WILHELM BÜCHNER HOCHSCHULE
Mobile University of Technology

Fachbereich Informatik
Hilpertstraße 31
D-64295 Darmstadt

Development of a domain-specific language for cronjobs with the language workbench JetBrains MPS

Betreuer:	Sebastian Lothary, M.Sc
Autor:	Alexander Pann
Matrikelnummer:	901084
Anschrift:	Hugo-Wolf-Gasse 8A Top 12
	8010 Graz, Österreich
Abgabetermin:	31. Mai 2020

Zusammenfassung

Die Automatisierung in Unternehmen ist ein Thema, das in Zukunft immer wichtiger wird. Unix und Unix-ähnliche Betriebssysteme verfügen zu diesem Zweck über einen sogenannten Cron-Dämon, der es erlaubt, wiederkehrende Aufgaben zu festen Zeiten auszuführen. Cron kann nur zeitbasierte Trigger verwenden und lässt einige Funktionen vermissen, die zur Verbesserung der Software-Automatisierung notwendig wären.
Diese Arbeit beschreibt eine neue domänenspezifische Sprache für Cronjobs, die zeit- und ereignisbasiert ist und die Language Workbench JetBrains MPS als Entwicklungswerkzeug für die Erstellung der Sprache und als integrierte Entwicklungsumgebung für in dieser Sprache erstellte Skripte verwendet.
Der zweite Teil dieser Arbeit besteht aus einer Java-Anwendung, die von MPS generierten Cronjobs ausführen und Informationen über diese Jobs in einer Datenbank speichern kann.

Abstract

Automation in companies is an increasingly important topic in the future. Unix and Unix-like operating systems have a so-called cron daemon for this purpose, which allows recurring tasks to be executed at fixed times. Cron can only use time-based triggers and misses some features that would be necessary to improve software automation.
This work describes a new domain-specific language for cronjobs that is time-based and event-based and uses the language workbench JetBrains MPS as a development tool for creating the language and as an integrated development environment for scripts created in this language.
The second part of this work consists of a Java application that can execute the cronjobs that were generated by MPS and save information about these jobs in a database.

Acknowledgements

I would like to thank my supervisor Sebastian Lothary, M.Sc, for taking this time to review my work and providing valuable feedback for the structure and content of my work.
Many thanks to the development team at JetBrains who worked on the language workbench MPS for 15 years and are still improving the product every year. They showed that projectional editing can be used in practice and that language-oriented programming is not just a language paradigm that can be used by researchers but also by companies in a model-driven environment.
I want to thank Professor Fabien Campagne, Ph.D. Department of Physiology and Biophysics, Professor of His Royal Highness Prince Alwaleed Bin Talal Bin Abdulaziz Alsaud Institute for Computational Biomedicine, Associate Director, Computational Biomedical Informatics Core, Clinical and Translational Science Center of the Weill Cornell Medical College of the Cornell University New York for improving my knowledge of the MPS system and mentoring me during my three months summer internship in New York City.
Last but not least I want to thank my parents Sabine and Peter Pann, M.Sc, for many discussions during my work.

Contents

List of Figures iv

List of Tables vi

Acronyms vii

1. Introduction 1
   - 1.1. Preface 1
   - 1.2. Motivation 1
   - 1.3. Goal 1
   - 1.4. Differentiation 2
   - 1.5. Scientific Approach 3
   - 1.6. Summary 3

2. State of the Art 4
   - 2.1. Cronjobs 4
     - 2.1.1. Crontab 4
     - 2.1.2. Format 5
     - 2.1.3. systemd 5
     - 2.1.4. Cron vs systemd timers 6
     - 2.1.5. Similar Works 6
   - 2.2. Language Oriented Programming 7
     - 2.2.1. Advantages 7
     - 2.2.2. Disadvantages 8
     - 2.2.3. Comparision with other Methods 8

2.2.4. Workflow Comparision with General-Purpose Languages . 9
2.3. JetBrains MPS 11
2.3.1. Terminology 11
2.3.2. Choice of Language Workbench 12
2.3.3. Domain-specific Modeling 14
2.3.4. Projectional Editing 15
2.3.5. Notations 17
2.3.6. Case Studies 19
2.4. Good Language Design 25
2.5. Summary 29

3. Concept **30**
3.1. DSL in MPS 31
3.2. Usage of Features in MPS 31
3.3. Utility Model in MPS 32
3.4. Java Server 33
3.5. Summary 33

4. Prototyping **34**
4.1. Swino 35
4.1.1. Language (swino.lang) 35
4.1.2. Editor 37
4.1.3. Type System and Constraints 38
4.1.4. Runtime model (swino.lang.runtime) 39
4.2. SwinoServer 40
4.2.1. Quartz Scheduler 40
4.2.2. SQLite Database 41
4.2.3. Database Schemes 41
4.2.4. From Clicking "schedule" to the Execution of a Job 43
4.2.5. Code Generation 43
4.3. Dependencies 45
4.4. Testing 45
4.5. Summary 48

5. Evaluation **49**

5.1. Cognitive Walkthrough 49

5.2. Result 51

5.3. Summary 52

6. Conclusion and Future Work **53**

6.1. Conclusion 53

6.2. Future Work 54

6.3. Summary 54

7. Bibliography **55**

A. Appendix **58**

A.1. Creation of the Database Schemes 58

A.2. Scheduler Server Configuration 62

A.3. Scheduler Client Configuration 63

List of Figures

2.1. Crontab Syntax 5

2.2. Aspects of a language 12

2.3. Multiple Editors for the same Concept 18

2.4. Mbeddr 20

2.5. MetaR 22

2.6. KernelF 23

2.7. Example for a good choice of syntax 28

3.1. Data Flow 30

4.1. Project Component Diagram 34

4.2. Concept: Script 35

4.3. Concept: Job 36

LIST OF FIGURES

4.4. Concept: Trigger 37

4.5. Editor for scripts 38

4.6. Uniqueness check 39

4.7. server.policy File 41

4.8. Model Transformation Steps 44

4.9. Root mapping rule for the concept: Job 44

4.10. Test: Unique Names 46

4.11. Test: Name Trigger After Job 47

List of Tables

3.1. Used MPS features 32

4.1. Database tables 42

4.2. Database table *CRON_EXECUTION_INFO* 42

4.3. Dependencies 45

5.1. Evaluation Results MPS Experts 51

5.2. Evaluation Results non MPS Experts 51

Acronyms

AST	abstract syntax tree
DSL	domain-specific language
DSM	domain-specific modeling
IDE	integrated development environment
LWB	language workbench
MPS	JetBrains MPS
RMI	Remote Method Invocation

1. Introduction

1.1. Preface

Automation in companies is an increasingly important topic in the future. Unix and Unix-like operating systems offer the so-called cron daemon for this purpose, which allows recurring tasks to be executed at fixed times. The required instructions are stored in a special table, the so-called crontab. Time-based rules can define at which minute, hour, day of the month, month and weekday a task should be processed.

1.2. Motivation

Cronjobs date back to the 1990s. They were primarily a handy automation tool for system administrators to automatically run scripts add specific times. They are reliable if set up correctly but they don't support any additional features, such as re-running tasks or canceling tasks. However these days, these supporting features are often necessary for automating and therefore better systems need to be developed.

1.3. Goal

The goal is to develop a DSL for the already mentioned cronjobs, which is not only time-based but also event-based. This language should allow creating cronjobs with an easy to understand language, where the execution can be influenced by different events or properties of the cronjobs (e.g. processor load, return value of last execution, etc.). The *Language Workbench MPS* should be used for creating

the DSL and used as an editor for script creation. The language won't be text-based but uses a projectional editor. Projectional editing is used by a few businesses and for research but was never used in combination with cronjobs and automation.
It should be possible to write cronjobs in text form and to visualize them with graphical tools. Important meta-information about the system and the cronjobs can be automatically stored in a database and thus be used by other programs, for example for analysis or creation of reports. To execute the cronjobs, Java source code must be generated by the DSL, which can then be executed by a job scheduler. This work not only tries to present an alternative to cronjobs but also introduces a new language for describing cronjobs.
The project uses MPS as an IDE and therefore users need to have a basic understanding of MPS and projectional editing which can be learned in a few hours. A user should have proficient computer skills to master these new topics.
The Swino language should be used as a replacement of cronjobs. Cronjobs are written by developers and most of the time by system administrators. They are the main target audience but also other types of users can use this language if they want to automate tasks. An end-user should also be able to write scripts if they get used to the different types of editing.
This language only builds the foundation for automating tasks. When it is used for a project, it has to be refined and extending to provide a better mapping from the problem domain to the solution domain. A domain expert is needed to analyze the requirements of the domain.
The mapping should be done by a language developer. He is responsible for modifying and extending the existing languages and developing extensions for the project's domain.
Good language design is hard and should, as a result, be done by a developer that has experience on this topic. Tests are also normally developed by the language designer.

1.4. Differentiation

This work tries to not use a text-based approach for describing cronjobs. An alternative scheduler library is used but code doesn't have to be written in a general-purpose language to use the library because the code is generated by the Language Workbench. The DSL is not supposed to support many features. It is merely a prototype language that can be extended to support other domains.

1.5. Scientific Approach

The multimethodological approach by Nunamaker, Chen, and Purdin [19] will be used:

Theory building In this stage, new ideas, concepts, and methods will be developed. Alternatives to cronjobs will be reviewed. Language oriented programming will be described briefly.

Experimentation Code experiments will be done. It will be checked if language-oriented programming is an appropriate software paradigm that can solve the mentioned problem. A language workbench will aid in the development of such prototypes.

Observations The problem domain will be examined and research methodologies such as case studies will be presented. The reader should get a better feeling for the domain.

System development The following stages of system development will be run through concept design, constructing the architecture, prototyping and product development.

Afterward, the language will be evaluated by users and a conclusion will be reached.

1.6. Summary

This chapter gave a short introduction to the topic of this work. The motivation and goal were presented and the work was differentiated from other topics. Lastly, the multimethodological approach that is used in this work was shortly explained.

2. State of the Art

This chapter describes cronjobs which are known and used by professionials for more than two decades. Language Oriented Programming, which was developed at the same time is also introduced and it will be shown how it can be implemented with MPS.

2.1. Cronjobs

The cron daemon (*crond*) [25] which is included in most Linux distributions allows scheduling of recurring jobs at specific intervals. It was first introduced to the public in an edition of the Linux Journal [14] in 1999. In multi-user mode it searches the following directories for jobs to execute: /var/spool/cron/crontabs, /etc/cron.d and /etc/crontab. It performs this search every minute and executes the jobs at the declared times. Optionally it can send the output to the owning user.

2.1.1. Crontab

Cronjobs are stored in cron tables/crontab files which can be edited with the command `crontab -e`. crontab [24] looks in the *VISUAL* and *EDITOR* environment variables for a suitable editor and use it. This command has to be used because it checks the files for syntax errors and refuses to save them if they contain errors. The crontab files are owned by the root user of the system.
The edited file can be viewed with the command `crontab -l`. To send mail to its owner, the environment variable *MAILTO* has to be set.

2.1.2. Format

A single line in a crontab file consists of the schedule and the command that should be executed. Figure 2.1 shows an overview.

```
* * * * * command to execute
┬ ┬ ┬ ┬ ┬
│ │ │ │ │
│ │ │ │ └──── weekday (0-7, Sunday is 0 or 7)
│ │ │ └────── month (1-12)
│ │ └──────── day (1-31)
│ └────────── hour (0-23)
└──────────── minute (0-59)
```

Figure 2.1.: Crontab Syntax

Further information can be found in the man pages [24].

2.1.3. systemd

Some Linux systems use the daemon *systemd* instead of crond. To add a new job two files must be created: a systemd unit file with a suffix of .timer and a matching .service file. There are two different timers:

Realtime timers They activate at a specific schedule the same way cronjobs do.

Monotonic timers According to its creator Poettering [20] they are activated after a time span relative to a varying starting point.

The .timer file activates and controls the service file.
The following commands can be used to manage these files:
Start a unit immediately: `systemctl start unit`
Stop a unit immediately: `systemctl stop unit`
Restart a unit: `systemctl restart unit`

2.1.4. Cron vs systemd timers

According to an answer on Unix and Linux Stack Exchange [11] systemd timers seem to be the better choice if systemd is available on the system: All systemd timer events are logged in the systemd journal, cronjobs are not logged. systemd timers are systemd services with the same capabilities (for e.g resource management and membership controls).
They can depend on other services and are not only time-dependent but also event-dependent. They can also be enabled and disabled easily as mentioned earlier. systemd timer use seconds as the minimal scheduling interval.
systemd timers can be run in a specific environment and jobs can be started independently of their timers which simplifies debugging There are also two disadvantages: Two files need to be created per job and there is no built-in support for sending emails.

2.1.5. Similar Works

There are already solutions for various programming languages such as the *Quartz Scheduler* and the software library *cron4j* for the Java programming language. However, these solutions require the design and execution of software programs that can only be written by software developers.
The goal of this DSL is to enable domain experts and also end-users to write cronjobs without much previous knowledge and to be able to analyze data about these tasks easily.
In contrast to the existing solutions, no general-purpose language like Java or C++ should be used, but a language that was designed directly for this specific domain (the creation of cronjobs).
Moreover, information and actions for individual tasks such as past execution results and buttons for testing cronjobs can be directly included in the declaration of the cronjob. There are also timers for the daemon *systemd* that can perform similar tasks as cronjobs.

2.2. Language Oriented Programming

Large software systems cause problems when traditional development methods are used. According to Brooks [3] some problems are:

Complexity can't be abstracted away. It is difficult to get an overview of the system and it can't be visualized easily.

Conformity Many systems need to conform to human institutions and systems.

Change Systems will be modified to a level, that it isn't capable of and be ported to other machines and environments.

Language-oriented programming is a software paradigm where DSLs are created to solve a problem. These DSLs are then used to solve problems in a way that is closer to the solution domain than an approach with a general-purpose language.

2.2.1. Advantages

Ward [31] mentions a few advantages:

Separation of Concerns Design issues and implementation issues can be separated. The language design can be kept simple while reducing the overall complexity of the system.

High Development Productivity The development process gets sped up because the number of lines of code is smaller. A language on a very high abstraction level can achieve a lot of work and productivity can be increased by restricting the language to a specialized domain.

Highly Maintainable Design More lines of code mean more maintenance effort. Language oriented programming leads to a small total size of code and is therefore highly maintainable. Design decisions are not spread through the codebase as it is the case with traditional programming methods but is

abstract away instead. A single decision will only affect a single part of the system.

Highly Portable Design Code portability gets simpler with this approach: Only middle-level languages in a hierarchy of languages need to be ported to the new system of the environment.

Opportunities for Reuse These middle-level languages can be reused for similar projects. The domain knowledge is already captured in these languages and existing programming languages don't need to be refined. This approach also works better than a collection of functions, abstract data types or objects.

User Enhancable System A top-level language in a hierarchy of languages is very close to the solution domain and very detailed specified. Most of the time it is interpreted and not compiled and can be used to create macros for lower-level languages. The user can enhance the system but these enhancements must not make the system worse. One solution to make these high-level languages safe is that code in this language always creates a meaningful program in the problem domain.

2.2.2. Disadvantages

There are also problems with language-oriented programming. One of the most important problems is that good language design is a difficult task. High-level languages can also hide efficiency issues that can be seen at a lower level. Introducing this programming paradigm can also be difficult because developers have to do high-level abstract thinking about the problem domain to produce good languages at the middle level.

2.2.3. Comparision with other Methods

Language oriented programming can be compared to more traditional approaches ([23] [16]):

Top-Down Development The development starts at a high level of abstraction

and steps down in a top-down approach. The most important issue with this approach is that it is unclear what the top-level structure is supposed to look like. A serious mistake at this level of abstraction can lead to a redevelopment of the whole system if the problem is found at a later stage of development.

Bottom-Up Development This approach starts from the bottom with utility routines and gets more domain-specific with each step. Especially on the middle level, it can be difficult to decide on the next level of abstraction. Sometimes a higher level implements the wrong functions or the functions are not needed.

Outside-In Development This approach combines the top-down and the bottom-up approach. Both developments are done in parallel and meet at the middle level. Problems from both approaches also appear in outside-in development. Additionally, it can happen that both design implementations don't meet or overlap and implement similar functionality.

Middle-Out Development Middle Out Development starts at the middle level with a very high-level language. It is just another name for language-oriented programming.

2.2.4. Workflow Comparision with General-Purpose Languages

Dmitriev [7] shows the different workflows of DSLs and general-purpose languages which can be seen as their opposites because they can be used to write software for many different application domains. The DSLs are written by using language-oriented programming. We start with the assumption that we have a task for which we have to find a solution. A program is needed (source) that then gets executed.

General-Purpose Programming Language

With a general-purpose language, a conceptual model of the solution is formed. A program is written that maps the solution into the programming language. After

this manual part, the code gets compiled and an executable is created. These last steps are automated.

Language-oriented Programming

In language-oriented programming, the conceptual model of the solution is also first formed. The solution then gets mapped into specific domain special languages. The generating and/or compiling of the code and creation of the executable is automated.

The mapping of the problem domain to the solution problem is the critical part that language-oriented programming can improve. The terms of the concepts and notions of the problem that should be solved can be used to describe and the ideas don't have to be translated into something that a general-purpose language can understand.
One platform that supports language-oriented programming is the language workbench MPS.

2.3. JetBrains MPS

Jetbrains MPS stands for Jetbrains Meta Programming System and is a Language Workbench developed by JetBrains. It uses a projectional editor and allows the creation of composable languages that can be edited in the same environment. It contains a DSL called *com.mps.baselanguage* that is a reimplementation of Java. There are a lot of extensions for this language. Language definition languages help language designers create new languages.

2.3.1. Terminology

AST A data structure in the form of a tree that represents hierarchies of nodes. These nodes form relationships between themself.

Module A top-level element in an MPS project that groups together multiple models. There are three types of modules: languages, solutions, and devkit

Model A lower-level element that groups concepts.

Concept Describes the structure of a syntax element.

Node A concrete instance of a concept.

Runtime Solution A solution that is required by a language.

Aspect An aspect describes a specific part of a language. These are some of the most common aspects that are used in this project:

Structure Uses concepts to describe the structure of the language AST.

Editor Describes how the AST will be presented in the editor.

Action Describes the completion menu customizations specific to a language (when Control + Space is pressed).

Constraints Describes the constraints on AST.

Behavior Describes the behavioral aspect of AST.

Type System Describes the rules for calculating types in a language.

Figure 2.2 shows a possible combination of aspects.

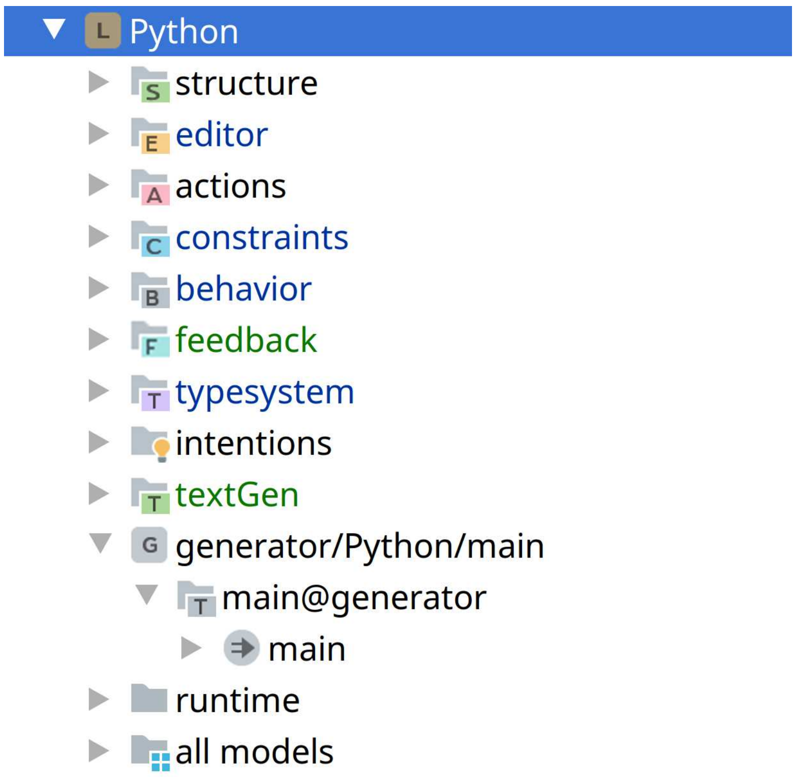


Figure 2.2.: Aspects of a language

2.3.2. Choice of Language Workbench

The first decision was to decide on a language workbench that could be used to develop a *DSL*. There are already two studies ([10] [9]) that compare and evaluate

different *LWB* implementations that were used to decide which LWB to use for this language. There are different features which should be supported by the LWB:

Notation
The LWB has to support textual and graphical notation.

Semantics
The LWB should be able to do model-to-model transformation or model-to-text transformation.

Validation
The LWB should be able to validate types, the structure of models and the naming of concepts.

Testing
The DSL can be tested within the LWB.

Composability
The LWB allows extending of languages and other DSLs.

Editing Mode
The editor can be a projectional editor or the editing is done as free-form.

Syntactic Services
The LWB should support syntactic completion.

Semantic Services
The following semantic services must be supported: reference resolution, semantic completion, and error marking.

All the mentioned features are supported by MPS. It's most prominent feature is the projectional editor.

2.3.3. Domain-specific Modeling

According to Kelly and Tolvanen [15], *DSM* allows a new level of abstraction beyond programming. The solution to a problem is directly specified using the concept of the problem domain and code is generated from this model. Source models are used instead of source code. It provides an automatic mapping from high-level concepts to low-level concepts.

Benefits

Productivity A higher level of abstraction leads to more productivity which is difficult to measure outside of research settings. It is mentioned that DSM is on average 300 - 1.000 % more productive than manual coding or general-purpose modeling languages. Adopting this new approach takes time and productivity increases steadily with time.
Maintaining software consists of bug fixes, reaction to new or changed requirements or environmental changes. Bug fixes are becoming less frequent because the code is generated.
Companies gain various advantages as a result of the increased productivity: They shorten the development time and can release new products quicker. Outsourcing is not needed as often and the development cost also decreases. Faster feedback loops from customers are another advantage. The models are closer to the requirements and changes can be made later in the process. New variants of a product can be developed quickly and new target customers can be reached.

Quality The quality of applications increases because the modeling language can analyze the model and check its correctness and search for other problems.
Generators also provide a way to map from a problem domain to a solution domain. The focus shifts from traditional design and implementation towards requirement capturing. The second domain is often on a lower level of abstraction and the code doesn't have to be edited manually after the generation. Nonetheless, the level of quality increment depends on the experience of the developers that define the modeling language, generators and supporting tools.
The mapping to the problem domain is also so close that customers and domain experts can participate in the process.
Testing is done by experienced developers and can be partly automated by DSM.

The application architecture is enforced at the design stage. Coding errors like forgotten initialization don't exit in DSM because the code is automatically generated. As a result, the style of code is also similar for different developers. DSM improves reuse of code because developers can use existing design, libraries or platform code. A model checker can validate the model at the domain level. Changes in the specifications are applied to domain concepts and don't cause non-domain-related problems.

Language Designers and Developers

Normally developers manually do the mapping from the problem domain to the solution problem. The quality of implementations varies greatly among them. In DSM only an experienced developer is responsible for this mapping. He defines the domain rules and can force guidelines and best practices for other developers. As the abstraction increases and gets closer to the problem domain, the number of potential developers increases because the complexity is hidden and less knowledge is necessary. In extreme cases, the domain expert can develop applications without the aid of a developer. Furthermore, less training and guidance is needed. Learning the implementation domain and mappings between domains is not required anymore. Instead, the focus should lie only on the problem domain.

2.3.4. Projectional Editing

Projectional editors don't use the traditional text-based approach in which a file is parsed and the AST is created. Instead, it skips the first step and allows to directly edit the AST.
It was shown by Voelter et al. [28] that projectional editing can be efficient for basic code-editing tasks but that advanced editing requires a lot of experience and understanding of the underlying structure. Therefore it has to be a goal to keep the DSL simple to minimize the amount of editing that has to be done by the user. There is ongoing research to make developing projectional editors more efficient. One of these solutions is called grammar cells [28]. They allow language designers to use declarative text notations which specify their interaction with the projectional editor. They also improve the usability of the editor.

Potentials and Limitations of Projectional Editors

Gagnon came to the following conclusions [13]:
The AST consists of nodes that are all connected through relationships which means that the user knows what he can and can't add to the program because the language imposes constraints through these relationships. New users can quickly start editing without any programming knowledge and documentation because the editor can guide them through the process.
This is also a drawback because people who are coming from a text editor background are not used to be restricted and this can lead to frustration. Berger et al. [1] showed that these drawbacks will fade after a few hours of using a projectional editor.
Using the AST allows rich syntaxes because the information can be presented not only in text form but also in graphical and symbolic form. The syntax can even be switched while editing which improves the traditional editing process. It also has its downsides: the different projections can be confusing and produce ambiguities in the language.
Languages in MPS can be combined. They can extend other languages or be extended.

Teaching MPS

It was shown by Ratiu, Pech, and Dummann [21] which lessons can be learned from teaching MPS. They noticed that after initial training the best approach is to provide continuous coaching for example in the form of hackathons where an experienced MPS developer works together with newcomers.
MPS can unfortunately not be easily learned because it has a steep learning curve. It is important to understand the basics before continuing with advanced topics.
Teaching trainees to implement languages is also easier than learning good language design because it requires more experience and technical knowledge. They also found out that their trainees were always concerned about industrial pragmatic topics like test-driven development and they would see this topic as a showstopper for integration MPS in their development environment if they couldn't support their concerns.

2.3.5. Notations

MPS supports different notations. A lot of them were contributed by a third party and described by Voelter and Lisson [26]:

Textual Notations The standard notation of programming languages.

Mathematical Symbols Support for mathematical notation.

Tables Two-dimensional representation of data in the form of tables.

Free Text Editing Editing without a constraining structure.

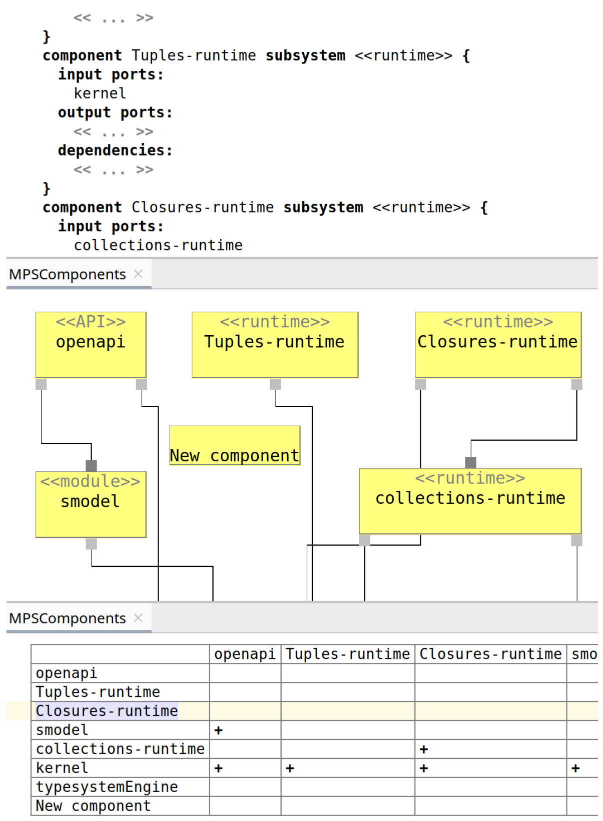
Margin Cells Cells that can display an editor and are located beyond the right editor margin.

Graphics Graphical notations.

Custom Cells Cells that the user can write himself.

Mixed Notations Mixing of different notations.

Multiple Editors One concept can define multiple editors. Figure 2.3 shows a textual, graphical and tabular editor for the same concept.



```
    << ... >>
}
component Tuples-runtime subsystem <<runtime>> {
  input ports:
    kernel
  output ports:
    << ... >>
  dependencies:
    << ... >>
}
component Closures-runtime subsystem <<runtime>> {
  input ports:
    collections-runtime
```

MPSComponents
<<API>> openapi
<<runtime>> Tuples-runtime
<<runtime>> Closures-runtime
New component
<<module>> smodel
<<runtime>> collections-runtime
MPSComponents

	openapi	Tuples-runtime	Closures-runtime	smo
openapi				
Tuples-runtime				
Closures-runtime				
smodel	+			
collections-runtime			+	
kernel	+	+	+	+
typesystemEngine				
New component				

Figure 2.3.: Multiple Editors for the same Concept

Partial Syntax The editor doesn't show the whole AST.

Query List Gets nodes that are outside of the scope of the current editor.

Tooltips They can contain editors.

Conditional Editors Shows an editor only if its condition is true.

Annotations The render additional syntax around an existing editor.

Read-only Contents Editors where the content can't be changed.

2.3.6. Case Studies

MPS is used in many different domains such as embedded systems, automotive, finance, and healthcare. The projects that were chosen for the chapter are all known in the MPS community. They all have articles published about them and are all of none trivial size.

Mbeddr

Mbeddr [29] is both a collection of languages and an IDE that is based on JetBrains MPS. It is meant to be used as an IDE for embedded software engineering. At its core, it implements a cleaned-up version of the C99 standard of the C programming language. It also contains a lot of other languages that simplify development.
It supports the following features:

Reporting and Logging Logging statements can be selectively disabled if needed. Their output is redirected to *sterr* but can also use other systems like serial ports.

Testing Tests can be group intro suites and support different assert statements. There is also an extra DSL for describing mocks.

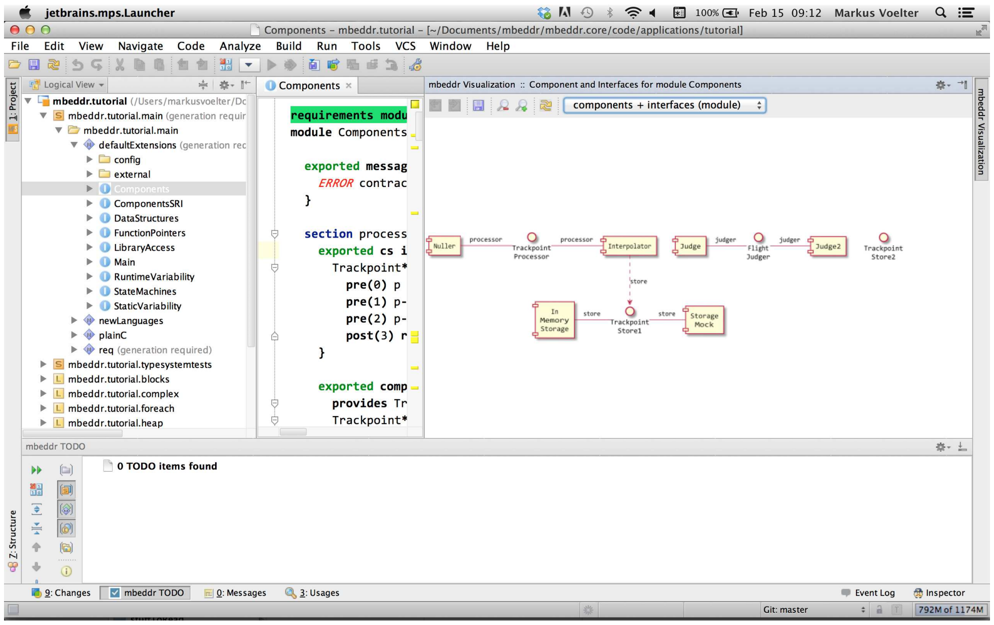
Physical Units Physical units can be used in code and are also supported by the type system.

State Machines They can be triggered by C code and can use different notations: text, tables and visual notation.

Interfaces and Components Different levels of modularizations are supported and can be visualized.

Requirements, Tracing and Docs There is a document-like language for describing requirements. Docs can also be created by using a DSL.

Product Line Variability There is support for product line variability which is often used in the development of embedded systems.



```
requirements modu
module Components

  exported messag
    ERROR contrac
  }

  section process
    exported cs i
      Trackpoint*
        pre(0) p
        pre(1) p-
        pre(2) p-
        post(3) r
    }

    exported comp
      provides Tr
      Trackpoint*
```

Figure 2.4.: Mbeddr [18]

Formal Verification Several verification tools can be used to check the written code. There is support for model checking of state machines, model checking/data flow analysis of C code for contract analysis as well as SMT solving for decision tables and feature models.

Execution and Debugging Programs can be executed and debugged by using the tools that are provided by MPS.

Version Control SVN and git can be used as a version control provider. The models are stored as XML files that are read and persisted by the language workbench.

Figure 2.4 shows a few of those features.

MetaR

MetaR [4] [5] implements the R language for MPS and is part of the NYoSh Analysis Workbench [22]. It has different target audiences:

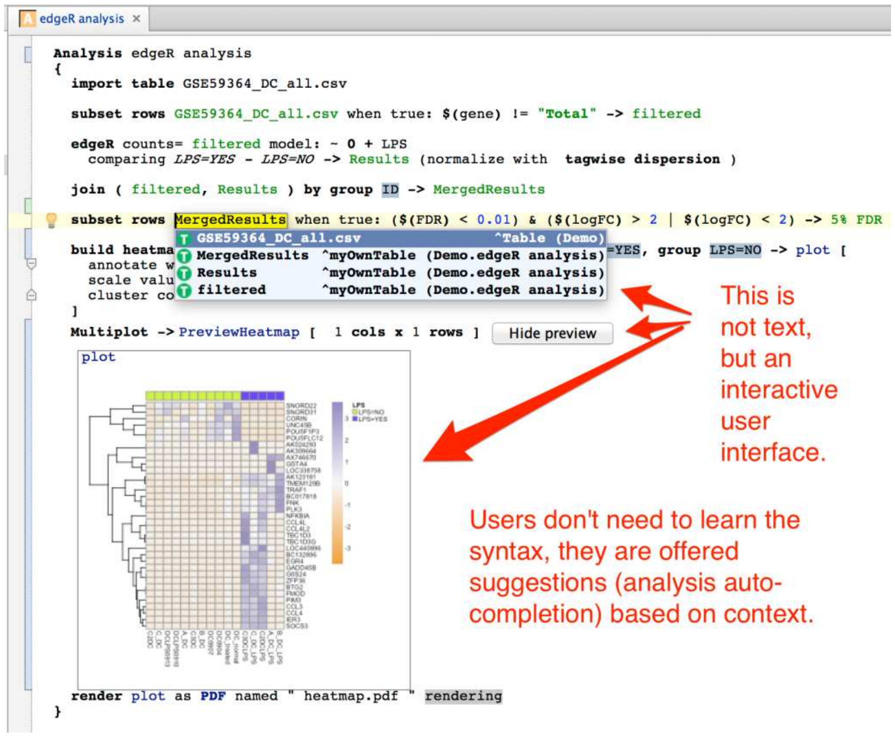
Biologists with limited programming experience

Bioinformaticians that need to do the repetitive analysis that can be optimized by using DSLs.

Bioinformaticians that want to include new analysis methods into MetaR analyses. As a result, users can use these methods of analysis experts in a more user-friendly way.

R Programmers that want to experience with languages that are composable and extendable.

Data can be transformed with different statements and displayed with tables. It can be visualized with typical data analysis visualizations like boxplot, histogram and Venn diagrams.Figure 2.5 shows a typical script in MetaR.



```
Analysis edgeR analysis
{
  import table GSE59364_DC_all.csv

  subset rows GSE59364_DC_all.csv when true: $(gene) != "Total" -> filtered

  edgeR counts= filtered model: ~ 0 + LPS
    comparing LPS=YES - LPS=NO -> Results (normalize with  tagwise dispersion )

  join ( filtered, Results ) by group ID -> MergedResults

  subset rows MergedResults when true: ($(FDR) < 0.01) & ($(logFC) > 2 | $(logFC) < 2) -> 5% FDR
              GSE59364_DC_all.csv                ^Table (Demo)
  build heatma MergedResults  ^myOwnTable (Demo.edgeR analysis) =YES, group LPS=NO -> plot [
    annotate w Results        ^myOwnTable (Demo.edgeR analysis)
    scale valu filtered       ^myOwnTable (Demo.edgeR analysis)
    cluster co
  ]
  Multiplot -> PreviewHeatmap [  1 cols x 1 rows ]

  render plot as PDF named " heatmap.pdf " rendering
}
```

This is not text, but an interactive user interface.

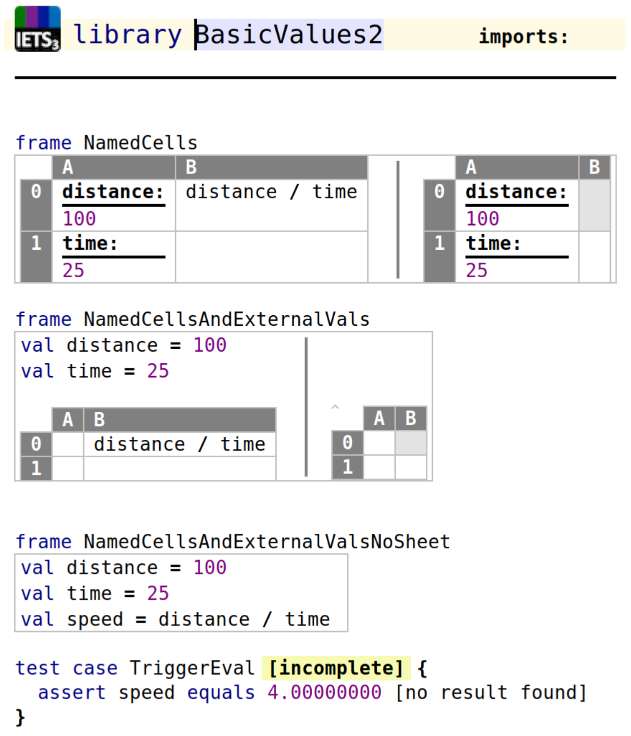
Users don't need to learn the syntax, they are offered suggestions (analysis auto-completion) based on context.

Figure 2.5.: MetaR [12]

To make research more reproducible, generated R scripts can be executed in an R environment that is installed inside a docker image or executed locally.
When parts of a script are changed, only those parts have to be re-executed. This technique called instant refresh saves a lot of time by checking which statements have changed and only executes those changes.
The implemented R language is complemented by other languages that implement specific R packages. This is a list of some of the packages that are partially incorporated: EdgeR, Limma Voom, Sleuth and Biomart.

KernelF

KernelF [30] is a functional language implemented in MPS. It is composable and can be embedded in other languages. Its design goals are simplicity, extensibility, embeddability, robustness, and portability. Figure 2.6 shows an example editor.



```
library BasicValues2                imports:

frame NamedCells
```

	A	B
0	**distance:** 100	distance / time
1	**time:** 25	

	A	B
0	**distance:** 100	
1	**time:** 25	

```
frame NamedCellsAndExternalVals
val distance = 100
val time = 25
```

	A	B
0		distance / time
1		

	A	B
0		
1		

```
frame NamedCellsAndExternalValsNoSheet
val distance = 100
val time = 25
val speed = distance / time

test case TriggerEval [incomplete] {
  assert speed equals 4.00000000 [no result found]
}
```

Figure 2.6.: KernelF

It contains three basic types: boolean, number and string and supports basic unary and binary operators. All expressions and statements are checked by the type system. Additionally, it contains null values and option types. Error handling is done using attempt types, a base type that represents the payload.
Functions, extension functions, function types, closures, function references, and high-order functions are also supported.
Covariant collections are also implemented in KernelF. *list*, *set* and *map* are subtypes of *collection*.
There are different forms of structured data: Tuples can be accessed the same way as arrays. Records are also structured and can be navigated using path expressions. Enums are implemented as a list of literals.
All values in KernelF are immutable and are used functionally.
Support for unit tests is also built-in.
KernelF uses the MPS type system but can also check constraints at runtime which can be attached to different nodes. Additional information (type tags) can be attached to existing types.

As a functional language, none of the expressions in KernelF have side effects. Nonetheless, it has support for stateful computations.
The following tools and actions are implemented in MPS:

- Interpreter
- Read-Eval-Print-Loop
- Debugger
- Test Execution
- Coverage Measurement
- Test Case Generation
- Mutation Testing

2.4. Good Language Design

DSL design is hard because the DSL has to cover the indented domain, be consistent and provide the right level of abstraction. Voelter et al. [27] provide seven dimensions that language designers should be familiar with:

Expressivity DSL can lead to increased expressivity in comparison to general-purpose languages. Programs are normally shorter because they are written closer to the solution domain. Language designers have to make assumptions about the target domain and incorporate their information into the language and execution strategy. Voelter et al. have defined:

A language L_1 is more expressive in domain D
than a language $L_2(L_1 \prec_D L_2)$,
if for each $p \in P_D \cap P_{l_1} \cap P_{L_2}, |p_{L_1}| < |p_{L_2}$ where
$|p_L|$ is the size of the program p in language L

The language should maximize expressivity but also cover enough of the target domain. The domain has to be fully understandable and structured by the designer to build a DSL. Expressivity should also increase in the domain hierarchy the higher the level of abstraction gets. Abstractions can be part of the language (linguistic abstractions) or expressed by concepts of the language (in-language abstracts).
A standard library can collect these abstractions so that they can be used by all users. The language itself is kept small with this approach.
Linguistic abstractions can be easily transformed to the target domain and make the languages suitable for domain experts. The abstractions need to be known beforehand or frequent changes to the DSL are necessary.
In-language abstractions allow more freedom because users can build new blocks with them but they need to be trained for this building process. Both approaches should not be mixed. A standard library is a compromise between both types where one developer creates abstractions that can be used by other developers.
Changing concepts of a language should node break existing models and migration techniques should be used if they are available. Backward compatibility and deprecation are topics that language designers have to keep in mind. DSLs should also hide algorithmic details if possible. The platform and execution engine should not affect the design of abstractions used in a language.

Coverage Ideally, the target domain is covered 100 %. A requirement for this goal is that the domain is well-defined and it is known what full coverage for this domain looks like. Sometimes only a subset of the domain is implemented on purpose for example because the domain is too big and complicated.

Semantics Semantics are divided into two types: Static semantics are realized with type system rules and by constraints. Constraints check the properties of a model and return either true or false. They should also check models with are structurally and syntactically correct as it is always the case with projectional systems. Type systems are a specific kind of constraint checking. Constraints can be enforced on multiple levels. The basic level is always enforced and the other levels are often optional and only checked when some conditions are met.
Execution semantics describe the behavior of an executed program. The execution engine must also be checked for correctness.
Unit tests should be written and coverage of the code should be kept high. When starting to write tests, tests for corner cases should be written first.
Test cases can also be written in the tested language by extending it to support test expressions. This is especially useful when there are multiple execution engines where the semantics of all generated representations should be the same and synchronized.
Unintended behaviors of the generated general-purpose program can only be found by trying to exploit the weakness of a program (penetration testing).
Transformations can also be done in multiple stages where the overall transformation is split into a chain of transformations that are applied one after another (cascading). An advantage of this approach is that a bigger problem can be divided into smaller problems and stages can be potentially reused.
Transformations also allow optimizations that result in efficient code but the techniques can be hard and expensive for the language designer especially when optimizations should be done on a global level. The goal is to have code that is efficient enough in most cases. Edge cases have to be written manually.
The generated code should also not be seen as disposable but treated the same way as manually written code and follow a reasonable coding standard. There is a tradeoff between well-structured code and complexity and generators that has to be considered. When code is highly optimized this rule doesn't have to be followed.
Instead of using transformations, an interpreter can act on the DSL. The level of abstraction must be suitable for the interpreter. Both approaches have advantages and disadvantages. Generated code can be inspected and

debugged. These tasks are harder for interpreters because they are meta programs. Performance and optimization are also in general better when using code generation because interpreters add another layer to the execution of the program. Generated code can be better matched to the target platform because it looks the same way as manually written code. Modularization can also be easier achieved with this approach. There are also two advantages of interpreters: The turnaround time is better and runtime changes are possible while the target system runs.
Sometimes the expressiveness of a language should be reduced so that it can be analyzed more easily. Formal verification tools can then be used to check the models. If the meaning of a DSL is not clear to the user, documentation in a tutorial style, test cases or simulators can help them to get a better understanding of a language.

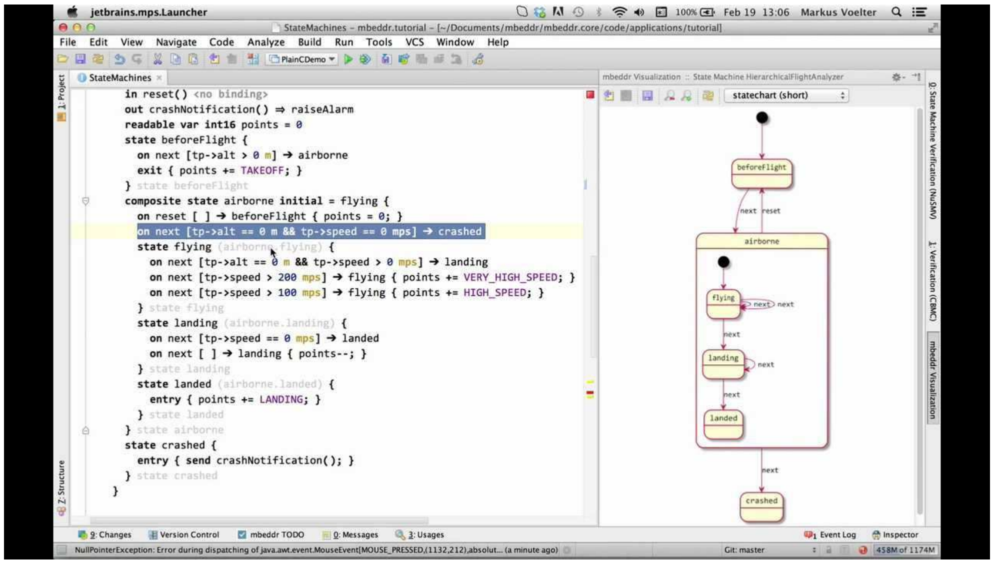
Separation of Concerns A domain might be composed of different concerns where each concern covers a different aspect of the domain. The language designer has to decide if a single language should address all concerns ore multiple DSLs address one or more concerns. Cross-cutting concerns which don't fit in the modularization approach can split into different classes: Some of them can be handled by the execution engine, others can be modularized on a DSL level. If the concerns cut across the executable system and can't be modularized on the DSL level, code has to be inserted in all relevant aspects manually or by the generator (weaving rules).
Projectional editors allow different views on programs and language designers have to choose the most suitable view.

Completeness Completeness refers to degrees to which a DSL can express programs that can be executed without a need for additional information. Incomplete DSL requires additional information like configuration files or code written in a lower-level language to be able to execute the program. If only a structure is generated the code has to be manually implemented by a developer. A DSL can also depend on a framework for executing the code. Another challenge can be roundtrip transformation where its original program can be completely recovered from the transformed program.

Language Modularization Language modularity can be achieved with four different composition techniques: referencing, extension, reuse, and embedding. A language designer has to decide if the language should consider possible composition partners when the language is developed (language dependencies) and if two languages can be mixed (Fragment structure).

Reusing languages is very important to avoid DSL hell, where developers create half thought-out DSLs that are not compatible with other DSLs and implement the same features.

Syntax The concrete syntax is one of the most important aspects. It can decide it a DSL gets accepted by the user community. Especially in the business application, the notations must work for the domain. The syntax should be writable, which means that the user doesn't have to type much and that it is concise.
Good editing support by an IDE can improve writability. In Figure 2.7 a good example can be seen. The syntax should be readable. According to the paper, a more concise syntax is not necessarily more readable, because context may be missing. The syntax should be learnable. This is particularly important for new users, who can learn it by exploring it in the IDE. It should also be effective in expressing common problems after users have learned the language.


jetbrains.mps.Launcher
StateMachines – mbeddr.tutorial – [~/Documents/mbeddr/mbeddr.core/code/applications/tutorial]
File Edit View Navigate Code Analyze Build Run Tools VCS Window Help
PlainCDemo
StateMachines
mbeddr Visualization :: State Machine HierarchicalFlightAnalyzer
statechart (short)
beforeFlight
next
reset
airborne
flying
next
landing
landed
crashed
9: Changes
Version Control
mbeddr TODO
0: Messages
3: Usages
Event Log
Inspector
Git: master

```
    in reset() <no binding>
    out crashNotification() ⇒ raiseAlarm
    readable var int16 points = 0
    state beforeFlight {
      on next [tp->alt > 0 m] → airborne
      exit { points += TAKEOFF; }
    } state beforeFlight
    composite state airborne initial = flying {
      on reset [ ] → beforeFlight { points = 0; }
      on next [tp->alt == 0 m && tp->speed == 0 mps] → crashed
      state flying (airborne.flying) {
        on next [tp->alt == 0 m && tp->speed > 0 mps] → landing
        on next [tp->speed > 200 mps] → flying { points += VERY_HIGH_SPEED; }
        on next [tp->speed > 100 mps] → flying { points += HIGH_SPEED; }
      } state flying
      state landing (airborne.landing) {
        on next [tp->speed == 0 mps] → landed
        on next [ ] → landing { points--; }
      } state landing
      state landed (airborne.landed) {
        entry { points += LANDING; }
      } state landed
    } state airborne
    state crashed {
      entry { send crashNotification(); }
    } state crashed
  }
```

```
NullPointerException: Error during dispatching of java.awt.event.MouseEvent[MOUSE_PRESSED,(1132,212),absolut... (a minute ago)
```

Figure 2.7.: Example for a good choice of syntax [17]

These design choices can be uncombinable and tradeoffs have to be made. One solution to this issues is using multiple different notations for the same concrete syntax. Readabilitiy can be improved by creating reports and visual-

izations which are readonly representations of a part of the model.

2.5. Summary

This chapter gave an introduction to the various topic areas that are mentioned in this work and gave an overview of the state of the art.
Cronjobs were described as well as alternatives that already exist.
Language-oriented programming and Jetbrains MPS were introduced to the reader as well as the importance of good language design. The next chapter will explain the thought process of coming up with the implementation.

3. Concept

The project should consist of two parts: an MPS project and a Java server program that replaces the cron daemon. This program will be a Java scheduler at its core. The MPS project should contain a DSL for describing cronjobs and also a utility model that helps to send jobs to the scheduler. The data flow is displayed in Figure 3.1.

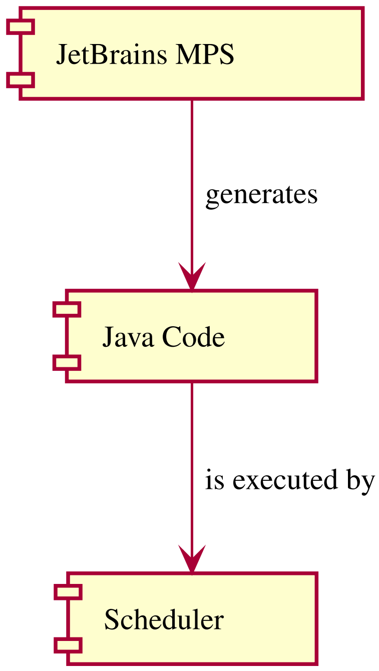


Figure 3.1.: Data Flow

3.1. DSL in MPS

The DSL has a few tasks that it should accomplish:

Simplicity vs Domain It should be easy enough that end users can understand it and develop scripts without or with little documentation. The language must be powerful enough that domain experts can describe their business logic in it.

Declaration of Cronjobs It should be possible to declare cronjobs and suitable triggers in this DSL.

Support for Time- and Event-based Triggers The user should be able to use the classical cronjob syntax but also use an additional syntax for reoccurring jobs.

UI for Scheduling of Jobs The DSL should include buttons that allow users to perform administrative tasks like scheduling and pausing jobs.

Generation of Code The LWB should be able to generate Java code that can be executed by the scheduler.

Conditional Jobs The DSL should allow adding conditions to jobs. These jobs are only executed when the condition evaluates to true.

Creation of Job Instances The *Trigger* concept should be able to create job instances and fill them with the necessary information so that they can be sent to the scheduler.

3.2. Usage of Features in MPS

MPS has a lot of features that can be used for developing languages, checking for errors and many other applications. Table 3.1 gives an overview of all the features which will be used and which will not be used.

Feature	Used	Comment
Constraints	yes	child and ancestor rules
Typechecking	yes	interference and checking rules
Text Generation	no	
Model to Model Generation	yes	generation of Java code
Custom Persistence	no	not necessary
Runtime Models	yes	
Stubs and Libraries	yes	needed for code generation and checking
Intentions	yes	
Execution and Debugging of Code	no	code is executed by scheduler
Dataflow Analysis	no	DSL too simple
Refactorings	no	
Migrations	no	
Testing	yes	

Table 3.1.: Used MPS features

3.3. Utility Model in MPS

The utility model should perform additional tasks and support the DSL:

Translate DSL for Triggers The model should transform the *Trigger* concepts of the language to Java objects that are understood by the scheduler.

Display Communication with the Scheduler Actions that are executed by the user or messages that are received from the scheduler should be displayed with a suitable user interface component.

Model Validation The scripts should be checked for errors before sending information to the scheduler.

3.4. Java Server

The program should use a Java software library that implements a scheduler. It also must perform the following tasks:

Save Execution Stats in a Database After executing a job, it must save meta-data about the job in a database (e.g. exit code, the runtime of jobs, etc.)

Compile and Execute It must compile the code that it receives from MPS and execute it.

Provide Jobs with Execution Stats The scheduler must query the database for execution stats of previous jobs and invoke the job object with this data as a parameter. The reason that the server needs to provide this data is that a script in MPS can use this information in the precondition before running a job.

3.5. Summary

Both parts of the projects were explained: The DSL *Swino* and the scheduling server *SwinoServer*. The features of the language workbench that were used not and used were presented and a short overview of the responsibilities of the Java server was given. Duties of the utility classes of the DSL were shortly explained. The next chapter will explain the implementation details of the project.

4. Prototyping

The project is divided into the MPS project called *Swino* and the Java Server called *SwinoServer*. An overview can be seen in the plantUML component diagram in Figure 4.1.

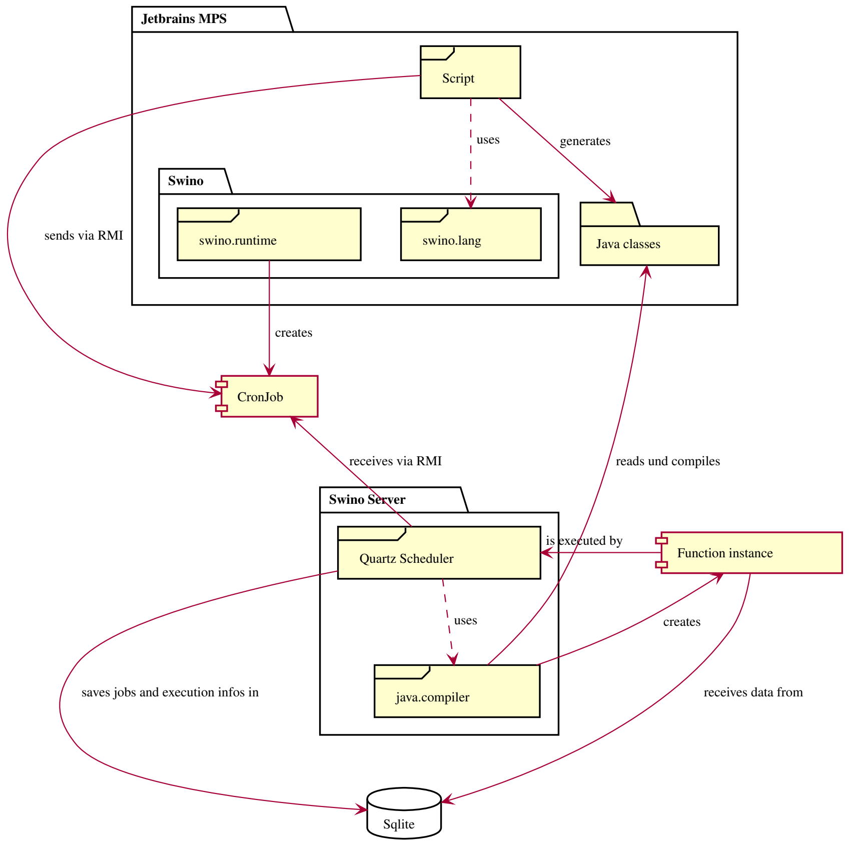


Figure 4.1.: Project Component Diagram

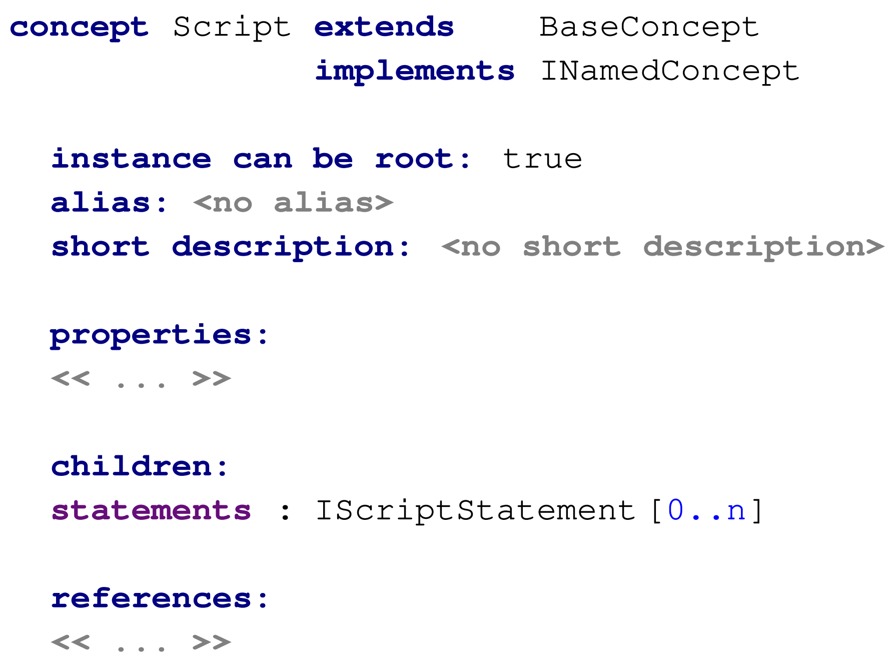
.

4.1. Swino

The MPS project consists of the language *swino.lang* and the runtime model *swino.runtime*. Users can create new models that contain instances of the concept *Script*.

4.1.1. Language (swino.lang)

Figure 4.2 shows the concept *Script* that acts as a root node in a model. That means that it can be added to a model by the user. It can contain concepts that implement the interface *IScriptStatement*.



```
concept Script extends     BaseConcept
               implements INamedConcept

  instance can be root: true
  alias: <no alias>
  short description: <no short description>

  properties:
  << ... >>

  children:
  statements  : IScriptStatement [0..n]

  references:
  << ... >>
```

Figure 4.2.: Concept: Script

Figure 4.3 shows the concept *Job* that contains all information about a job: the command to execute, a description, a name, and a condition that decides if the job gets executed. The condition is a list of java statements that has to return a boolean value.

```
concept Job extends     BaseConcept
            implements INamedConcept
                       IScriptStatement

  instance can be root:  false
  alias: job
  short description: <no short description>

  properties:
  command      : string
  description  : string

  children:
  condition  : Condition[0..1]

  references:
  << ... >>
```

Figure 4.3.: Concept: Job

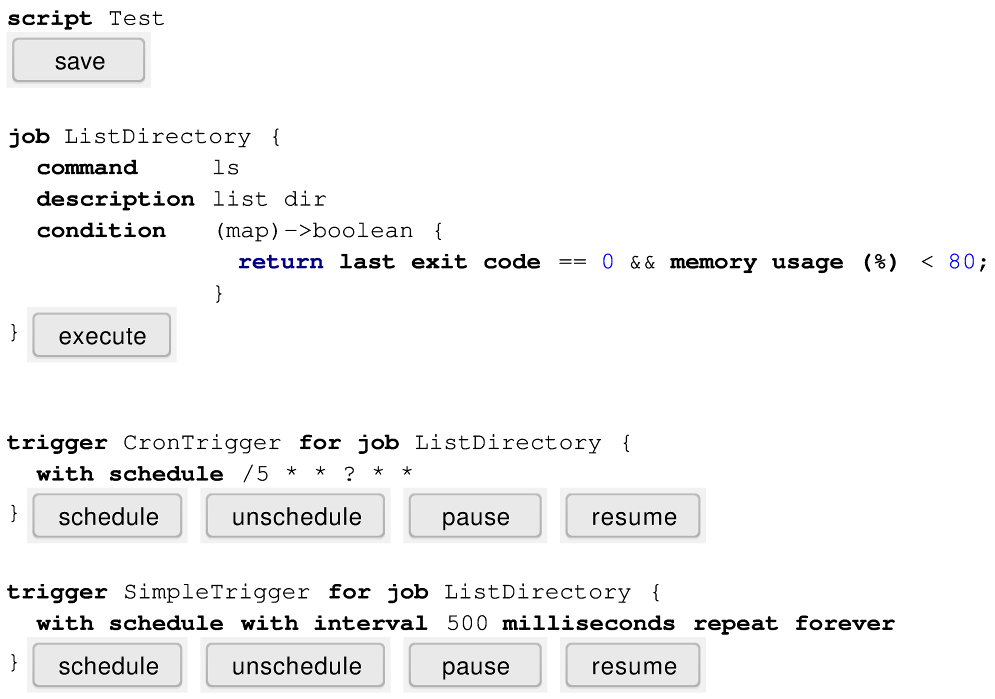
Figure 4.4 shows the concept *Trigger* that contains a reference to a node of type Job and an instance of the interface ISchedule which is a node that implements a scheduling pattern.

```
concept Trigger extends     BaseConcept
                implements  INamedConcept
                            IScriptStatement

  instance can be root:  false
  alias: trigger
  short description: <no short description>

  properties:
  << ... >>

  children:
  schedule  : ISchedule[0..1]
  job       : JobReference[1]

  references:
  << ... >>
```

Figure 4.4.: Concept: Trigger

4.1.2. Editor

The editor contains both text and graphical elements. The graphical parts consist of buttons of the GUI toolkit Swing which are embedded into the editor. Figure 4.5 shows how the user edits a script.

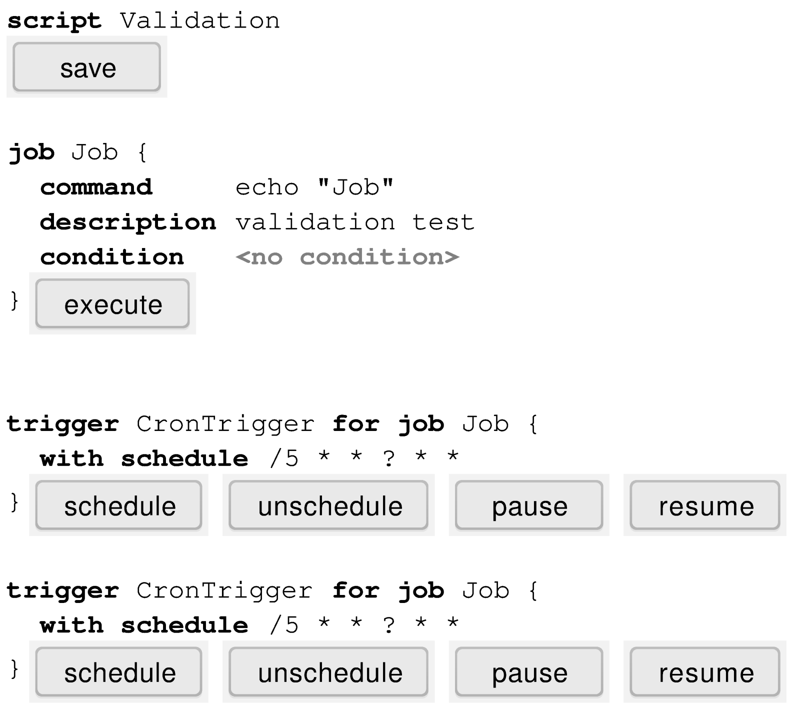


```
script Test

job ListDirectory {
  command     ls
  description list dir
  condition   (map)->boolean {
                return last exit code == 0 && memory usage (%) < 80;
              }
}

trigger CronTrigger for job ListDirectory {
  with schedule /5 * * ? * *
}

trigger SimpleTrigger for job ListDirectory {
  with schedule with interval 500 milliseconds repeat forever
}
```

Figure 4.5.: Editor for scripts

4.1.3. Type System and Constraints

The type system aspect contains so-called checking-rules that show errors when jobs and triggers are not unique within a script. The script in Figure 4.6 would for example show an error because the names are not unique. The system also checks the uniqueness of scripts within a model. Interference rules set the type of these custom expressions that can be used in the precondition of jobs. They have the same type as the generated Java expressions.



```
script Validation
```

```
job Job {
  command     echo "Job"
  description validation test
  condition   <no condition>
}
```

```
trigger CronTrigger for job Job {
  with schedule /5 * * ? * *
}
```

```
trigger CronTrigger for job Job {
  with schedule /5 * * ? * *
}
```

Figure 4.6.: The model checker detects that names are not unique. Squiggly red lines are drawn under the offending name (not shown in this graphic)

.

The cron schedule patterns are also validated and show an error if they are invalid. All node names are checked for emptiness.

4.1.4. Runtime model (swino.lang.runtime)

The runtime model contains a class *SchedulerManager* which prepares cronjobs and sends them to the server via RMI. It also renders some buttons of the language, displays popup messages which show messages from the editor and SwinoServer and checks the model for errors before sending commands to the server.

4.2. SwinoServer

The SwinoServer is the second part of the project which contains the scheduler that is used instead of the cronjob daemon.

4.2.1. Quartz Scheduler

Quartz is a job scheduling library written in Java and is mostly used in enterprise applications. It is open-source software that was first described in the book Quartz Job Scheduling Framework: Building Open Source Enterprise Applications [6]. It supports all the features that are needed for this project: The library can be embedded in a Java application and run as a stand-alone program. In this case, jobs can be sent via RMI to the scheduler. It supports cron based triggers but also more customizable triggers. The smallest time unit that can be used is milliseconds. Jobs can be implemented as Java classes and instantiated by the scheduler. It can also listen to job and trigger events. Jobs can be persisted to a database and continued after the server restarts.

Alternatives

According to the book's author Cavaness, some Java classes could be used instead (Java SDK Timer and TimerTask classes) which don't provide the same functionality as Quartz. He also advises against home-grown solutions and argues that creating a job scheduler is not an easy task. There are other commercial solutions available but Quartz Scheduler was chosen because it is open source and free.

Configuration

The configuration file for the server is the main file. It sets the name of the scheduler, RMI specific options, configures the threads that are used and configures the JobStore class. Additionally, a RMI policy file has to allow all permissions for the communication between the server and client and vice versa. The configuration file can be seen in Figure 4.7.

```
grant {
      permission java.security.AllPermission;
 };
```

Figure 4.7.: server.policy File

The path to this file must be set when starting the java server.
The client configuration file must contain the same instance name of the server and define the server's host and port so that it can connect to it.
The full configuration file for the server and client can be found in section A.2 and section A.3 respectively.

4.2.2. SQLite Database

According to Bhosale, Patil, and Patil [2], SQLite is a self-contained database that doesn't need a server or configuration. The code is open source and the database has bindings for all major programming languages. It doesn't have a separate server process and reads and writes from an ordinary file. The application interface is simple and doesn't have any external dependencies. There is also cross-platform supported for all major platforms.

4.2.3. Database Schemes

The Quartz scheduler library can store jobs in memory or databases. For this project, the JDBCJobStore was chosen. It keeps all the data in an SQLite database via JDBC. The tables are automatically populated in queried by the scheduler. A list of all tables including the table that stores information about the execution of jobs can be seen in Table 4.1.

Table Name	Managed by scheduler
QRTZ_JOB_DETAILS	true
QRTZ_TRIGGERS	true
QRTZ_SIMPLE_TRIGGERS	true
QRTZ_CRON_TRIGGERS	true
QRTZ_SIMPROP_TRIGGERS	true
QRTZ_BLOB_TRIGGERS	true
QRTZ_CALENDARS	true
QRTZ_PAUSED_TRIGGER_GRPS	true
QRTZ_FIRED_TRIGGERS	true
QRTZ_SCHEDULER_STATE	true
QRTZ_LOCKS	true
sqlite_sequence	true
CRON_EXECUTION_INFO	false

Table 4.1.: Database tables

Before using this project the schemes have to exist. The SQL code for creating them can be found in appendix A.1. The table *CRON_EXECUTION_INFO* is the only interesting table because the server manually populates and queries data from this table. Its schema can be seen in Table 4.2.

column	type
id	INTEGER
JOB_NAME	VARCHAR
GROUP_NAME	VARCHAR
FIRE_TIME	TIMESTAMP
META_DATA	VARCHAR

Table 4.2.: Database table *CRON_EXECUTION_INFO*

Its data is managed by an object-relational mapping library which allows the use of plain Java objects to manage the table.
It is difficult to find a good structure for optional data in databases that's why the meta-data about the job executions is stored as a string. The data is first collected as key-value pairs and stored in a GSON object. This object is then persisted as a string and stored in the database.
When the meta-data needs to be retrieved, the process is reversed: The data is first

queried from the database and then deserialized to a JSON object. In the last step, the object is converted back to key-value pairs.

4.2.4. From Clicking "schedule" to the Execution of a Job

After the user presses "schedule", the *Trigger* node creates an instance of the job class and fills it with data. The *SchedulerManager* connects to the *SwinoServer* via RMI. Then it checks if the trigger already exists. If it exists, it reschedules the job, otherwise, it schedules the job. The job contains only the class name of the generated file and its full path. After receiving the job, the Quartz scheduler saves the job in the database. When the trigger is activated, the *SwinoServer* unpacks the class name and path of the generated job file and reads the file into memory. Then the Java library joor [8] invokes the Internal Java compiler and compiles the loaded string to a Java object that implements the interface Function<T, R>. T is the type of the input to the function, and R is the type of the result of the function. Before executing the function, the server loads the last execution stats of the job into the input parameter. After the execution of the cronjob, the name, group and trigger fire time are saved in the database. Additional parameters like the exit status code are first grouped to a JSON object and then serialized and also saved in the same table as the other data.

4.2.5. Code Generation

To generate Java code a model-to-model transformation is necessary. The *Script* concept from the *swino.lang* gets generated to a *Class* concept of the language *com.mps.baselanguage*. After this step, the text generator is automatically invoked on the concepts of the *com.mps.baselanguage* which generates Java output files. This method implies that the model that contains a *Script* root node must be generated before a job can be sent to the scheduler. The user can a) built the containing model c) built the containing solution or press the save button at the top of the script. This button also builds the containing model.

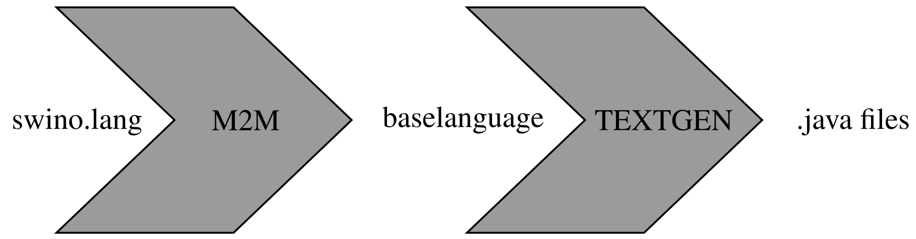


Figure 4.8.: Model Transformation Steps

The output of the *Script* node can be ignored. The most important concept is the *Job* concept that gets generated into a Java class through a root mapping rule. The generation template can be seen in Figure 4.9.

```
 root template
 input Job

public class $ Job  implements Function<HashMap, Integer> {

  $IF$ private boolean isConditionMet(HashMap map) {
         $COPY_SRC$ true;
       }

  @Override
  public Integer apply(HashMap map) {
    $IF$ if (!isConditionMet(map)) { return null; }
    string command = "$ command ";
    Process p;
    try {
      p = Runtime.getRuntime().exec(command);
      return p.waitFor();
    } catch (Exception e) {
      return -1;
    }
  }
}
```

Figure 4.9.: Root mapping rule for the concept: Job

A few special expressions which can be used in the condition block of a *Job* node are reduced to Java expressions. If a node needs an additional method present in the generated class, a so-called weaving rule can be used. Its purpose is to add extra nodes in the output model. The generation process is displayed in Figure 4.8.

4.3. Dependencies

SwinoServer used the build tool Gradle and has run time dependencies as displayed in Table 4.3:

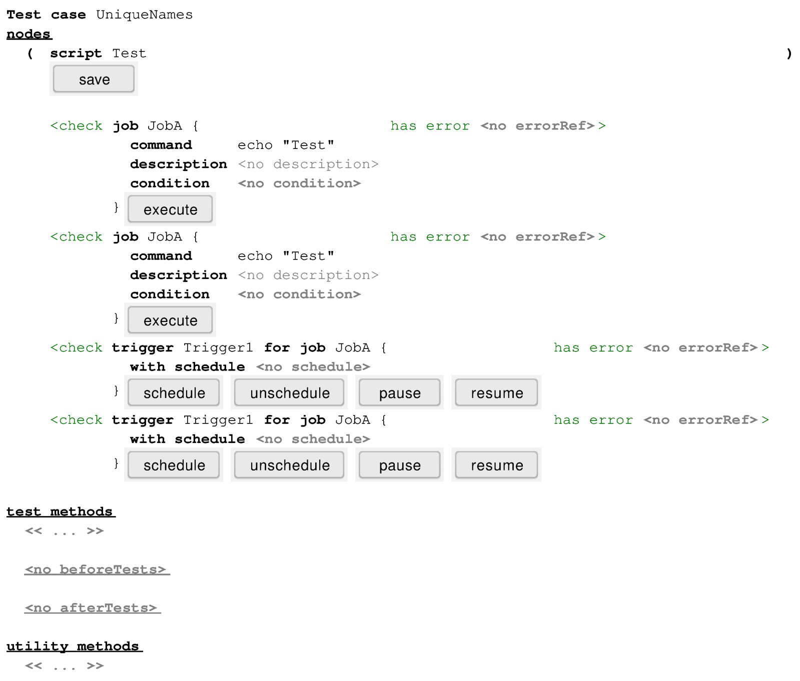
group	**name**	**version**	**description**
org.quartz-scheduler	quartz	2.3.2	scheduler that is used instead of the cron daemon
ch.qos.logback	logback-classic	1.2.3	logging library
org.jooq	joor	0.9.12	runtime compilation of Java code
org.xerial	sqlite-jdbc	3.30.1	SQLite database driver
com.github.goxr3plus	javasysmon2	8.0.0	library that reads system informations
com.j256.ormlite	ormlite-jdbc	5.1	lightweight object relational mapping library
com.google.code.gson	gson	2.8.6	string to json serializer/json to string deserializer

Table 4.3.: Dependencies

4.4. Testing

The different language definition aspects all have to be test differently:

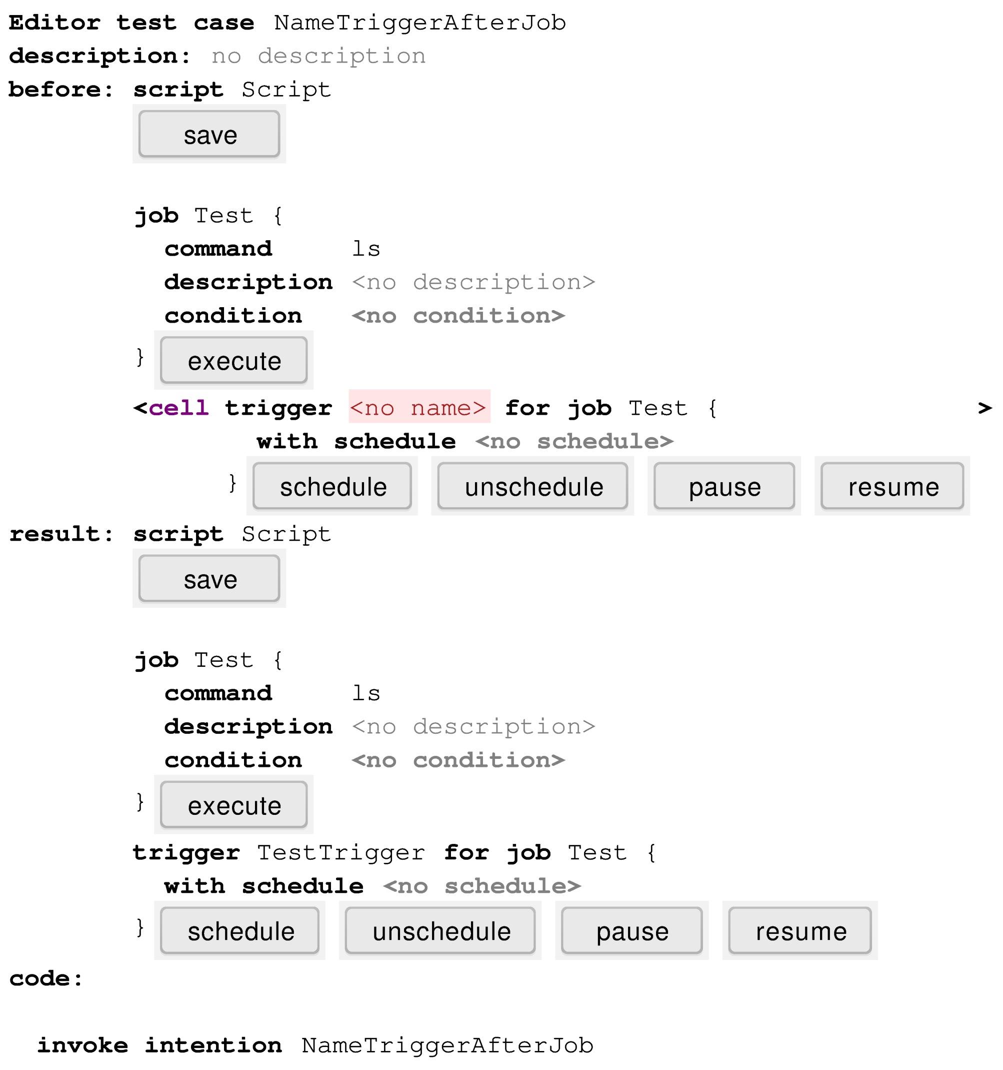
Structure and Editor Both aspects are tested by creating a sandbox model and checking if every node can be inputted and that the editor displays the correct values. Alternatively, *NodesTestCases* nodes could be used.



```
Test case UniqueNames
nodes
  (  script Test                                                                   )
     save

     <check job JobA {                     has error <no errorRef> >
              command     echo "Test"
              description <no description>
              condition   <no condition>
            } execute
     <check job JobA {                     has error <no errorRef> >
              command     echo "Test"
              description <no description>
              condition   <no condition>
            } execute
     <check trigger Trigger1 for job JobA {                  has error <no errorRef> >
              with schedule <no schedule>
            } schedule  unschedule  pause  resume
     <check trigger Trigger1 for job JobA {                  has error <no errorRef> >
              with schedule <no schedule>
            } schedule  unschedule  pause  resume

test methods
  << ... >>

  <no beforeTests>

  <no afterTests>

utility methods
  << ... >>
```

Figure 4.10.: Test: Unique Names

Intentions *EditorTestCases* nodes are used. You specify the state of the node before the action is executed and the state of the node after the action is executed. The test compares the structure of the two nodes and fails if they are not the same. Figure 4.11 shows such a test case.



```
Editor test case NameTriggerAfterJob
description: no description
before: script Script


         job Test {
           command      ls
           description  <no description>
           condition    <no condition>
         }
         <cell trigger <no name> for job Test {                  >
                   with schedule <no schedule>
                 }
result: script Script


         job Test {
           command      ls
           description  <no description>
           condition    <no condition>
         }
         trigger TestTrigger for job Test {
           with schedule <no schedule>
         }
code:

  invoke intention NameTriggerAfterJob
```

Figure 4.11.: Test: Name Trigger After Job

Constraints and Type System *NodesTestCases* nodes are used. You specify snippets of correct code that should not should an error or warning and snippets of invalid code where an error or warning should be reported.

Generator Generators can't be tested with built-in tools at the moment. Some known practices work: Generated models can be compared to expected out-

put by using a diff tool or changes can be seen in a version control system like git. Corner cases can also be checked to see if the generator successfully generates output or fails on them. The generated code should be compiled and executed to increase confidence in the correctness of the generated code. In this work, the last approach was used. When MPS compiles a model that uses these languages, Java code is generated and also compiled. When compilation errors occur, an error is thrown that the generation was not successful.

A new model of type test has to be created. It should contain a *TestInfo* root node that contains the path to the root of the MPS project. The *Swino* project uses the environment variable *swino_home* to find the path.
It must be set in File -> Settings -> Appearance & Behavior -> Path variables.

4.5. Summary

The chapter described the implementation details of the *Swino* language and the supporting runtime model and the Java server *SwinoServer*. The dependencies were listed and the testing methods were explained. The next chapter describes the evaluation process of the DSL with real users.

5. Evaluation

For the evaluation of the project a group of experts where chosen. Most of them didn't have experience with glsmps before but were proficient in using cronjobs and the crontab command. Most of them were system administrators and developers.

5.1. Cognitive Walkthrough

The experts will test the language in a pre-configured environment. The environment will have the Java programming language and JetBrains MPS installed.
The experts will test the *Swino* component of this project. The *SwinoServer* part will be already running in the background as a Java process.
The server doesn't need to be evaluated by the experts directly because he will interact with it through the DSL. The language workbench MPS will already be opened and the expert will have to follow this walkthrough:

Create a New Solution A module of type solution should be created in MPS and be named "Evaluation".

Create a New Model The expert needs to make a new model inside the solution. He can choose an appropriate name on his own.

Import the DSL He should import the language *com.github.fxlex.swino.lang* in the model properties.

Create a new Script A new root node of type *Script* should be added to the created model. It should be named *Echo*.

Add a Job that Succeeds The job should be named *Echo Success* and execute

a command that is provided by the tester. The command displays a notification in the right bottom corner of the screen.

Add a Job that Fails A second job should be added that has the same purpose as the first job but fails with an exit code 0. The command will also be provided by the tester. It should have a pre-condition that tells the scheduler to only run the job if the last exit code was 0.

Execute the Jobs The expert should click on "execute" below each job and check if they jobs work.

Create Triggers He should add a few triggers to the script. The triggers should reference the two declared jobs. Both simple and cron schedules should be entered by the expert.

Schedule and Unschedule Jobs The buttons below triggers should be used to schedule, unschedule, pause and resume jobs. The effect of the actions should be noticed by the expert.

Schedule without Saving the Model The expert should try to schedule a job but don't save the current model and observe what happens.

Play with the Scripts in the Sandbox An additional solution will be provided that contains sample scripts that can't be executed because they are only for demonstration purposes. The first script deals with converting images. the second script monitors the current server's memory usage and disk space. The first script trains a neural network to detect faces. The last script contains errors on purpose.
The expert should have a look at the scripts and try to make changes to them. He should provide feedback about the syntax that is used for declaring the jobs and triggers and tell the tester what he likes and dislikes about the DSL.

Create Nodes that don't have unique Names The expert should create jobs, triggers or scripts that have the same name as other nodes of the same type. The editor should highlight the errors which can be corrected by the expert.

Create a Trigger that is named after the Job He should use an intention on the *Trigger* node to name it after the referenced job.

5.2. Result

All experts were able to complete the tasks when given enough guidance. The experts that never used MPS before, understood all the steps to create a new script but they would need a few hours of training to getter better at editing their scripts without help. The results can be seen in Table 5.1 and Table 5.2.

Task	**Without Help**	**With Help**
Create a New Solution	90%	10%
Create a New Model	90%	10%
Import the DSL	80%	20%
Create a new Script	100%	0%
Add a Job that Succeeds	80%	20%
Add a Job that Fails	90%	10%
Execute the Jobs	80%	20%
Create Triggers	80%	20%
Schedule and Unschedule Jobs	100%	0%
Schedule without Saving the Model	100%	0%
Play with the Scripts in the Sandbox	90%	10%
Create Nodes that don't have unique Names	90%	10%
Create a Trigger that is named after a Job	90%	10%

Table 5.1.: Evaluation Results MPS Experts

Task	**Without Help**	**With Help**
Create a New Solution	70%	30%
Create a New Model	70%	30%
Import the DSL	50%	50%
Create a new Script	80%	20%
Add a Job that Succeeds	60%	40%
Add a Job that Fails	80%	20%
Execute the Jobs	70%	30%
Create Triggers	60%	60%
Schedule and Unschedule Jobs	100%	0%
Schedule without Saving the Model	100%	0%
Play with the Scripts in the Sandbox	70%	30%
Create Nodes that don't have unique Names	80%	20%
Create a Trigger that is named after a Job	80%	20%

Table 5.2.: Evaluation Results non MPS Experts

Creating a new solution and module was not an issue for anybody as well as creating a new script. The combination of textual and graphical elements within a script was a bit confusing and experts had to understand the importance of using autocompletion and using the keyboard instead of the mouse. Some struggled because they couldn't transfer their keyboard navigation skills to the projectional editor where textual and graphical elements where used.
Existing MPS experts didn't have any particular problems because they know the differences that projectional editing brings with it.
There were also issues that all experts hat in common. Most of them forgot to save the script and then they executed old versions of their scripts as a result. This problem is understandable because MPS experts normally don't know the concept of saving models. In this work, it is important because the jobs need to be generated. Furthermore, experts would need some documentation that is embedded in the language to better understand all of its features. For example, a trigger that uses the cronjob format for scheduling doesn't work the same way as a normal cronjob. The format that is used is based on the format that the Quartz Scheduler uses which is an extension of the cronjob format but includes seconds in its pattern. Experts tried to write a pattern but the validation failed because they failed to use the correct format. Other similar issues were observed (e.g. using whitespace in job names).

5.3. Summary

This chapter described the testing environment and components that were evaluated by the experts. The tasks that they had to perform were listed and the result was discussed. The next chapter will be the conclusion of the project and will mention future improvements.

6. Conclusion and Future Work

This chapter concludes the work and explains issues that could be resolved in future additions to this project.

6.1. Conclusion

As seen in the examples, real scripts can be developed with the *swing.lang* DSL. Although the DSL works for simple tasks and is expressive enough, a language designer would need to extend the language to make the models closer to the requirements of a project or domain. New expressions would have to be added to the language. They could then be used to refer to resources like database or meta-information about cronjobs. In the latter case, the meta-information first has to be added to the database in the scheduler.
The evaluation showed that builtin documentation is very important and needs to be added to the language. External documentation should also be provided.
Another issue was the save button which was often forgotten by the experts. Two solutions could be used to remove this button. The first approach would be to use run configurations in MPS which would generate the code first and then schedule the jobs automatically. The second approach would be to generate the models on change or after a specific interval. This solution might not be possible because the end of the generation phase can't be easily established.
In the evaluation, the environment was already installed and pre-configured, two steps which are not as easy for people who are not familiar with MPS and Java. These steps could be obstacles for new people who want to switch from cronjobs. The learning curve for cronjobs is much shallower and shorter in comparison to this approach. They can be mastered in a short amount of time because there aren't many concepts that have to be understood. In contrast, this approach requires A lot of training and good documentation is needed.
The author of this paper believes that the projectional editor and domain-specific

modeling could be a viable alternative to cronjobs if the language workbench would be adopted by more people because at the moment the learning curve for new users of such a DSL would be too high and the advantages of such an approach are not apparent to the end-user or domain expert. Good language design is also very important or the user experience with using such languages will be very poor.

6.2. Future Work

The quartz framework that powers the server component of this project has many options and features that were not implemented in this DSL (interruptable jobs, retrying failed jobs, transactions, and other features). Many features could be implemented in the future. One of the most important features would be setting a starting time for triggers that use the simple scheduling approach (not cron-based scheduling).
One problem that was not fully explored is the global uniqueness of job and trigger names. The uniqueness constraint is only enforced in the same model. If two models contain a script with the same and a job node with the same name, the scheduling of one of the jobs overrides the other job. This problem could be partially solved by including the full model name in the identifier but even this approach is not enough: model names don't have to be unique.
Cronjobs have support for automatic sending of emails in case a job fails. This feature was not implemented but could be a nice addition for a future version of the DSL.
Scheduling jobs is currently done manually and without priority. Priority based scheduling could be easily implemented by using the already existing feature of the Quartz Scheduler. Scheduling could also be done a lot smarter. Data about jobs could be saved in a graph database and analyzed by artificial intelligence. The best schedule could then be statically or dynamically decide based on the result.

6.3. Summary

This chapter gave some final statements about the work's conclusion and gave some hints about the remaining problems and possible improvements.

7. Bibliography

[1] Thorsten Berger et al. “Efficiency of projectional editing: A controlled experiment”. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 763–774.

[2] ST Bhosale, T Patil, and P Patil. “Sqlite: Light database system”. In: *International Journal of Computer Science and Mobile Computing* 4.4 (2015), p. 882.

[3] Frederick P Brooks. “No silver bullet”. In: *Software state-of-the-art* (1975), pp. 14–29.

[4] Fabien Campagne, William ER Digan, and Manuele Simi. “MetaR: simple, high-level languages for data analysis with the R ecosystem”. In: *BioRxiv* (2016), p. 030254.

[5] Fabien Campagne et al. “MetaR: simple, high-level languages for data analysis with the R ecosystem”. In: (2016).

[6] Chuck Cavaness. *Quartz Job Scheduling Framework: Building Open Source Enterprise Applications*. Pearson Education, 2006.

[7] Sergey Dmitriev. “Language oriented programming: The next programming paradigm”. In: *JetBrains onBoard* 1.2 (2004), pp. 1–13.

[8] Lukas Eder. *jOOR - Fluent Reflection in Java jOOR*. `https://github.com/jOOQ/jOOR`. 2011.

[9] Sebastian Erdweg et al. “Evaluating and comparing language workbenches: Existing results and benchmarks for the future”. In: *Computer Languages, Systems & Structures* 44 (2015), pp. 24–47.

[10] Sebastian Erdweg et al. “The state of the art in language workbenches”. In: *International Conference on Software Language Engineering*. Springer. 2013, pp. 197–217.

[11] User F.sb. *Cron vs systemd timers*. Unix & Linux Exchange. [Online; accessed Januar 31, 2020]. URL: `https://unix.stackexchange.com/a/281203/392155`.

[12] Fabien Campagne. *metaR - Snapshot*. [Online; accessed Januar 31, 2020]. URL: `https://campagnelab.org/files/MetaR_Better_Snapshot-1024x840.png`.

[13] Antoine Gagnon. "Analysing the potential and limitations of projectional editors in language engineering and code generation in Jetbrains MPS". In: ().

[14] Michael S. Keller. "Take Command: Cron: Job Scheduler". In: *Linux J.* 1999.65es (Sept. 1999), 15–es. ISSN: 1075-3583.

[15] Steven Kelly and Juha-Pekka Tolvanen. *Domain-specific modeling: enabling full code generation*. John Wiley & Sons, 2008.

[16] Carl Kessler and John Sweitzer. *Outside-in software development: a practical approach to building successful stakeholder-based products*. Pearson Education, 2007.

[17] mbeddr team. *mbeddr - screencasts*. [Online; accessed Januar 31, 2020]. URL: `https://i.ytimg.com/vi/aJ8d7ISFat4/maxresdefault.jpg`.

[18] mbeddr team. *mbeddr User guide - 7.4.9 Visualizing Components*. [Online; accessed Januar 31, 2020]. URL: `http://mbeddr.com/userguide/images/tutorial/v_components.png`.

[19] Jay F. Nunamaker, Minder Chen, and Titus D. M. Purdin. "Systems Development in Information Systems Research". In: *J. of Management Information Systems* 7 (1990), pp. 89–106.

[20] Lennart Poettering. "systemd.timer(5) - Linux man page". In: *systemd.timer - Timer unit configuration - Linux man page* (2019). [Online; accessed Januar 31, 2020]. URL: `http://man7.org/linux/man-pages/man5/systemd.timer.5.html`.

[21] D. Ratiu, V. Pech, and K. Dummann. "Experiences with Teaching MPS in Industry: Towards Bringing Domain Specific Languages Closer to Practitioners". In: *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. Sept. 2017, pp. 83–92. DOI: `10.1109/MODELS.2017.15`.

[22] Manuele Simi and Fabien Campagne. "Composable languages for bioinformatics: the NYoSh experiment". In: *PeerJ* 2 (2014), e241.

[23] Sumner, Sara. *The Three Systems Engineering Approaches: Top-down, Middle-out, and Bottom-up*. [Online; accessed Januar 31, 2020]. URL: `http://community.vitechcorp.com/index.php/the-three-systems-engineering-approaches-top-down-middle-out-and-bottom-up.aspx`.

[24] Paul Vixie. “crontab(5) - Linux man page”. In: *crontab(5): tables for driving cron - Linux man page* (2007). [Online; accessed Januar 31, 2020]. URL: `https://linux.die.net/man/5/crontab`.

[25] Paul Vixie and Marcela Mašláňová. “cron(8) - Linux man page”. In: *cron(8): daemon to execute scheduled commands - Linux man page* (2007). URL: `https://linux.die.net/man/8/cron`.

[26] Markus Voelter and Sascha Lisson. “Supporting Diverse Notations in MPS’ Projectional Editor.” In: *GEMOC MoDELS*. 2014, pp. 7–16.

[27] Markus Voelter et al. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages.* dslbook.org, 2013, pp. 1–558. ISBN: 978-1-4812-1858-0.

[28] Markus Voelter et al. “Efficient development of consistent projectional editors using grammar cells”. In: *Proceedings of the 2016 ACM SIGPLAN International Conference on Software Language Engineering*. 2016, pp. 28–40.

[29] Markus Voelter et al. “mbeddr: an extensible C-based programming language and IDE for embedded systems”. In: *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*. 2012, pp. 121–140.

[30] Markus Völter. “KernelF-an Embeddable and Extensible Functional Language”. In: 2017.

[31] Martin P Ward. “Language-oriented programming”. In: *Software-Concepts and Tools* 15.4 (1994), pp. 147–161.

A. Appendix

A.1. Creation of the Database Schemes

```
DROP TABLE IF EXISTS QRTZ_FIRED_TRIGGERS;
DROP TABLE IF EXISTS QRTZ_PAUSED_TRIGGER_GRPS;
DROP TABLE IF EXISTS QRTZ_SCHEDULER_STATE;
DROP TABLE IF EXISTS QRTZ_LOCKS;
DROP TABLE IF EXISTS QRTZ_SIMPLE_TRIGGERS;
DROP TABLE IF EXISTS QRTZ_SIMPROP_TRIGGERS;
DROP TABLE IF EXISTS QRTZ_CRON_TRIGGERS;
DROP TABLE IF EXISTS QRTZ_BLOB_TRIGGERS;
DROP TABLE IF EXISTS QRTZ_TRIGGERS;
DROP TABLE IF EXISTS QRTZ_JOB_DETAILS;
DROP TABLE IF EXISTS QRTZ_CALENDARS;


CREATE TABLE QRTZ_JOB_DETAILS
  (
    SCHED_NAME VARCHAR(120) NOT NULL,
    JOB_NAME VARCHAR(200) NOT NULL,
    JOB_GROUP VARCHAR(200) NOT NULL,
    DESCRIPTION VARCHAR(250) NULL,
    JOB_CLASS_NAME VARCHAR(250) NOT NULL,
    IS_DURABLE VARCHAR(1) NOT NULL,
    IS_NONCONCURRENT VARCHAR(1) NOT NULL,
    IS_UPDATE_DATA VARCHAR(1) NOT NULL,
    REQUESTS_RECOVERY VARCHAR(1) NOT NULL,
    JOB_DATA BLOB NULL,
    PRIMARY KEY (SCHED_NAME,JOB_NAME,JOB_GROUP)
);
CREATE TABLE QRTZ_TRIGGERS
```

```
  (
    SCHED_NAME VARCHAR(120) NOT NULL,
    TRIGGER_NAME VARCHAR(200) NOT NULL,
    TRIGGER_GROUP VARCHAR(200) NOT NULL,
    JOB_NAME VARCHAR(200) NOT NULL,
    JOB_GROUP VARCHAR(200) NOT NULL,
    DESCRIPTION VARCHAR(250) NULL,
    NEXT_FIRE_TIME BIGINT(13) NULL,
    PREV_FIRE_TIME BIGINT(13) NULL,
    PRIORITY INTEGER NULL,
    TRIGGER_STATE VARCHAR(16) NOT NULL,
    TRIGGER_TYPE VARCHAR(8) NOT NULL,
    START_TIME BIGINT(13) NOT NULL,
    END_TIME BIGINT(13) NULL,
    CALENDAR_NAME VARCHAR(200) NULL,
    MISFIRE_INSTR SMALLINT(2) NULL,
    JOB_DATA BLOB NULL,
    PRIMARY KEY (SCHED_NAME,TRIGGER_NAME,TRIGGER_GROUP),
    FOREIGN KEY (SCHED_NAME,JOB_NAME,JOB_GROUP)
        REFERENCES QRTZ_JOB_DETAILS(SCHED_NAME,JOB_NAME,
           ↪ JOB_GROUP)
);

CREATE TABLE QRTZ_SIMPLE_TRIGGERS
  (
    SCHED_NAME VARCHAR(120) NOT NULL,
    TRIGGER_NAME VARCHAR(200) NOT NULL,
    TRIGGER_GROUP VARCHAR(200) NOT NULL,
    REPEAT_COUNT BIGINT(7) NOT NULL,
    REPEAT_INTERVAL BIGINT(12) NOT NULL,
    TIMES_TRIGGERED BIGINT(10) NOT NULL,
    PRIMARY KEY (SCHED_NAME,TRIGGER_NAME,TRIGGER_GROUP),
    FOREIGN KEY (SCHED_NAME,TRIGGER_NAME,TRIGGER_GROUP)
        REFERENCES QRTZ_TRIGGERS(SCHED_NAME,TRIGGER_NAME,
           ↪ TRIGGER_GROUP)
);

CREATE TABLE QRTZ_CRON_TRIGGERS
  (
    SCHED_NAME VARCHAR(120) NOT NULL,
    TRIGGER_NAME VARCHAR(200) NOT NULL,
```

```
    TRIGGER_GROUP VARCHAR(200) NOT NULL,
    CRON_EXPRESSION VARCHAR(200) NOT NULL,
    TIME_ZONE_ID VARCHAR(80),
    PRIMARY KEY (SCHED_NAME,TRIGGER_NAME,TRIGGER_GROUP),
    FOREIGN KEY (SCHED_NAME,TRIGGER_NAME,TRIGGER_GROUP)
        REFERENCES QRTZ_TRIGGERS(SCHED_NAME,TRIGGER_NAME,
            ↪ TRIGGER_GROUP)
);

CREATE TABLE QRTZ_SIMPROP_TRIGGERS
  (
    SCHED_NAME VARCHAR(120) NOT NULL,
    TRIGGER_NAME VARCHAR(200) NOT NULL,
    TRIGGER_GROUP VARCHAR(200) NOT NULL,
    STR_PROP_1 VARCHAR(512) NULL,
    STR_PROP_2 VARCHAR(512) NULL,
    STR_PROP_3 VARCHAR(512) NULL,
    INT_PROP_1 INT NULL,
    INT_PROP_2 INT NULL,
    LONG_PROP_1 BIGINT NULL,
    LONG_PROP_2 BIGINT NULL,
    DEC_PROP_1 NUMERIC(13,4) NULL,
    DEC_PROP_2 NUMERIC(13,4) NULL,
    BOOL_PROP_1 VARCHAR(1) NULL,
    BOOL_PROP_2 VARCHAR(1) NULL,
    PRIMARY KEY (SCHED_NAME,TRIGGER_NAME,TRIGGER_GROUP),
    FOREIGN KEY (SCHED_NAME,TRIGGER_NAME,TRIGGER_GROUP)
    REFERENCES QRTZ_TRIGGERS(SCHED_NAME,TRIGGER_NAME,
        ↪ TRIGGER_GROUP)
);

CREATE TABLE QRTZ_BLOB_TRIGGERS
  (
    SCHED_NAME VARCHAR(120) NOT NULL,
    TRIGGER_NAME VARCHAR(200) NOT NULL,
    TRIGGER_GROUP VARCHAR(200) NOT NULL,
    BLOB_DATA BLOB NULL,
    PRIMARY KEY (SCHED_NAME,TRIGGER_NAME,TRIGGER_GROUP),
    FOREIGN KEY (SCHED_NAME,TRIGGER_NAME,TRIGGER_GROUP)
        REFERENCES QRTZ_TRIGGERS(SCHED_NAME,TRIGGER_NAME,
            ↪ TRIGGER_GROUP)
```

```
);

CREATE TABLE QRTZ_CALENDARS
  (
    SCHED_NAME VARCHAR(120) NOT NULL,
    CALENDAR_NAME VARCHAR(200) NOT NULL,
    CALENDAR BLOB NOT NULL,
    PRIMARY KEY (SCHED_NAME,CALENDAR_NAME)
);

CREATE TABLE QRTZ_PAUSED_TRIGGER_GRPS
  (
    SCHED_NAME VARCHAR(120) NOT NULL,
    TRIGGER_GROUP VARCHAR(200) NOT NULL,
    PRIMARY KEY (SCHED_NAME,TRIGGER_GROUP)
);

CREATE TABLE QRTZ_FIRED_TRIGGERS
  (
    SCHED_NAME VARCHAR(120) NOT NULL,
    ENTRY_ID VARCHAR(95) NOT NULL,
    TRIGGER_NAME VARCHAR(200) NOT NULL,
    TRIGGER_GROUP VARCHAR(200) NOT NULL,
    INSTANCE_NAME VARCHAR(200) NOT NULL,
    FIRED_TIME BIGINT(13) NOT NULL,
    SCHED_TIME BIGINT(13) NOT NULL,
    PRIORITY INTEGER NOT NULL,
    STATE VARCHAR(16) NOT NULL,
    JOB_NAME VARCHAR(200) NULL,
    JOB_GROUP VARCHAR(200) NULL,
    IS_NONCONCURRENT VARCHAR(1) NULL,
    REQUESTS_RECOVERY VARCHAR(1) NULL,
    PRIMARY KEY (SCHED_NAME,ENTRY_ID)
);

CREATE TABLE QRTZ_SCHEDULER_STATE
  (
    SCHED_NAME VARCHAR(120) NOT NULL,
    INSTANCE_NAME VARCHAR(200) NOT NULL,
    LAST_CHECKIN_TIME BIGINT(13) NOT NULL,
    CHECKIN_INTERVAL BIGINT(13) NOT NULL,
```

```
    PRIMARY KEY (SCHED_NAME,INSTANCE_NAME)
);

CREATE TABLE QRTZ_LOCKS
  (
    SCHED_NAME VARCHAR(120) NOT NULL,
    LOCK_NAME VARCHAR(40) NOT NULL,
    PRIMARY KEY (SCHED_NAME,LOCK_NAME)
);
```

A.2. Scheduler Server Configuration

```
#================================================
# Configure Main Scheduler Properties
#================================================

org.quartz.scheduler.instanceName: Swino
org.quartz.scheduler.rmi.export: true
org.quartz.scheduler.rmi.registryHost: localhost
org.quartz.scheduler.rmi.registryPort: 1099
org.quartz.scheduler.rmi.createRegistry: true

org.quartz.scheduler.skipUpdateCheck: true

#=================================================
# Configure ThreadPool
#=================================================

org.quartz.threadPool.class: org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount: 10
org.quartz.threadPool.threadPriority: 5

#=================================================
# Configure JobStore
#=================================================

org.quartz.jobStore.misfireThreshold: 60000
```

```
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.
    ↪ JobStoreTX
org.quartz.jobStore.tablePrefix = QRTZ_
org.quartz.dataSource.SQLiteDB.driver = org.sqlite.JDBC
org.quartz.dataSource.SQLiteDB.URL = jdbc:sqlite:./db/cronjobs.
    ↪ db
org.quartz.dataSource.SQLiteDB.maxConnections = 30
org.quartz.jobStore.driverDelegateClass=com.github.fxlex.swino.
    ↪ CustomJDBCDelegate
org.quartz.jobStore.dataSource = SQLiteDB

#==================================================
# Configure Plugins
#==================================================
org.quartz.plugin.shutdownhook.class: org.quartz.plugins.
    ↪ management.ShutdownHookPlugin
org.quartz.plugin.shutdownhook.cleanShutdown: true
```

A.3. Scheduler Client Configuration

```
# Properties file for use by StdSchedulerFactory
# to create a Quartz Scheduler Instance.
#

# Configure Main Scheduler Properties
    ↪ ======================================

org.quartz.scheduler.instanceName: Swino
org.quartz.scheduler.logger: schedLogger
org.quartz.scheduler.skipUpdateCheck: true
org.quartz.scheduler.rmi.proxy: true
org.quartz.scheduler.rmi.registryHost: localhost
org.quartz.scheduler.rmi.registryPort: 1099
```

Eidesstattliche Erklärung

Studierender:	Alexander Pann
Matrikelnummer:	901084

Hiermit erkläre ich, dass ich diese Arbeit selbstständig abgefasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

. . . .
Ort, Abgabedatum   Unterschrift (Vor- und Zuname)

Wilhelm Büchner Hochschule • Private Fernhochschule Darmstadt •