix} : \text{Var(UNKNOWN)}}$$

$$(VAR_{SCL})\ \frac{S_g, S_s, S_m, S_f \vdash v : < \text{Var}(T) >}{S_g, S_s, S_m, S_f \vdash \$v\$: \text{Var}(T)}$$

$$(VAR_{SCL})\ \frac{S_g, S_s, S_m, S_f \vdash v : < \text{Var}(\_) >}{S_g, S_s, S_m, S_f \vdash \$v\$_{suffix} : \text{Var(UNKNOWN)}}$$

$$(STMT_{meta})\ \frac{\begin{array}{c} S_g, S_s, S_m, S_f \vdash stmt : < \text{Statement} > \\ \text{isUnitBlock}(ctx) :: \text{OK} \qquad \text{isControlStmt}(ctx) :: \text{OK} \end{array}}{S_g, S_s, S_m, S_f \vdash \$stmt\$; \rightarrow S_m}$$

Figure 5.24: Type rules for splicing operations

$$\begin{array}{c} \text{int} \Rightarrow \text{Implicit8BitConstant} \\ \text{real} \Rightarrow \text{REAL} \\ \text{bool} \Rightarrow \text{BOOLEAN} \\ \text{Device}(DTs) \Rightarrow \text{STRUCT}(s) \\ < T > \Rightarrow T \\ < \text{Var}(T) > \Rightarrow \text{Var}(T) \end{array}$$

Figure 5.25: Type mappings from meta-level objects to object-level types

$$(EXP_{SCL})\ \frac{S_g, S_s, S_m, S_f \vdash e : T1 \qquad \text{type\_map}(T1) = T2}{S_g, S_s, S_m, S_f \vdash \$e\$: T2}$$

Figure 5.26: Type rule for expression splicing

Var(T) is an intermediate type. When used within the object-level type system, it will be converted to type T. In the meta domain it is only eligible for quoting, leading to an object of type <Var(T)>.

Splice suffixing is only allowed for the objects of type <Var(T)>. This is due to the fact that <Var(T)> is the only data type that has the intrinsic properties to ensure that suffixing it will lead to another valid identifier. The type of the identifier cannot be determined, hence why the resulting type is Var(UNKNOWN), but it is guaranteed to be a valid identifier. Quoting a splice of an object of type <Var(T)> should still result in an object of type <Var(T)>. For this reason it is important to maintain this type, even though for the object-level domain, only type T is relevant. However, since type <Var(T)> is not part of the object language's type system, type validation would fail. To prevent this, an additional subtyping rule has to be defined for the object language: Var(T) $<:$ T. This will allow to type system to see those objects as their internally represented type.

Splicing generally occurs from within the object language. However, there is one form of splicing allowed from within the meta domain, which is statement splicing. It is only allowed when inside the Statement Section of a Unit Block, and only if the splice occurs from within a meta-level control statement. This restriction is depicted by the last rule in Figure 5.24. The first rule allows splicing inside the Statement Section of a Unit Block without being in a meta-level control statement. The difference is that this rule is part of the object-level type system, whereas the other is part of the meta-language.

Devices

The rule in Figure 5.26 has been extracted from the larger set of splicing rules. One thing that this rule hides is what happens when e is of type $Device(\_)$. While there is indeed a type-mapping, something else has preceded this event. This is related to the PLC type of the device fields, something which was grazed upon in section 5.4.2.

The IEC standard [52] specifies the technical implementation of data type, however, it leaves the implementing PLC languages free to support the type or not. For example, SCL does not explicitly support the LINT data type. The languages are also free to introduce additional types as they see fit. Because of these liberties it is not possible to assign a type to a device definition which is guaranteed to work in each target language. Instead, the device definitions use a PLC agnostic set of types. The mixin language introduces a mapping to their object language's counterpart.

$$true \Rightarrow BoolConstant(\text{“}TRUE\text{”})$$
$$false \Rightarrow BoolConstant(\text{“}FALSE\text{”})$$
$$int(num) \Rightarrow IntConstant(num)$$
$$real(num) \Rightarrow RealConstant(num)$$
$$Device(name) \Rightarrow RegularIdentifier(name)$$

Figure 5.27: Mapping from meta-variable to object-level AST

In case of SCL-T, this mapping turns devices into STRUCT containing fields of SCL supported types. From the perspective of SCL's type system, a Device(_) is nothing more than a STRUCT. By mapping devices this way, no additional rules have to be designed. Besides, encoding devices as STRUCTs is only marginally different from encoding them as a Data Block. After all, Data Block are also build on STRUCTs.

So the step performed by the type system is to turn the device's field scope into a STRUCT. However, the type system has been provided with a language specific set of rules to be able to do so. While it is trivial to do, it is crucial when building a similar language for a different PLC.

5.5.3 Object Representation

One thing which has not been addressed throughout this entire chapter is how to represent object-level code. The easiest way to deal with splices is to insert AST terms in the surrounding AST.

For code data types, this is pretty straight-forward. When quoting a section of object-level code, it is already parsed. If this AST is stored as the value of the code object, it can be spliced in without needing any conversion. For the none-code data types a special AST mapping is required. For the Device, int, real, and bool data types, this mapping to an object-level AST is easy. Their respective conversions are shown in Figure 5.27. These conversions are all eligible to be spliced in as an Expression. Since the type system enforces that these can only be spliced in that context, this suffices. Collections, strings and pairs cannot be spliced, and for that reason don't have an AST mapping.

After all splices have been replaced the resulting AST can be exploded (i.e. pretty printed) to its concrete equivalent. Exploding to concrete syntax is always possible as no unknown terms can be produced through splicing. While it is guaranteed that the resulting code can be parsed by SCL-T, there can be situations in which type violation occurs. This may be caused by the use of suffixed splices.

Chapter 6

Evaluation

SCL-T has been introduced; its features have been introduced. Additionally, it has been compared to UAB's existing language to show the differences. While those results were promising, they do not indicate how well SCL-T meets its goals. This is will be covered next. After that, some light will be shed on how the tool was received at CERN. Lastly, the process, the language's contributions, and potential research subjects related to PLC meta-programming will be discussed.

6.1 Evaluation of SCL-T

A list of goals for the new language facility was defined in section 2.3. These were formed by taking UAB's original goals, and adding to those the lessons that were taken from the initial language setup.

To determine whether the redesign of the languages is successful, it will now be held against these criteria:

- The tool should support statically loaded information, i.e. configuration and specification files, in all of its functionalities.
- The object and meta syntax should be distinguishable.
- The overall syntax should be readable.
- The user should feel like they are using an extended version of the object language.
- The boilerplate code should be kept to a minimum.
- Design one way to do something, and do it good.
- The meta-programs need to be validated, syntactically and semantically.
- The quality of the generated code should be safeguarded.
- Meta-expressions should be eligible for splicing without having to be lifted into variables.

- The language design needs to be portable to other PLC languages, preferably by reusing components.

By embedding the meta-language in the object language, rather than the other way around, the feeling of using an extended version of SCL is partially achieved. Since SCL is not suitable for homogeneous meta-programming, and extending it to appear homogeneous is treacherous (see section 3.2), a heterogeneous meta-language is the only option. Heterogeneity will always limit the degree in which it feels like an extended version of the object language, therefore this criteria is only partially achieved.

The reduction of boilerplate code also contributes to the feeling of using an extended language, rather than a meta-tool that can generate object-level code. The languages are largely distinguishable without explicit annotations. Whenever they are not, explicit annotations can be used to indicate whether something is a meta- or an object level element. These explicit annotations allow users to lift and splice meta-level code into object level code. The boilerplate code has been entirely reduced to these annotations, which has benefits to the overall readability as well.

Additionally, templates are now more resemblant of the final code and no longer suffer from mixed indentation rules. This greatly improves the overall readability of the templates. The comprehensibility is further improved by the removal of arbitrary code concatenation at object-level, and the toning down of the meta-language. The idea behind the latter was to 'Design one way to do something, and do it good'. The meta-facility should no longer allow the users to achieve similar functionality in multiple ways.

The templates are not only easier to read, but are also validated syntactically, and are semantically sound. If no error is detected when validating the template, it is guaranteed that the generation phase will not encounter a typing problem. Since other validation mechanism (e.g. data flow analysis) are not implemented, other type of execution errors might still occur at runtime.

As for the quality of the generated code, there is an almost conclusive guarantee that the code is syntactically correct. The only language features where this guarantee does not hold, are the usages of dynamically-formed names. Semantics-wise, the generated code is soundy. There are quite some elements that are either impossible to validate statically (since not all information is present), or whose gains do not outweigh their respective costs. As a result, even if all information was statically available, there would still be certain errors that remain undetected. This offers room for improvement, but overall the type system is able to detect common errors. If at any time the validation is deemed insufficient, only a few additional scoping rules would have to be added.

SCL-T is composed of three sub-languages. The idea behind this is that components are interchangeable, and can be reused to create new templating languages. The underlying architecture is shown in Figure 5.1. The same meta-language can be used to provide meta-programming for Siemens SCL, as well as for Schneider ST and other PLC languages. As detailed in section 5.1, this is not limited to reusing the same meta-language; more exotic combinations are possible.

All in all, it can be concluded that SCL-T meets the set requirements, though some only partially. As a result, SCL-T does not just meet the original set of goals for UAB, but also fills in the areas where the original language could be improved upon.

6.2 Prototype Reception at CERN

The decision to improve UAB with a new language was initially met with skepticism. Developers felt they had been promised a lot when UAB was initially released, and it did not always deliver. As a result, the promise of improvement initially fell on deaf ears. Nonetheless, the users were willing to share their experiences, which helped in gaining an understanding of how UAB could be improved upon.

As mentioned in section 2.2.2 one way to successfully introduce a new tool is to mimic an established one. The idea behind this is to reduce the hurdle of introducing a new language. Throughout the design process, I tried to stay as close to the original language(s) as possible. While there are still some obvious artifacts from the old approach, the languages have gone through quite a transformation. The overall syntax has changed, the meta-language capabilities have been altered, and the embedding of the language has been swapped around.

While still bearing some resemblance to the original approach, the concept of mimicry would most likely have not been applicable to SCL-T. For this reason, the new languages were introduced in smaller increments. As a result the developers were confronted with the changes in iterations, reducing the perceived impact. A welcome side-effect of the iterative approach was that the users' feedback was received during the build process.

The key in each prototype was to introduce a change that was undeniably a step forward from the previous, or to make the changes so minor that among all the things that were not altered, their effect to the developers workflow was negligible.

As indicated in chapter 2, UAB's original approach was sub-optimal, so many of the changes offered a significant improvement over the old way. This allowed the new tool to gain a significant foothold from the first prototype onwards, which only featured the improved syntax and syntactic validation.

Each prototype was created using Spoofax, a language workbench which makes it easy to create DSLs. An added advantage of Spoofax is that its DSLs run on top of the JVM, making them well suited for the intended purpose.

Advancing through the various prototypes, the down-scaled meta-language was introduced, as well as two static, single-staged type systems, one for the meta code, and one for the object code. While the addition of the type systems was welcomed dearly, the initial reactions on the limitations imposed upon the meta-language were mixed. Some key users saw the advantages of having restricted freedom (comprehensibility, consistency), where others were afraid that due to the limitations they would no longer be able to write the logic they needed. To mitigate these fears, some of the more complex templates were changed to the new format, showing that all logic required was still accessible in one way or the other. The key users were also informed to try and use the tool, and whenever they felt hindered in doing something, they were to report this. With the exception of a few issues that could be mitigated by providing an alternative approach,

there was no report of missing features, suggesting that the subset of functions and methods as implemented by SCL-T suffices.

Since my internship agreement with CERN ended, no new prototypes have been introduced, and no additional feedback was received on the latest prototype that exists at CERN. It is also unknown whether this prototype is still being improved upon, or even being used at all. As detailed in this study, the theoretical development of SCL-T has since continued. The syntax has been enhanced further, and the cross-stage type validation has been added. Additionally, the explicit annotations as mentioned in section 4.5 were added to give the users more control of code insertion and staging. A prototype for the final design was also created. This was made possible because the team behind Spoofax recently added Statix [55]: a DSL language which provides the functionality to declare type systems based on scope graphs.

6.3 Discussion

6.3.1 Weaknesses and Liabilities

The biggest weakness of this study is that some of the arguments used are based on the point of view from developers, both in the analysis of the existing language, and in the feedback on the intermediate prototype, while there is no quantitative analysis of their opinions to back-up the arguments. The only factual point that is documented with regards to the developers discontent towards the current approach, is CERN's request to redesign the language. The absence of the quantitative analysis can be seen as a hindrance in driving home the point regarding the improvement between the old language and the intermediate prototype. As a result it can also weaken the foundation on which the final design of the language is based. However, I believe that throughout this study I have presented enough objective evidence that suggests that the old approach was sub-optimal, and that the new meta-programming language not only meets the goals and requirements of UAB better, but is a better designed language overall.

Another identified external liability in this study is the absence of up-to-date documentation of the Siemens' SCL language. Apart from the additional work for the developers to fix the erroneous logic files, the impact of this is minimal, since the suggested approach is language independent. This means that any mistake made in the implemented object-language can be corrected without harming the overall language composition. At worst it requires some changes to be made to the object language and the mixin-language.

The last limitation in this study is that not all existing meta-programming approaches and type systems were evaluated. As discussed in chapter 3, there are simply too many different variants, and as such the analysis was done using a combination of taxonomies. By making choices based on subdivisions made in the taxonomies, I might have prematurely eliminated approaches which could have contained great ideas regarding meta-programming. Because of this I realise that SCL-T might not be the best, or most advanced, meta-programming language. However, as SCL-T meets all the set criteria, I do believe that for this purpose it is sufficient. Additionally, I believe the overall language design is built to last, as it was designed to be adaptable.

6.3.2 Alternative Approach

SCL-T is a technical solution to a problem which is not entirely related to technical implementation. One aspect which has been mentioned, but not thoroughly addressed is the role which the developers play in the problems encountered. For example, the varying levels of expertise in Python, and the (understandable) tendency to fix errors in the generated code without backporting the solution.

I believe that some problems would have been easier or more effective to fix by taking a procedural approach, rather than a technical one. The best example for this is related to code validation. I've reduced the chances of faulty templates being created in the first place. However, I'm in no way convinced that this problem is now resolved. In fact, as indicated in chapter 5, the type system is not fully sound, meaning that there is a chance that the developers still manage to produce erroneous generated code. This will now be more of an exception than a rule, as many of the common errors are detected before generation. However, I wonder whether the developers will not be even more enticed to fix the generated code rather than the templates. Addressing this issue from a management perspective might have removed the problem altogether (without any validation being added). My solution does not rule out a more procedural one from being put into place. In fact, it might have made it easier. Since the chances of error occurring have been reduced, putting in place a strict guideline might be met with less resistance, since the impact has been limited.

As for the decision to replace Python by a slightly adapted, greatly reduced version, I still stand by that approach, mainly because this was in line with the desired solution. However, I do think that an educational approach would have yielded welcome results as well. My point of view is that training the developers to ensure that a minimum level of Python knowledge existed, would have avoided some of the pre-existing difficulties. This would have made it possible to keep using the original Python meta-language. Of course, this would have had implications on the final product. It would have been more powerful (allowing more freedom), but at the cost of type validation. Additionally, it would also require a significant adaptation to the way the meta-language is embedded within the object language,

6.3.3 Future Work

One interesting aspect of PLCs is that they all need to adhere to a certain standard, yet none seem to be identical. In the current approach I have provided a meta-programming facility which is specialised for a single target language. However, given that there is a some sort of common ground, it feels like a lost opportunity not to chase a general meta-programming tool. Achieving a general templating tool that can generate specialised and equivalent code for various PLC manufacturers would be a powerful feat, as it would enable users to swap between manufacturers at the touch of a button. Determining whether this is possible will take a lot of knowledge on the various PLC language, especially to determine how they differ from each other. Additionally, it would require users of the various target PLCs to provide their input on what is needed.

Another area which I find worthy of investigation is performance optimalisation of the generated code. Since the DSL has intrinsic knowledge of the PLC code it might be possible to design a code generator that not only takes care of generating SCL code, but is also capable of

optimising/suggesting optimisations for the code. PLCs are ”real time” systems which need to process information within a restricted amount of time in order to prevent unintended effects, therefore it could be beneficial to be able to optimise the code so that the system can run faster. This is in line with the research being performed by Adiego et al [2], who try to achieve model verification for different PLC languages.

Lastly, fully-automated template generation would be very interesting. As mentioned in section 3.8, there currently is no need for multi-stage meta-programming. However, it was also noted that there might be a possibility for higher-level abstractions across the templates. It would be interesting to see whether these higher-level abstractions can be analysed and exploited to predict and form the set of templates needed for a specific setup. This would allow users to create an entire system solely by defining the device instances and their parameters.

Chapter 7

Conclusions

In this study SCL-T has been presented. It is a meta-programming language for Siemens' SCL's Programmable Logic Controllers (PLCs).

SCL-T builds upon UAB's approach to generate PLC code from templates. Even though UAB was successful, it was in a sub-optimal form. Scrutinising its design revealed some major weaknesses. Overall, the language design was counter-intuitive and it lacked any sensible validation. This severely impacted the tool's potential. Building on what was good, these weaknesses were used to formulate a set of goals that the successor to UAB's meta-programming language was meant to fulfill. Combining the lessons learned from UAB with a thorough analysis of existing literature on meta-programming, led to the design of SCL-T.

SCL-T's type system is based on scope graphs, and is statically and semantically sound with respect to the meta-stage. Additionally, it is able to guarantee syntactic correctness of the to-be-generated code. It also offers strong guarantees regarding the type-safety of that code, even though it is not fully sound. SCL-T allows user to write SCL code with a twist. Meta-statement are allowed at fixed locations in the syntax, and blocks of SCL code can be utilised through quoting and splicing.

One of the goals set out for SCL-T was for its design to be applicable to other PLC language. While SCL-T is tailored towards the SCL language, the underlying architecture allows a large degree of code re-use and was designed to support other PLC languages with their own fitted meta-programming language. The architecture makes it possible for components to be interchangeable, which allows for easier maintenance, and more freedom in language composition. The intention is that SCL-T provides a well-documented example of how to implement a similar meta-programming language for other PLC languages.

Bibliography

[1] Michael D Adams. Towards the essence of hygiene. In *ACM SIGPLAN Notices*, volume 50, pages 457–469. ACM, 2015.

[2] Borja Fernández Adiego, Dániel Darvas, Jean-Charles Tournier, Enrique Blanco Viñuela, and Víctor M González Suárez. Bringing automated model checking to plc program developmenta cern case study. *IFAC Proceedings Volumes*, 47(2):394–399, 2014.

[3] Hendrik van Antwerpen, Pierre Néron, Andrew Tolmach, Eelco Visser, and Guido Wachsmuth. A constraint language for static semantic analysis based on scope graphs. In *Proceedings of the 2016 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*, pages 49–60. ACM, 2016.

[4] R Barillere, R Nogueira Ferandes, E Blanco Vinuela, I Prieto Barreiro, and B Fernandez Adiego. Model oriented application generation for industrial control systems. In *Conf. Proc.*, volume 111010, page WEAAULT02, 2011.

[5] Bruno Barras, Samuel Boutin, Cristina Cornes, Judicael Courant, Jean-Christophe Filliatre, Eduardo Gimenez, Hugo Herbelin, Gerard Huet, Cesar Munoz, Chetan Murthy, et al. *The Coq proof assistant reference manual: Version 6.1*. PhD thesis, Inria, 1997.

[6] Jonathan Bartlett. The art of metaprogramming. *IBM developerWorks, October*, 2005.

[7] Don Batory. Product-line architectures. In *Smalltalk and Java Conference*, 1998.

[8] Martin Berger, Laurence Tratt, and Christian Urban. Modelling homogeneous generative meta-programming. *arXiv preprint arXiv:1602.06568*, 2016.

[9] Gilad Bracha and David Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *OOPSLA*, volume 93, pages 215–230. Citeseer, 1993.

[10] D. Bradutanu. *Resistance to Change - a New Perspective: A Textbook for Managers Who Plan to Implement a Change*. Lulu.com, 2015.

[11] Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(5):552–593, 2013.

[12] Edwin C Brady and Kevin Hammond. Scrapping your inefficient engine: using partial evaluation to improve domain-specific language implementation. In *ACM Sigplan Notices*, volume 45, pages 297–308. ACM, 2010.

[13] James R Cordy and Medha Shukla. Practical metaprogramming. In *Proceedings of the 1992 conference of the Centre for Advanced Studies on Collaborative research-Volume 1*, pages 215–224. IBM Press, 1992.

[14] Frank Cullen. Bridging legacy data with xml. *MYERSON, JM Enterprise Systems Integration. Second Edition. New York: Auerbach Publications*, pages 137–142, 2001.

[15] Robertas Damaševičius and Vytautas Štuikys. Taxonomy of the fundamental concepts of metaprogramming. *Information Technology and Control*, 37(2), 2008.

[16] Mathias D Dutour. Software factory techniques applied to process control at cern. Technical report, 2007.

[17] R Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in scheme. *Lisp and symbolic computation*, 5(4):295–326, 1993.

[18] Matthias Felleisen, Robert Bruce Findler, Matthew Flatt, Shriram Krishnamurthi, Eli Barzilay, Jay McCarthy, and Sam Tobin-Hochstadt. The racket manifesto. In *1st Summit on Advances in Programming Languages (SNAPL 2015)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[19] Ira R Forman, Nate Forman, and John Vlissides Ibm. Java reflection in action. 2004.

[20] Tokihiro Fukatsu, Masayuki Hirafuji, and Takuji Kiura. Massively distributed monitoring system application of field monitoring servers using xml and java technology. In *Proc. of the Third Asian Conference for Information Technology in Agriculture*, pages 414–417, 2002.

[21] Yoshihiko Futamura. Partial evaluation of ccomputation process-an approach to a compiler-compiler. *Systems, Compurters, Controls*, 25:45–50, 1971.

[22] Richard P Gabriel, Jon L White, and Daniel G Bobrow. Clos: Integrating object-oriented and functional programming. *Communications of the ACM*, 34(9):29–38, 1991.

[23] Philippe Gayet, Renaud Barillere, et al. Unicos a framework to build industry-like control systems, principles and methodology. *10th ICALEPCS*, 2005.

[24] Irwin D. Greenwald and Maureen Kane. The share 709 system: Programming and modification. *J. ACM*, 6(2):128–133, April 1959.

[25] Qi Hao, Weiming Shen, Zhan Zhang, Seong-Whan Park, and Jai-Kyung Lee. Agent-based collaborative product design engineering: An industrial case study. *Computers in industry*, 57(1):26–38, 2006.

[26] Lynn Hazan, Michaël Zugaro, and György Buzsáki. Klusters, neuroscope, ndmanager: a free software suite for neurophysiological data processing and visualization. *Journal of neuroscience methods*, 155(2):207–216, 2006.

[27] Andrew Hunt and David Thomas. The pragmatic programmer: from journeyman to master. 2000.

[28] Stephen C Johnson et al. *Yacc: Yet another compiler-compiler*, volume 32. Bell Laboratories Murray Hill, NJ, 1975.

[29] Christopher A Jones and Fred L Drake Jr. *Python & XML: XML Processing with Python*. " O'Reilly Media, Inc.", 2001.

[30] Mika Karaila and Tarja Systa. Applying template meta-programming techniques for a domain-specific visual language–an industrial experience report. In *29th International Conference on Software Engineering (ICSE'07)*, pages 571–580. IEEE, 2007.

[31] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. No Starch Press, 2018.

[32] Eugene Kohlbecker, Daniel P Friedman, Matthias Felleisen, and Bruce Duba. Hygienic macro expansion. In *Proceedings of the 1986 ACM conference on LISP and functional programming*, pages 151–161. ACM, 1986.

[33] Thomas Lawrence and Roy Suddaby. Institutions and institutional work. In *Handbook of Organization Studies*, volume 2, pages 215–254. Sage, 01 2006.

[34] Leon S Levy. A metaprogramming method and its economic justification. *IEEE Transactions on Software Engineering*, (2):272–277, 1986.

[35] Geoffrey Mainland. Explicitly heterogeneous metaprogramming with metahaskell. *SIGPLAN Not.*, 47(9):311–322, September 2012.

[36] Michael D. McCool, Zheng Qin, and Tiberiu S. Popa. Shader metaprogramming. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '02, pages 57–68, Aire-la-Ville, Switzerland, Switzerland, 2002. Eurographics Association.

[37] Chris McCord. *Metaprogramming Elixir: Write Less Code, Get More Done (and Have Fun!)*. Pragmatic Bookshelf, 2015.

[38] David Megginson. *Imperfect XML: Rants, Raves, Tips, and Tricks ... From an Insider*. Addison-Wesley Professional, 2004.

[39] Hervé Milcent, Enrique Blanco, Frédric Bernard, and Philippe Gayet. Unicos: An open framework. *Proceedings of ICALEPCS 2009*, 2009.

[40] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of standard ML: revised*. MIT press, 1997.

[41] Emir Pasalic. *The Role of Type Equality in Meta-programming*. PhD thesis, 2004. AAI3151199.

[42] Benjamin C Pierce and C Benjamin. *Types and programming languages*. MIT press, 2002.

[43] F Rideau and DV Ban. Metaprogramming and free availability of sources. In *Proc. of Autour du Libre Conference, Bretagne*, 1999.

[44] Jaideep Roy and Anupama Ramanujan. Xml schema language: taking xml to the next level. *IT professional*, 3(2):37–40, 2001.

[45] Tim Sheard. Accomplishments and research challenges in meta-programming. In Walid Taha, editor, *Semantics, Applications, and Implementation of Program Generation*, pages 2–44, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[46] Tim Sheard and Simon Peyton Jones. Template meta-programming for haskell. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, pages 1–16. ACM, 2002.

[47] Richard M Stallman and Zachary Weinberg. The c preprocessor. *Free Software Foundation*, 1987.

[48] Vytautas Štuikys and Robertas Damaševičius. *Meta-programming and model-driven meta-program development: principles, processes and techniques*, volume 5. Springer Science & Business Media, 2012.

[49] Don Syme. Leveraging. net meta-programming components from f#: integrated queries and interoperable heterogeneous execution. In *Proceedings of the 2006 workshop on ML*, pages 43–54. ACM, 2006.

[50] Walid Taha. *Multi-stage programming: Its theory and applications*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1999.

[51] Walid Taha and Tim Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical computer science*, 248(1-2):211–242, 2000.

[52] Michael Tiegelkamp and Karl-Heinz John. *IEC 61131-3: Programming industrial automation systems*. Springer, 1995.

[53] Laurence Tratt. Compile-time meta-programming in converge. 2004.

[54] Laurence Tratt. Domain specific language implementation via compile-time meta-programming. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(6):31, 2008.

[55] Hendrik van Antwerpen, Casper Bach Poulsen, Arjen Rouvoet, and Eelco Visser. Scopes as types. *Proc. ACM Program. Lang.*, 2(OOPSLA):114:1–114:30, October 2018.

[56] Guido Van Rossum et al. Python programming language. In *USENIX Annual Technical Conference*, volume 41, page 36, 2007.

[57] Todd L. Veldhuizen. Tradeoffs in metaprogramming. In *Proceedings of the 2006 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, PEPM '06, pages 150–159, New York, NY, USA, 2006. ACM.

[58] E Blanco Viñuela, M Koutli, T Petrou, and J Rochez. Opening the floor to plcs and ipcs: Codesys in unicos. *ICALEPCS13, San Francisco, USA*, 2013.

[59] John Von Neumann. First draft of a report on the edvac. *IEEE Annals of the History of Computing*, 15(4):27–75, 1993.

[60] Greg Wilson, Dhavide A Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven HD Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, et al. Best practices for scientific computing. *PLoS biology*, 12(1):e1001745, 2014.

Appendix A

Glossary

In this appendix we give an overview of frequently used terms and abbreviations.

SCL: Structured Control Language (Siemens' PLC language)

SCL-T: Templating language for Siemens SCL (SCL-Templating)

ST: Structured Text (Schneider's PLC language)

PLC: Programmable Logic Controller

UAB: UNICOS Application Builder

UNICOS: UNified Industrial Control System

Appendix B

Simple Template

```
LpVector = given;
name = given;
master = given;
Lp1,Lp2,Lp3=S7_DefaultAlarms_Tplt.LpSplit(LpVector);
Unit = Lp3;

plugin.writeSiemensLogic('''
FUNCTION $name$_DL : VOID
TITLE = '$name$_DL'
AUTHOR: 'UNICOS'
NAME: 'Logic_DL'
FAMILY: 'PCO'

BEGIN
$name$.AuOnR    :=  ($master$.RunOSt AND
                    (DB_GRAPH_$master$.RUN_UNIT$Unit$.X
                     OR
                     DB_GRAPH_$master$.START_UNIT$Unit$.X));
$name$.AuOffR   :=  NOT $name$.AuOnR;
$name$.AuCStopR :=  0;
$name$.AuOpMoR := 0.0;

(*IoSimu and IoError*****************)
DB_ERROR_SIMU.$name$_DL_E := 0; // To complete
DB_ERROR_SIMU.$name$_DL_S := 0; // To complete

END_FUNCTION
''')
```

Appendix C

Template in Old and New Syntax

```
supplied string name;
supplied string master;

FUNCTION $name$_CL : VOID
if master == name {
    $<$name$.IhAuMRW := 1;>$;
}
    $name$.CStopFin := 0;
if master == name or master.lower() == "no_master" {
    $<$name$.AuOpMoR := 0.0;>$;
}
    DB_ERROR_SIMU.$name$_CL_S := 0;
END_FUNCTION
```

Figure C.1: New Syntax

A basic example of the new syntax is given in Figure C.1. Figure C.2 displays the equivalent template in the old format. For comparison the same highlighting has been applied to the old template as can be achieved in the new, however for SCL-T this is syntax-based, whereas for UAB specific keywords are highlighted. This means that in UAB invalid code constructs would still be highlighted. As can be seen, there are still quite some resemblances between the old and the new format, but much has changed. The most obvious change is the reduction in the boilerplate code that is achieved by wrapping the meta-language in the object language, rather than vice versa. Many of the other changes, like the new type system, are internal and therefore do not directly influence the way the code looks.

Besides the new type system, SCL-T also offers full syntactic validation of the templates. Figure C.3 is a duplication of the problematic scenario sketched in Figure 2.8. Where the error would remain undetected in UAB, SCL-T is able to determine that the keyword is missing. The equivalent scenario in the new syntax is provided in Figure C.4, where the red line indicates the missing keyword “END_IF”.

```
def func(name, master, plugin):
   plugin.writeSiemensLogic('''
FUNCTION $name$_CL : VOID
    ''')
  if master == name:
    plugin.writeSiemensLogic('''
    $name$.IhAuMRW := 1;
    ''')
  plugin.writeSiemensLogic('''
    $name$.CStopFin := 0;
  ''')
  if master == name or master.lower() == "no_master":
    plugin.writeSiemensLogic('''
    $name$.AuOpMoR := 0.0;
    ''')
  plugin.writeSiemensLogic('''
    DB_ERROR_SIMU.$name$_CL_S := 0;
END_FUNCTION
  ''')
```

Figure C.2: Old Syntax

```
plugin.writeSiemensLogic('''
IF CONDITION_1 THEN
    # Common statements
    A := 1;
    B := 2;
''')
if condition_1:
    plugin.writeSiemensLogic('''
    IF CONDITION_2 THEN
        $name$.AuRStart := TRUE;
    END_IF;
    ''')
else:
    plugin.writeSiemensLogic('''
    IF CONDITION_3 THEN
        $name$.AuRStop := TRUE;
    END_IF;
    ''')
```

Figure C.3: Missing keyword

```
IF CONDITION_1 THEN
    # Common statements
    A := 1;
    B := 2;
if condition_1 {
    $ < IF CONDITION_2 THEN
        $name$.AuRStart := TRUE;
    END_IF; > $;
}
else {
    $ < IF CONDITION_3 THEN
        $name$.AuRStop := TRUE;
    END_IF; > $;
}
```

Figure C.4: Missing Keyword

Appendix D

Meta-Language Functions and Methods

len(s)	int	s:set(_)
x in s	bool	x:T; s:set(T)
x not in s	bool	x:T; s:set(T)
s.issubset(t)	bool	s:set(T); t:set(T)
s.issuperset(t)	bool	s:set(T); t:set(T)
s.union(t)	set(T)	s:set(T); t:set(T)
s.intersection(t)	set(T)	s:set(T); t:set(T)
s.difference(t)	set(T)	s:set(T); t:set(T)
s.symmetric_difference(t)	set(T)	s:set(T); t:set(T)
s.copy()	set(T)	s:set(T)
s.deepcopy()	set(T)	s:set(T)
s.add(x)	bool	s:set(T); x:T
s.remove(x)	bool	s:set(T); x:T
set(l)	set(T)	l:list(T)

Table D.1: Set functions and methods

len(l)	int	l:list(_)
x in l	bool	x:T; l:list(T)
x not in l	bool	x:T; l:list(T)
l.copy()	list(T)	l:list(T)
l.deepcopy()	list(T)	l:list(T)
l.add(x)	bool	l:list(T); x:T
l.remove(x)	bool	l:list(T); x:T
l[i]	T	l:list(T); i:int

Table D.2: List functions and methods

len(m)	int	m:dict(_,_)
x in m	bool	x:K; m:dict(K,_)
x not in m	bool	x:K; m:dict(K,_)
m.copy()	dict(K,V)	m:dict(K,V)
m.deepcopy()	dict(K,V)	m:dict(K,V)
m.add(k,v)	bool	m:dict(K,V); k:K; v:V
m.remove(k)	bool	m:dict(K,V); k:K
m[k]	V	m:dict(K,V); k:K
m.keys()	list(K)	m:dict(K,V)
m.values()	list(V)	m:dict(K,V)
m.items()	list(pair(K,V))	m:dict(K,V)

Table D.3: Dictionary functions and methods

p.left()	L	p:pair(L,R)
p.right()	R	p:pair(L,R)

Table D.4: Pair methods

len(s)	int	s:string
s.upper()	string	s:string
s.lower()	string	s:string
s1.join(s2)	string	s1:string; s2:string
s.strip()	string	s:string
s1.replace(s2,s3)	string	s1:string; s2:string; s3:string
s1.split(s2)	string	s1:string; s2:string

Table D.5: String functions and methods

abs(n)	int	n:int
abs(n)	real	n:real
any(l)	bool	l:list(bool)
all(l)	bool	l:list(bool)
range(i1,i2)	list(int)	i1:int; i2:int

Table D.6: General collection functions

not a	bool	a:bool
a and b	bool	a:bool; b:bool
a or b	bool	a:bool; b:bool

Table D.7: Logical operators

a is b	bool	a:_; b:_
a is not b	bool	a:_; b:_
a == b	bool	a:_; b:_
a != b	bool	a:_; b:_
a {<, <=, >, >=} b	bool	a:real; b:real

Table D.8: Equality operators

a + b	string	a:_; b:string
a + b	string	a:string; b:_
a {+, -, *, /, **} b	int	a:int; b:int
a {+, -, *, /, **} b	real	a:real; b:real

Table D.9: Arithmetic operators

Appendix E

Meta-Language Syntax

⟨*Program*⟩ ::= ⟨*Statement+*⟩
| ⟨*FunctionDef+*⟩

Figure E.1: Meta-Language: Start Symbols

⟨*Type*⟩ ::= ‘`bool`’
| ‘`int`’
| ‘`real`’
| ‘`string`’
| ‘`Device(`’ ⟨{*ID*”,”}+⟩ ‘`)`’
| ‘`Device(*)`’
| ‘`list(`’ ⟨*Type*⟩ ‘`)`’
| ‘`set(`’ ⟨*Type*⟩ ‘`)`’
| ‘`dict(`’ ⟨*Type*⟩ ‘`,`’ ⟨*Type*⟩ ‘`)`’
| ‘`pair(`’ ⟨*Type*⟩ ‘`,`’ ⟨*Type*⟩ ‘`)`’

Figure E.2: Meta-Language: Types

⟨*Statement*⟩ ::= ⟨*ControlStatement*⟩
| ⟨*Assignment*⟩
| ⟨*Call*⟩ ‘`;`’

⟨*ControlStatement*⟩ ::= ‘`if`’ ‘`(`’ ⟨*Exp*⟩ ‘`)`’ ‘`{`’ ⟨*Statement**⟩ ‘`}`’ ⟨*ElseIf**⟩ (‘`else`’ ‘`{`’ ⟨*Statement**⟩ ‘`}`’)?
| ‘`while`’ ‘`(`’ ⟨*Exp*⟩ ‘`)`’ ‘`{`’ ⟨*Statement**⟩ ‘`}`’
| ‘`for`’ ‘`(`’ ⟨*AssignmentVar*⟩ ‘`in`’ ⟨*Exp*⟩ ‘`)`’ ‘`{`’ ⟨*Statement**⟩ ‘`}`’
| ‘`break`’ ‘`;`’
| ‘`continue`’ ‘`;`’
| ‘`return`’ ⟨*Exp?*⟩ ‘`;`’

⟨*ElseIf*⟩ ::= ‘`elif`’ ⟨*Exp*⟩ ‘`{`’ ⟨*Statement**⟩ ‘`}`’

⟨*Assignment*⟩ ::= ⟨*AssignmentVar*⟩ ‘`=`’ ⟨*Exp*⟩ ‘`;`’
| ⟨*ID*⟩‘`[`’⟨*Exp*⟩‘`]`’ ‘`=`’ ⟨*Exp*⟩ ‘`;`’

⟨*AssignmentVar*⟩ ::= ‘`var`’ ⟨*ID*⟩
| ⟨*Type*⟩ ⟨*ID*⟩
| ⟨*ID*⟩

Figure E.3: Meta-Language: Statements

⟨*Exp*⟩ ::= ⟨*ID*⟩
| ⟨*REAL*⟩
| ⟨*INT*⟩
| ⟨*STRING*⟩
| ‘`true`’
| ‘`false`’
| ‘`[`’ ⟨{*Exp*”,”}*⟩ ‘`]`’
| ‘`(`’ ⟨*Exp*⟩ ‘`,`’ ⟨*Exp*⟩ ‘`)`’
| ‘`{`’ ⟨{*DictEntry*”,”}*⟩ ‘`}`’
| ⟨*Exp*⟩ ‘`[`’ ⟨*Exp?*⟩ ‘`:`’ ⟨*Exp?*⟩ ‘`]`’
| ⟨*Exp*⟩ ‘`[`’ ⟨*Exp*⟩ ‘`]`’
| ⟨*Call*⟩
| ⟨*Exp*⟩ {‘`+`’|‘`-`’|‘`*`’|‘`/`’|‘`%`’|‘`**`’} ⟨*Exp*⟩
| ⟨*Exp*⟩ {‘`==`’|‘`!=`’|‘`<`’|‘`<=`’|‘`>`’|‘`>=`’|‘`is`’|‘`is not`’} ⟨*Exp*⟩
| ⟨*Exp*⟩ {‘`and`’|‘`or`’} ⟨*Exp*⟩
| not ⟨*Exp*⟩ {**avoid**}
| {‘`+`’|‘`-`’} ⟨*Exp*⟩
| ‘`(`’ ⟨*Exp*⟩ ‘`)`’

⟨*DictEntry*⟩ ::= ⟨*Exp*⟩ ‘`:`’ ⟨*Exp*⟩

Figure E.4: Meta-Language: Expressions

⟨*FunctionDef*⟩ ::= ‘`def`’ ⟨*Type*⟩ ⟨*ID*⟩ ‘`(`’ ⟨{*ParameterDecl*”,”}*⟩ ‘`)`’ ‘`{`’ ⟨*Statement**⟩ ‘`}`’

⟨*ParameterDecl*⟩ ::= ⟨*Type*⟩ ⟨*ID*⟩

⟨*Call*⟩ ::= ⟨*ID*⟩ ‘`(`’ ⟨*Exp**⟩ ‘`)`’
| ⟨*Exp*⟩ ‘`.`’ ⟨*ID*⟩ ‘`(`’ ⟨*Exp**⟩ ‘`)`’

Figure E.5: Meta-Language: Functions