Bachelor's Thesis

SAMT: Compiler and Tools for an Extensible API Modeling Language

Date:	2023-06-16
Term:	Spring 2023
Authors:	Pascal Honegger Marcel Joss Leonard Schütz
Advisor:	Prof. Dr. Olaf Zimmermann
External Co-Examiner:	Dr. Daniel Lübke
Internal Co-Examiner:	Prof. Laurent Metzger
Industry Partner:	**Zürcher Kantonalbank** André Lehner

OST
Eastern Switzerland
University of Applied Sciences

Abstract

Zürcher Kantonalbank maintains various services built on diverse technologies, using proprietary domain-specific languages to model technology-agnostic data and service contracts. This proven approach is implemented using the Xtext framework, which provides the core infrastructure for parsing source code, validating the resulting model, and integrating with the Eclipse IDE. As the maintenance roadmap for Xtext is uncertain, Zürcher Kantonalbank faces a long-term operational risk and is looking for a sustainable solution.

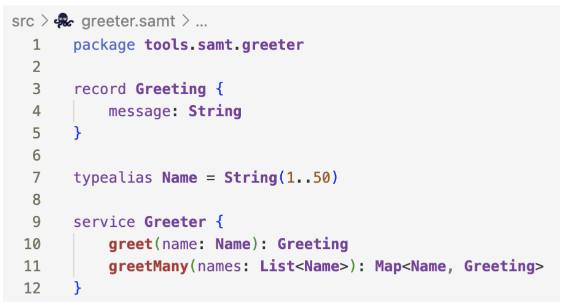
The goal of this project is to design and develop a new open-source domain-specific language called Simple API Modeling Toolkit, or SAMT for short. It retains the technology-agnostic modeling capabilities and employs custom generators that provide support for a specific target language and technology. A Visual Studio Code extension is developed to provide an easy-to-use and modern development experience. The development process includes an initial requirements engineering and language design phase, guided by developers familiar with the existing modeling language.

All critical requirements were fully met, with substantial "should-have" and "could-have" requirements also realized. The project successfully developed the core systems of the new language, including a proof-of-concept code generator for the Kotlin-based Web framework Ktor and the SAMT Visual Studio Code Extension. Usability tests with employees of Zürcher Kantonalbank have yielded positive feedback. Future work includes improving the generator architecture and adding more functionality to the SAMT Visual Studio Code Extension.

Management Summary

Zürcher Kantonalbank runs many services and applications that are based on different technologies. They developed a custom domain-specific language used to model the data and service contracts between the different services. A set of code generators can automatically generate the client and server network interfaces required to communicate and exchange information with each other. These tools have been very helpful in organizing the inherent complexity associated with running a large number of interconnected services. The language depends on the Xtext framework for its core functionality. Unfortunately, the maintenance roadmap for Xtext is uncertain, which raises concerns about the long-term maintainability of the language.

This project aims to develop a successor language, called Simple API Modeling Toolkit (SAMT), that improves upon the existing design by providing a more pleasant developer experience. SAMT retains the ability to model data and services in a technology-agnostic manner but employs custom configurable generators that can provide support for any technology. The new language leaves behind the dependency on the Xtext framework and builds the entire toolchain from scratch. An editor extension developed for Visual Studio Code replaces the existing customized version of the Eclipse IDE.



```
package tools.samt.greeter

record Greeting {
    message: String
}

typealias Name = String(1..50)

service Greeter {
    greet(name: Name): Greeting
    greetMany(names: List<Name>): Map<Name, Greeting>
}
```

Figure 1: Example SAMT file being edited in Visual Studio Code

An example of the new language is shown in Figure 1. SAMT has been implemented in Kotlin, a modern programming language that runs on the Java Virtual Machine. To avoid relying on yet another external framework, the core components of the language processing pipeline, such as the lexer and parser, were built from scratch. Building these components from scratch yielded fine-grained control over diagnostics and error reporting. The editor extension has been published to the official Visual Studio Marketplace as shown in Figure 2, where it is available for anyone to download and use. It can also be ported to other compatible editors with minimal effort, as it is built on top of the Language Server Protocol (LSP).

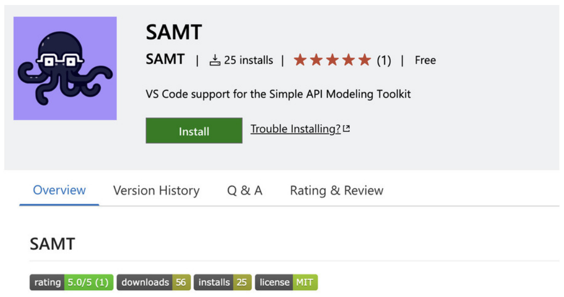


Figure 2: Visual Studio Marketplace page for the SAMT extension

All the critical project requirements have been met, with a considerable number of secondary tasks also completed. A set of example generators were developed, showcasing the generation of Kotlin code with the Ktor framework. Feedback from developers familiar with the old system has been positive. In addition, every feature and system that has been worked on is accompanied by its own comprehensive test suite. Several developer feedback sessions have been conducted, asking developers familiar with the old language to perform certain tasks using our system, with mostly positive results. SAMT was also presented at a developer conference within Zürcher Kantonalbank, where it received positive feedback and interest from the audience.

Due to time constraints, several language features remain unimplemented, such as the ability to model type inheritance. The current generator architecture works well for simple use cases, but is still considered work-in-progress. The editor is still missing context-aware autocompletion and refactoring support. Additionally, some sort of dependency management system should be implemented to allow for the reuse of common code between different projects. In conclusion, a solid foundation has been built for the future of the language. The project is available as open-source software so that anyone can contribute to its further development. Zürcher Kantonalbank can now decide how to integrate SAMT, or parts of it, into their IT landscape.

Acknowledgements

We would like to express our sincere gratitude to Prof. Dr. Olaf Zimmermann for his valuable support and guidance throughout this project. His feedback and advice have been instrumental in shaping our work and pushing us to do our best.

We would also like to thank André Lehner from Zürcher Kantonalbank for his support and insights. His feedback helped us to better understand the needs and expectations of the industry. Furthermore, we would like to thank Zürcher Kantonalbank and its employees for giving us the opportunity to work on this project. The collaborative environment and resources provided by the organization were critical to our success.

Finally, we extend our thanks to our friends and family for their unwavering support and encouragement throughout this project.

Thank you all for your contributions and support throughout this endeavor!

Pascal Honegger, Marcel Joss, Leonard Schütz

Table of Contents

Chapter 1

Introduction

Operating within an ecosystem of diverse technologies, Zürcher Kantonalbank has historically employed a variety of different languages and tools to model data and service contracts. These tools have proven to be difficult to manage, excessively complex and often inaccessible to non-technical personnel. Furthermore, some of these languages support only a single transport protocol, which reduces their applicability. Further complications arise when you consider the need to adapt to new transport technologies such as HTTP, Websockets, gRPC, and Kafka messaging. Every one of these requires a unique modeling language and associated tools. Any migration to a new modeling framework would be inherently costly and time-consuming. Not all projects within the organization would want to upgrade to the new standard immediately, which ultimately requires both the previous and new versions to be maintained concurrently.

Recognizing these challenges, Zürcher Kantonalbank has developed their own proprietary modeling languages which allow them to model their application interfaces in a technology-agnostic manner. Code generation tools output the client and server interfaces for a specific target programming language. Their project relies on Xtext to implement its core functionalities (lexer, parser, semantic validation, and Eclipse IDE integration). Xtext is also used to generate a custom distribution of the Eclipse IDE that provides the development environment for the language. However, the maintenance roadmap for Xtext is uncertain[1], which raises concerns about the long term maintainability of the project. Teams within Zürcher Kantonalbank are content with the value the language provides, but look forward to a more modern tech stack.

The objective of this project is to establish a new open-source domain-specific language that retains the technology-agnostic modeling capabilities of the current language while being more modern, user-friendly, and extendable via custom generators. Additionally, a new extension for Visual Studio Code will be developed that provides a sophisticated and

[1] C. Dietrich, *Call to action: Secure the future maintenance of xtext*, 2020-03. [Online]. Available: https://github.com/eclipse/xtext/issues/1721 (visited on 2023-06-11).

user-friendly development environment. This report details the development process of the new language, starting from an initial requirements engineering and language design phase and later describing the various aspects of our architecture. Several user feedback studies will be conducted, with their results guiding the development.

Chapter 2

Functional Requirements

This chapter documents the functional requirements of the final product.

2.1 Language Features

This section describes the functional requirements related to the type system, constraints, scalability and metadata handling.

2.1.1 Type System

FR-LF-PRIMITIVETYPES	
Description	The following primitive data types are supported: **Int** 32-bit whole number, signed **Long** 64-bit whole number, signed **Float** 32-bit floating point number, signed **Double** 64-bit floating point number, signed **Decimal** Arbitrary precision number, fixed amount of digits before and after decimal point **Boolean** Can be true or false **String** UTF-8 encoded text **Bytes** Arbitrary byte buffer **Date** Timezone-agnostic date value **DateTime** UTC timestamp, millisecond precision **Duration** Time duration, millisecond precision
Priority	Must have

FR-LF-RECORD	
Description	Records define a new data type that contains a list of fields. Records must have a name. Each field in the record has an associated name and type. Records can contain other records, however the overall structure cannot be circular.
Priority	Must have

FR-LF-INHERITANCE	
Description	Records can inherit from another record while keeping all the fields of the parent record. Records can be declared abstract, which prevents them from being used directly, but allows inheriting classes to use their fields.
Priority	Should have

FR-LF-MULTIPLEINHERITANCE	
Description	Records can inherit from multiple different records while keeping all the fields of the parent records. No duplicate field names are allowed in the parent records.
Priority	Could have

FR-LF-ENUM	
Description	Enum types have a name and a set of values. Enums cannot define a type for their values. Values are referenced only via their name. Target language generators are free to represent enums as either integer or string types.
Priority	Must have

FR-LF-CONSTANTS	
Description	Constant declarations can be used to bind a single value to an identifier (primitive values only, e.g. no records are allowed). This identifier can then be used to reference that constant in the future (e.g. inside constraint declarations, config parameters, etc.).
Priority	Could have

FR-LF-FAULT	
Description	Operations can define errors that might occur and their structure.
Priority	Could have

2.1.2 Constraints

FR-LF-NULLABILITY	
Description	Record fields, operation parameters, and return types can be marked as nullable, making the presence of a value optional. Types themselves can also be marked as optional, retaining their nullability status wherever they are used.
Priority	Must have

FR-LF-TYPECONSTRAINTS	
Description	Type constraints restrict the set of possible values that are allowed in a given type. Constraints can be either open- or close-ended. Close-ended constraints must specify two values for the respective minimum and maximum values. Open-ended constraints can omit either the minimum or maximum value. **Range** limit the value of a number between a minimum and maximum value **Size** limit string, list or map to min and max size, where string size is defined as the number of unicode codepoints in a string **Value** limit string, number or boolean to specific value **Pattern** limit string to match regular expression
Priority	Must have

FR-LF-MODULARCONSTRAINTS	
Description	Constraints are modular and can be arbitrarily combined to model any sort of constraint relationship. Different building blocks are provided to model any sort of boolean expression using the original constraint limits. **And** All child constraints must be satisfied **Or** At least one child constraint must be satisfied **Xor** Exactly one of the child constraints must be satisfied **Not** All child constraints must not be satisfied
Priority	Could have

FR-LF-RECORDCONSTRAINTS	
Description	Record constraints are used for constraints spanning across more than one field. They can be defined for a record and reference any fields within the record. In this way, modular constraints can be added to fields conditionally, for example to ensure exactly one of two fields can be optional.
Priority	Could have

FR-LF-LIST	
Description	Lists are ordered containers of values, which can be of any type.
Priority	Must have

FR-LF-MAP	
Description	Maps are unordered containers of key-value pairs. The key can only be an Int, String or Enum. The value can be of any type.
Priority	Could have

FR-LF-ALIAS	
Description	The alias can bind a complex type signature (such as a type with constraints) to a new name. That name can then be used to reference that type. The newly defined name and the original type signature are identical and interchangeable.
Priority	Must have

2.1.3 Services

FR-LF-SERVICE	
Description	Services group a set of operations under a common name. They are protocol and technology agnostic.
Priority	Must have

FR-LF-OPERATION	
Description	Operations are individual actions that can be performed within a service. Each operation has a name, an optional list of parameters and an optional return type. Each parameter has a type and a name. Consumers will wait synchronously for the operation to complete, even if no return type is specified.
Priority	Must have

FR-LF-ONEWAYOPERATION	
Description	Operations can be defined as one way, allowing consumers to skip waiting for the operation to complete. Oneway operations must not have a return type.
Priority	Could have

FR-LF-ASYNCOPERATION	
Description	Operations can be defined as asynchronous. For example, the generated consumer code could allow the caller to specify a callback, which is called once a response is available. An optional timeout value can be configured.
Priority	Could have

FR-LF-PROVIDER	
Description	Providers expose a service via a specific network protocol and technology, for example, a combination of HTTP/Java or SOAP/Python. Consumers of the provider must have the same network protocol, but can have a different technology implementing that protocol. Providers have a name, a service that is being provided and a set of configuration options regarding which network protocol and technology the given service is provided as.
Priority	Must have

FR-LF-CONSUMER	
Description	Consumers use one or multiple services through a provider. They inherit the network protocol specified by the provider but are free to choose their own technology. Consumers have a name and a set of configuration options for the implementing technology.
Priority	Must have

2.1.4 Scalability

FR-LF-PACKAGE	
Description	Packages provide a common namespace for types, services, providers and consumers. Packages have a name. Packages can be included in other packages. Definitions within the same package can be used directly, while definitions from other packages must be referenced by their package name.
Priority	Must have

FR-LF-SEPARATIONOFFILES	
Description	A file can import an arbitrary amount of packages. Importing a package causes all definitions inside that package to become directly visible to the compiler, without having to prefix them with their package name. If there are any name ambiguities, definitions from the current package are preferred. Import statements can also import specific definitions from a package only.
Priority	Must have

FR-LF-DEPENDENCIES	
Description	Packages can declare other packages as dependencies and import them. Types, services and providers declared in the imported package become accessible in the importing package. Packages contain meta information such as name, description, and author. Packages are referenced via a unique url (e.g. GitHub repository) and a version (e.g. hash, git tag, semantic version identifier).
Priority	Could have

2.1.5 Metadata

FR-LF-DOCUMENTATION	
Description	Ability to declare documentation about records, record fields, type aliases, services, operations and providers in the source files. Documentation can come in two varieties, single-line and multi-line. Documentation should be included in the meta-information data structure generated by the parser, allowing it to be used during subsequent processes.
Priority	Should have

FR-LF-ANNOTATION	
Description	Ability to mark certain elements with pre-defined annotations. These annotations can have a wide range of purposes, for example **@Secret** might prevent a value from being logged.
Priority	Should have

2.2 Tooling Requirements

FR-T-PARSERLIB	
Description	A general parser library that can be used to easily parse and access the meta information of a package. The exported data structure contains all information necessary to generate code (e.g. a Java provider) from the parsed SAMT model. It includes syntax validation and semantic consistency checks.
Priority	Must have

FR-T-CODEGENLIB	
Description	A SAMT model contains only technology-agnostic descriptions, which can be used by a generator to output any desired format (e.g. Java source code). A framework or library should exist to facilitate code-generation from a given SAMT model, thus reducing the amount of redundant work that each generator has to do.
Priority	Must have

FR-T-IDESYNTAX	
Description	An IDE plugin that supports syntax highlighting.
Priority	Must have

FR-T-IDEWARNERROR	
Description	The IDE plugin should provide realtime warnings and errors. Both syntax and semantic problems should be detected.
Priority	Should have

FR-T-IDEHINT	
Description	The IDE plugin can autocomplete types, names and other language structures.
Priority	Should have

FR-T-IDEFORMAT	
Description	The IDE plugin can automatically indent code and insert line breaks where appropriate to decrease manual formatting work and increase code readability.
Priority	Should have

FR-T-IDEREFACTOR	
Description	The IDE plugin can automatically refactor code segments, jump to the definition of a type and rename fields in all occurring places.
Priority	Could have

Chapter 3

Non-Functional Requirements

This chapter documents the non-functional requirements of the final product.

3.1 Stakeholder Archetypes

To support the requirements engineering as well as the language and software design, we establish a set of potential stakeholders for SAMT, partly inspired by the real use cases of Zürcher Kantonalbank. The "Authored/Consumed SAMT Constructs" reference the language features introduced in Section 2.1. The following stakeholder archetypes have been defined:

Service Owner	
Description	Developer in charge of a service. Needs to provide SAMT models of services and their implementations to developers of other services. Additionally, code generation may alleviate the need to manually create endpoints and data types in code.
Authored SAMT Constructs	records, faults, services, operations, providers
Consumed SAMT Constructs	-
Estimated Frequency of Use	weekly

System Integrator	
Description	Developer who wants to integrate their service with another service that has SAMT models available. Can use SAMT to generate client code.
Authored SAMT Constructs	consumers
Consumed SAMT Constructs	services, operations, providers, consumers
Estimated Frequency of Use	monthly

Business Domain Expert	
Description	Non-technical person who wants to model their business domain, without diving into technical details
Authored SAMT Constructs	records, services, maybe faults
Consumed SAMT Constructs	-
Estimated Frequency of Use	a few times a year

Enterprise Integration Architect	
Description	Software architect responsible for managing integration of many different projects. Can use SAMT to communicate the structure of services to their teams in a clearly defined and formal way.
Authored SAMT Constructs	records, services, providers, consumers, faults
Consumed SAMT Constructs	-
Estimated Frequency of Use	weekly

SAMT Expert	
Description	Developer with a high level of proficiency in the SAMT DSL and familiarity with the tooling and internals. They may do things such as implementing a custom generators, assisting other developers and providing libraries of common types to their organization.
Authored SAMT Constructs	records, services, providers, consumers, faults
Consumed SAMT Constructs	-
Estimated Frequency of Use	multiple times a week

Depending on the organization structure, the same person may have multiple of these roles. For example, the same developer may act as a Service Owner when working on the service development and as a System Integrator when integrating a service maintained by another team.

3.2 Usability

NFR-U-SERVICEOWNER	
Requirement	SAMT should be easy to use for a service owner
Measure(s)	A professional developer with no prior training in SAMT can define a simple API in 30 minutes with the help of the documentation. A simple API contains • three structs • one service • two operations • one provider
Priority	High

NFR-U-SYSTEMINTEGRATOR	
Requirement	SAMT should be easy to use for a system integrator
Measure(s)	A professional developer with no prior training in SAMT can consume a SAMT service in 15 minutes with the help of the documentation, create a single consumer for an already existing service and provider, and integrate the generated code into their project.
Priority	High

NFR-U-DOMAINEXPERT	
Requirement	SAMT should be easy to use for a domain expert
Measure(s)	A non-technical person can define a simple business API in 30 minutes with the help of the documentation. A simple business API contains • six structs • one service • three operations
Priority	High

3.3 Compatibility

NFR-C-OS	
Requirement	SAMT should work on all common desktop operating systems
Measure(s)	SAMT and its tools runs on Windows 10 and 11, macOS Ventura and Ubuntu 22.04 LTS
Priority	High

3.4 Maintainability

NFR-M-TESTS	
Requirement	Expected behavior of all software deliverables is verified by tests
Measure(s)	Automated tests should achieve line coverage within the following landing zones: 1. Minimum: 70% 2. Target: 80% 3. Outstanding: 90%
Priority	High

NFR-M-QUALITY	
Requirement	Code is maintainable
Measure(s)	A code quality monitoring tool must report no problems when running on the SAMT codebase
Priority	Medium

3.5 Security

NFR-S-VULNERABILITIES	
Requirement	The dependencies of all software deliverables must not have known vulnerabilities
Measure(s)	A vulnerability scanner which checks against vulnerability databases informs about vulnerable dependencies. From the point of discovery, as soon as a fix is available, a fix version will be released within three days.
Priority	High

NFR-S-CODEGEN	
Requirement	The example generator must not generate insecure code
Measure(s)	Code generated by the example generator does not contain any of the vulnerabilities described in the OWASP Top 10[1]
Priority	High

3.6 Performance

NFR-P-COMPILATION	
Requirement	SAMT is performant and finishes tasks within a reasonable time
Measure(s)	Given a benchmark project with the following content: • 30 providers • 30 interfaces • 15 enums and typedefs • 300 structs with 3 fields on average SAMT can parse the model and generate code with the example generator in times defined by the following landing zones: • Minimum: 30 seconds • Target: 10 seconds • Outstanding: 5 seconds Compilation of the generated code is not included in these times.
Priority	High

3.7 Extensibility

NFR-E-EXTENSIBILITY	
Requirement	SAMT is extensible with additional network protocols and technologies
Measure(s)	There exists documentation, which shows how to create a new language/protocol combination e.g. Kotlin/gRPC. Thus, an experienced developer can create a new generator with functionality equivalent to that of the example generator in one work week.
Priority	High

Chapter 4

Deliverables

This chapter lists the deliverables that are expected by the end of the project.

4.1 Source Code

The source code of the SAMT compiler, code generator, and IDE-plugin must be published under an open-source license. Every artifact must be accompanied by a README file describing how to build and use it. The builds of each artifact must be reproducible.

4.2 IDE Plugin

A plugin must be provided to implement support for the SAMT language in a major IDE. This plugin must be easy to install, ideally using the IDE's built-in plugin manager.

4.3 Public Documentation

To make the SAMT language accessible to users, comprehensive user documentation must be provided. This documentation must include the SAMT syntax and features, setup instructions, and sample files demonstrating common usage patterns.

In addition to the user documentation, developer documentation must be provided. This documentation must include an API reference that describes the programmatic interface of SAMT.

4.4 Language Specification

To ensure that the SAMT language is well-defined, a specification of the SAMT language must be provided. This specification must include an EBNF-style grammar which describes the syntax of the SAMT language.

4.5 Project Report

The formal documentation of the project must be provided in the form of a bachelor's thesis (this document). All guidelines and specifications set out by OST must be followed.

Chapter 5

Quality Measures

This chapter documents the steps we have taken to ensure the quality of the final product.

5.1 Continuous Integration

Every commit pushed to the project repository is checked by a CI pipeline. Pull requests can only be merged into the main branch if all checks are successful. The following checks are part of the CI-pipeline:

- Compile whole project
- Run unit tests
- Run linter

5.2 Continuous Delivery

The artifacts of this project are regularly published to industry standard repositories. This release process is automated as far as possible, and at most requires a single click to publish an artifact.

5.3 Code Review

Code is frequently reviewed by other project members, which further improves code quality.

5.4 Working with Git

In order to ensure a clean and consistent history, we adhere to the following rules when working with git:

- We use conventional commits[1]
- We use pull requests in GitHub, direct pushes to main are forbidden
- We rebase our commits locally and create a merge commit when merging into main

5.5 Definition of Done (DoD)

The following listings define the criteria that must be met for a task to be considered done.

Feature (Defect / Story)

- Acceptance criteria / issue of user story is met
- Code follows best practices and guidelines
- No TODOs without an issue number are allowed
- Project builds without errors or warnings
- Tests are written and all tests are passing
- Peer code review performed
- Documentation updated
- Associated with epic and release
- Hours worked on are logged correctly

Sprint

- DoD of each item included in the sprint is met
- Product backlog updated
- Sprint has a defined goal

Release

- Release is documented
- Code complete
- Environments are prepared for release
- QA is done and all issues resolved
- Manual tests pass

[1]https://www.conventionalcommits.org/en/v1.0.0/

- Check that no unfinished work has been left in any development or staging environment.

5.6 Definition of Ready (DoR)

The following lists define the criteria that must be met for a task to be considered ready to be prioritized in the backlog.

Defect

- Steps to reproduce
- Severity
- Screenshots

Story

- Use-Case (e.g. configure pretty-print for json transport)
- Estimate
- Epic link
- Acceptance criteria list specified
- Should be doable within a week, otherwise try to split it

5.7 Test Concept

This section describes the methodology and tools used to validate the correctness of the software.

Unit Tests

- Business Logic will be unit tested
- Run on every push to GitHub, merging is blocked until failing tests are fixed
- Automated

Integration Tests

- Framework / technology dependent code will be integration tested
- Mainly targets IDE support and interactions with different terminals and operating systems
- Manual

Usability Tests

- See Appendix C
- Project will be given to multiple people for testing (hallway testing)
- NFR-U-SERVICEOWNER: SAMT should be friendly to use for a service owner
- NFR-U-SYSTEMINTEGRATOR: SAMT should be friendly to use for a service consumer
- NFR-U-DOMAINEXPERT: SAMT should be friendly to use for a domain expert
- Manual

5.8 Code Quality Tools

We have chosen to integrate a set of code quality tools to ensure we meet the requirements from Section 3.4 mentioned below:

- NFR-M-TESTS: Desired behavior of all software deliverables is verified by tests
- NFR-M-QUALITY: Code is maintainable
- Automated

5.8.1 Qodana

To ensure the integrity of our code we have integrated Qodana into our CI pipeline. Qodana's community edition can be hosted in the cloud and is free for open-source projects. Other reasons are the easy integration into our CI pipeline, compatibility with IntelliJ and the fact that Qodana works well with all our other used technologies. As a result, code smells, potential bugs, dead code and general potential improvements in the overall code structure are reported.

5.8.2 Kover

To be able to reach our goal of 80% test-coverage, Kover[2] was integrated. Kover is a Gradle plugin for native Kotlin code coverage, and provides an easy setup. It is important to note that Kover is still in its incubator phase, however multiple reviews have shown promising results, which is why we decided it was fitting for our Kotlin setup. We have integrated a job in our CI pipeline that fails if the code coverage does not reach our specified minimum and provides a report that shows where the coverage needs to be improved.

[2]`https://github.com/Kotlin/kotlinx-kover`

Chapter 6

Language Design

This chapter documents the design phase for the SAMT language and the software architecture of the final SAMT toolkit.

6.1 Comparing Against Existing Solutions

The goal of this section is to highlight the strengths and weaknesses of existing modeling solutions, both from a conceptual and technical aspect.

To showcase the grammar of each solution, a simple example API is modeled that contains a single operation. The `greet` operation accepts a `GreetRequest` and responds with a `GreetResponse`. Due to conceptual differences between the solutions, the models are not 100% identical.

6.1.1 OpenAPI

OpenAPI[1] is a specification for building, documenting, and consuming HTTP APIs.[2] It allows developers to describe the functionality of their APIs in a standardized way, which makes it easier for other developers to understand and use their APIs. OpenAPI's biggest advantage is its standardization and widespread adoption. It is an open-source specification that has been developed and maintained by a large community of contributors, which has led to its broad acceptance across industries and organizations.

However, one major drawback of OpenAPI is the dependency on HTTP APIs. Other protocols like gRPC and SOAP cannot be modeled using OpenAPI. Lastly, the fact that it relies heavily on YAML leads to a vast amount of characters required compared to tailor-made DSLs, as shown in Listing 6.1.

[1] `https://www.openapis.org/`

[2] D. Miller *et al.*, *OpenAPI Specification v3.1.0*, 2022-02. [Online]. Available: `https://spec.openapis.org/oas/v3.1.0` (visited on 2023-06-14).

greeter.yml

```yaml
openapi: 3.0.0
info:
  title: Greeter OpenAPI
  version: 1.0.0
paths:
  /greet:
    post:
      summary: Greet the Server
      description: Greet the Server, which greets you back
      operationId: greetId
      requestBody:
        description: The caller
        content:
          application/json:
            schema:
              $ref: '#/components/schemas/GreetRequest'
        required: true
      responses:
        '200':
          description: The greeting response from the server
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/GreetResponse'
components:
  schemas:
    GreetRequest:
      type: object
      properties:
        name:
          type: string
    GreetResponse:
      type: object
      properties:
        message:
          type: string
```

Listing 6.1: OpenAPI greeter model

6.1.2 Microservice DSL

The Microservice DSL[3] (MDSL) is a Domain-Specific Language to specify (micro-)service contracts, data representations and API endpoints.[4] Instead of only focusing on API modeling in isolation, MDSL also incorporates design principles and API patterns. An example of the Microservice DSL is shown in Listing 6.2.

greeter.mdsl

```
API description GreeterMDSL

data type GreetRequest (
    "name":D<string>
)

data type GreetResponse (
    "message":D<string>
)

endpoint type GreetEndpoint
exposes
  operation greet
    expecting payload "in": GreetRequest
    delivering payload GreetResponse

API provider GreetProvider
  offers GreetEndpoint
  at endpoint location "https://localhost:8080"
  via protocol HTTP
    binding resource GreetResource at "/greet"
      operation greet to POST

API client GreetConsumer
  consumes GreetEndpoint
  from GreetProvider
  via protocol HTTP
```

Listing 6.2: MDSL greeter model

[3] https://microservice-api-patterns.github.io/MDSL-Specification/

[4] O. Zimmermann *et al.*, *Patterns for API design* (Addison-Wesley Signature Series (Vernon)). Boston, MA: Addison Wesley, 2023-01, Appendix C.

6.1.3 Existing Xtext-Based Solution

The existing modeling languages, which SAMT is supposed to replace, was developed around 2012 using the Xtext framework. At the time, Xtext was a very prominent and widely used framework for developing textual DSLs, which made it a good choice for the project. However, the maintenance roadmap for Xtext is uncertain[5], which poses a risk for the long-term maintainability of the existing solution. An example of the existing Xtext-based solution is shown in Listings 6.3 to 6.6.

greeter.sstdsl

```
package greeter

interface Greeter {
  GreetResponse greet(request: GreetRequest)
}

struct GreetRequest {
  name: string
}

struct GreetResponse {
  message: string
}
```

Listing 6.3: Existing Xtext-based greeter core model

greeter-provider.srvdsl

```
package greeter

provider GreeterRest {
  implements: Greeter
  transport: REST {
    rest-mappings: GreeterMapping
  }
  protocol: JSON

  target-platform: JAVA
}
```

Listing 6.4: Existing Xtext-based greeter provider

[5]Dietrich, *Call To Action: Secure the future maintenance of Xtext*.

greeter-consumer.srvdsl

```
package greeter

consumer GreeterRestConsumer {
  consumes: {
    from GreeterRest: greet
  }

  target-platform: JAVA
}
```

Listing 6.5: Existing Xtext-based greeter consumer

greeter.srvdsl

```
package greeter

mapping GreeterMapping {
  path: ""
  operation: greet {
    http-method: POST
    path: "/greet"
  }
}
```

Listing 6.6: Existing Xtext-based greeter rest mapping

6.1.4 Other Solutions

There exists a plethora of other API modeling solutions, each with their own strengths and weaknesses. The following modeling and programming languages provide a broad overview of ways to model APIs.

WSDL

Web Services Description Language (WSDL) is a platform, programming language, and protocol independent description language for web services.[6] The main problem with WSDL is its verbose XML notation. An example of WSDL is shown in Listing 6.7.

[6] R. Chinnici *et al.*, "Web services description language (wsdl) version 2.0 part 1: Core language," *W3C recommendation*, vol. 26, no. 1, p. 19, 2007.

greeter.xml

```xml
<?xml version="1.0" encoding ="utf-8"?>
<definitions name="Greet the Server"
  targetNamespace="greeter.wsdl"
  xmlns:tns="greeter.wsdl"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://schemas.xmlsoap.org/wsdl/">
  <!-- definition of datatypes -->
  <types>
  <schema targetNamespace="greeter.xsd"
    xmlns="http://www.w3.org/2000/10/XMLSchema">
    <element name="GreetRequest">
    <complexType><all>
      <element name="name" type="string"/></all></
         complexType>
    </element>
    <element name="GreetResponse">
    <complexType><all>
      <element name="message" type="string"/></all></
         complexType>
    </element>
  </schema>
  </types>
  <!-- request messages -->
  <message name="GreetRequestMessage">
  <part name="request" type="xsd:GreetRequest"/>
  </message>
  <!-- response messages -->
  <message name="GreetResponseMessage">
  <part name="response" type="xsd:GreetResponse"/>
  </message>
  <!-- server's services -->
  <portType name="Greeter">
    <operation name="greet">
    <input message="tns:GreetRequestMessage"/>
    <output message="tns:GreetResponseMessage"/>
    </operation>
  </portType>
  <!-- server encoding -->
  <binding name="Greeter_webservices" type="tns:Greeter">
  <soap:binding style="rpc" transport="http://schemas.
```

```
     xmlsoap.org/soap/http"/>
    <operation name="greet">
    <soap:operation soapAction="urn:xmethods-delayed-quotes#
      greet"/>
    <input><soap:body use="encoded" namespace="urn:xmethods-
      delayed-quotes"
     encodingStyle="http://schemas.xmlsoap.org/soap/
        encoding/"/></input>
    <output><soap:body use="encoded" namespace="urn:xmethods
      -delayed-quotes"
     encodingStyle="http://schemas.xmlsoap.org/soap/
        encoding/"/></output>
    </operation>
  </binding>
  <!-- access to service provider -->
  <service name="GreeterProvider">
  <port name="GreeterProvider_Port" binding="
     Greeter_webservices">
  <soap:address location="https://localhost:8080/greet"/>
  </port>
  </service>
</definitions>
```
Listing 6.7: WSDL greeter

Protocol Buffers

Protocol Buffers[7] are a language-neutral, platform-neutral extensible mechanism for serializing structured data.[8] It is commonly used in conjunction with gRPC[9], a high-performance RPC framework. This combination is not transport agnostic, but still represents a modern and relatively simple way of modeling an API, as shown in Listing 6.8.

greeter.proto

```
syntax = "proto3";

package greeter;

service Greeter {
  rpc Greet(GreetRequest) returns (GreetResponse);
}

message GreetRequest {
  string name = 1;
}

message GreetResponse {
  string message = 1;
}
```

Listing 6.8: gRPC greeter model

[7] https://protobuf.dev/

[8] K. Varda, "Protocol Buffers: Google's Data Interchange Format," 2008-07. [Online]. Available: https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html (visited on 2023-06-14).

[9] https://grpc.io/

Jolie

Jolie[10] is a service-oriented programming language.[11] Although the language does not focus on API modeling, it does provide some nice features that are relevant to the topic. An example of Jolie is shown in Listing 6.9.

greeter.ol

```
type GreetRequest {
  name: string
}

type GreetResponse {
  message: string
}

interface GreeterInterface {
requestResponse:
  greet( GreetRequest )( GreetResponse )
}

service GreeterService {
  execution: concurrent

  inputPort GreeterInput {
    location: "https://localhost:8080"
    protocol: http { format = "json" }
    interfaces: GreeterInterface
  }
}
```

Listing 6.9: Jolie greeter model

[10]https://www.jolie-lang.org/

[11]F. Montesi *et al.*, "Composing services with JOLIE," in *Fifth European Conference on Web Services (ECOWS'07)*, IEEE, 2007, pp. 13–22.

AsyncAPI

AsyncAPI[12] is a specification for building, documenting, and consuming asynchronous APIs.[13] It allows developers to describe the functionality of their asynchronous APIs in a standardized way. Different transport technologies are supported, including AMQP, HTTP, JMS, Kafka and MQTT. A simple example of an AsyncAPI specification is shown in Listing 6.10.

greeter.yml

```yaml
asyncapi: 2.0.0
info:
  title: Greeter AsyncAPI
  version: 1.0.0
channels:
  greet:
    publish:
      operationId: greetRequest
      message:
        $ref: '#/components/messages/GreetRequest'
    subscribe:
      operationId: greetResponse
      message:
        $ref: '#/components/messages/GreetResponse'
components:
  messages:
    GreetRequest:
      payload:
        type: object
        properties:
          name:
            type: string
    GreetResponse:
      payload:
        type: object
        properties:
          message:
            type: string
```

Listing 6.10: AsyncAPI greeter model

[12] https://www.asyncapi.com/

[13] *AsyncAPI Specification*, AsyncAPI Initiative, 2023-02. [Online]. Available: https://www.asyncapi.com/docs/reference/specification/v2.6.0 (visited on 2023-06-14).

6.1.5 Capability Matrix

Table 6.1 highlights the main conceptual differences between a chosen subset of API modeling solutions, focusing on the features shared by SAMT.

	OpenAPI	**MDSL**	**Existing Xtext DSL**	**Feature-Complete SAMT**
Model Syntax	YAML and JSON	Custom Grammar		
IDE Support	Very broad	Eclipse Plugin Web-Editor	Eclipse-Based IDE	Visual Studio Code Plugin
Generator	Extensible	Part of Core Package		Extensible
Programming Language	If Supported by Generator	If Supported by Core Package		If Supported by Generator
Transport Protocol	Only HTTP	If Supported by Core Package & Generator		
Type Constraints	Yes	No	Yes	
Intra-Model References	Via Published Schema	No	Via Published Model	
Enforced File Separation	No		Enforced	Compiler Warning

Table 6.1: Comparison of modeling language concepts

6.2 Language Syntax

Since our language is primarily intended to replace and extend the implementation currently used by Zürcher Kantonalbank, the new syntax should not deviate too much from the original. While it is our goal to simplify and modernize the existing grammar, we aim to keep it as close to the original as possible.

6.2.1 EBNF Grammar

Figure 6.1 shows the grammar of the SAMT language in EBNF syntax. The grammar does not include the syntax for model comments, as they are filtered out in an early step in the compiler and thus allowed everywhere. Likewise, the following grammar nodes are also omitted from the listing, as describing them with a sentence is more concise than using EBNF syntax:

- ⟨Letter⟩: Upper and lowercase latin letters as well as the underscore character.
- ⟨Number⟩: Either an ⟨Integer⟩ (whole number) or a ⟨Float⟩ (number with decimal point), includes the negative sign for negative numbers.
- ⟨String⟩: A sequence of UTF-8 code-points or escaped special characters (e.g. \n for a newline character), enclosed in double quotes.

```
                         ⟨File⟩ → { TopLevelStatement }
             ⟨TopLevelStatement⟩ → ⟨ImportStatement⟩ | ⟨PackageDeclaration⟩ | ⟨Declaration⟩
               ⟨ImportStatement⟩ → “import” ⟨ImportBundleIdentifier⟩ [“as” ⟨Identifier⟩]
            ⟨PackageDeclaration⟩ → “package” ⟨BundleIdentifier⟩
                   ⟨Declaration⟩ → ⟨RecordDeclaration⟩ | ⟨EnumDeclaration⟩ | ⟨TypeAliasDeclaration⟩
                                   | ⟨ServiceDeclaration⟩ | ⟨ProviderDeclaration⟩ | ⟨ConsumerDeclaration⟩
             ⟨RecordDeclaration⟩ → {⟨Annotation⟩} “record” ⟨Identifier⟩ [“extends” ⟨BundleIdentifier⟩
          {“,” ⟨BundleIdentifier⟩}] [“{” { RecordField } “}”]
                   ⟨RecordField⟩ → {⟨Annotation⟩} ⟨Identifier⟩ “:” ⟨Expression⟩
               ⟨EnumDeclaration⟩ → {⟨Annotation⟩} “enum” ⟨Identifier⟩ “{” [⟨IdentifierList⟩] “}”
          ⟨TypeAliasDeclaration⟩ → {⟨Annotation⟩} “typealias” ⟨Identifier⟩ “=” ⟨Expression⟩
            ⟨ServiceDeclaration⟩ → {⟨Annotation⟩} “service” ⟨Identifier⟩ “{” { ⟨OperationDeclaration⟩
          | ⟨OnewayOperationDeclaration⟩ } “}”
          ⟨OperationDeclaration⟩ → {⟨Annotation⟩} [ “async” ] ⟨Identifier⟩ “(” ⟨OperationParameterList⟩ “)”
          [“:” ⟨Expression⟩ ] [“raises” ⟨ExpressionList⟩]
    ⟨OnewayOperationDeclaration⟩ → {⟨Annotation⟩} “oneway” ⟨Identifier⟩ “(” ⟨OperationParameterList⟩ “)”
        ⟨OperationParameterList⟩ → [⟨OperationParameter⟩ {“,” ⟨OperationParameter⟩ }]
            ⟨OperationParameter⟩ → {⟨Annotation⟩} ⟨Identifier⟩ “:” ⟨Expression⟩
           ⟨ProviderDeclaration⟩ → “provide” ⟨Identifier⟩ “{” { ⟨ProviderDeclarationStatement⟩ } “}”
  ⟨ProviderDeclarationStatement⟩ → ⟨ProviderImplementsStatement⟩ | ⟨ProviderTransportStatement⟩
   ⟨ProviderImplementsStatement⟩ → “implements” ⟨BundleIdentifier⟩ [“{” [⟨IdentifierList⟩] “}”]
    ⟨ProviderTransportStatement⟩ → “transport” ⟨Identifier⟩ [⟨Object⟩]
           ⟨ConsumerDeclaration⟩ → “consume” ⟨BundleIdentifier⟩ “{” {⟨ConsumerUsesStatement⟩} “}”
         ⟨ConsumerUsesStatement⟩ → “uses” ⟨BundleIdentifier⟩ [“{” [⟨IdentifierList⟩] “}”]
                    ⟨Annotation⟩ → “@” ⟨Identifier⟩[“(” [⟨ExpressionList⟩] “)”]
                    ⟨Expression⟩ → ⟨BundleIdentifier⟩ | ⟨Number⟩ | ⟨Boolean⟩ | ⟨String⟩ | ⟨Range⟩
                                   | ⟨Object⟩ | ⟨Array⟩ | ⟨CallExpression⟩ |
                                   | ⟨GenericSpecialization⟩ | ⟨OptionalPostOperator⟩ | ⟨Wildcard⟩
                                   | ( “(” ⟨Expression⟩ “)” )
                ⟨CallExpression⟩ → ⟨Expression⟩ “(” [⟨ExpressionList⟩] “)”
         ⟨GenericSpecialization⟩ → ⟨Expression⟩ “⟨” ⟨ExpressionList⟩ “⟩”
          ⟨OptionalPostOperator⟩ → ⟨Expression⟩ “?”
                         ⟨Range⟩ → ⟨Expression⟩ “..” ⟨Expression⟩
                        ⟨Object⟩ → “{” [ObjectFieldDeclaration {“,” ObjectFieldDeclaration } ] “}”
        ⟨ObjectFieldDeclaration⟩ → ⟨Identifier⟩ “:” ⟨Expression⟩
                         ⟨Array⟩ → “[” [ ⟨ExpressionList⟩ ] “]”
                ⟨ExpressionList⟩ → ⟨Expression⟩ {“,” ⟨Expression⟩ }
                ⟨IdentifierList⟩ → ⟨Identifier⟩ {“,” ⟨Identifier⟩ }
                      ⟨Wildcard⟩ → “*”
                    ⟨Identifier⟩ → [“^”] ⟨Letter⟩ { ⟨Letter⟩ | ⟨Digit⟩ }
              ⟨BundleIdentifier⟩ → ⟨Identifier⟩ { “.” ⟨Identifier⟩ }
        ⟨ImportBundleIdentifier⟩ → ⟨Identifier⟩ { “.” ⟨Identifier⟩ } [“.” “*”]
                       ⟨Boolean⟩ → “true” | “false”
```

Figure 6.1: EBNF grammar of the SAMT language

6.2.2 Language Semantics

In addition to being syntactically valid, some semantic conditions have to be met in order for a SAMT model to be considered valid:

Unique names Each name in a package, namely for records, enums, type aliases, services and providers, must be unique. Multiple fields in the same record or values of an enum must not have the same name. Each operation of a service and each parameter of an operation must have a unique name.

No cycles Records must not have cyclic field types. For example, if a record `A` has a field of type `B` and `B` has a field of type `A`, that is an error. However, a record may have a list of its own type, as that recursion can be terminated by an empty list at runtime and is a common use case for hierarchical structures. Similarly, type aliases must not be cyclic either.

Types Record fields, operation parameters, and return types must be a valid data type, meaning no services or provider types. Each implements-statement in a provider must refer to a service. If an implements-statement specifies operations, they must actually exist in the service. Consumers can only consume a provider, and each uses-statement must refer to a service. If a uses-statement specifies operations, they must actually exist in the service and be implemented by the provider.

Constraints Type constraints can only be applied in the ways described in the requirement FR-LF-TYPECONSTRAINTS in Section 2.1.1.

Ranges The start of a range must be smaller than the end of a range and both ends of the range cannot be a wildcard.

Annotation `@Description` and `@Deprecated` annotations are supported and can only be applied once to each element. `@Description` annotations must have a string as its only argument. `@Deprecated` annotations can have an optional deprecation message as their argument.

6.2.3 Notable Design Decisions

Extensible Transport Configuration

A key aspect of SAMT is its extensibility by allowing users to write their own generators. However, in order to extend SAMT in a corporate setting, the underlying transport protocol must be extensible as well. An initial proposal to allow generators to define their own transport configuration has been rejected as it is impossible to guarantee interoperability. If every generator defines its own transport, one can imagine a scenario where a Java generator encodes a boolean as either “1” or “0” while the corresponding python generator uses “true” or “false”.

After more careful consideration, it was decided that both generators and transport protocols can be extended through plugins. These plugins can be written in any JVM-compatible language and are loaded at runtime. This allows for a standardized transport configuration within SAMT, as well as custom transport configurations by the community or a corporation. As a result, code generators can reference transport configurations at runtime to ensure that two generators, which rely on the same transport, are compatible with each other.

Due to time constraints, this architecture was not fully implemented. Instead, the transport configuration and code generators are encapsulated in their own modules with the plugin system not yet implemented. Although extending SAMT currently requires a fork of the project, we are confident that this feature can be implemented in the future.

Dedicated Generator Configuration

The existing Xtext-based DSL used by Zürcher Kantonalbank contains a target configuration for the provider and consumer that specifies programming language-specific things, such as package names in Java. An example of this feature is shown in Listing 6.11.

```
provide GreetProvider {
    implements GreetService

    target java {
        sourcePackage: "com.enterprise.example"
    }
}
```

Listing 6.11: Example target declaration in existing language

This violates the separation of concerns between the transport modeling and generator invocation. Hence, it was decided to remove generator-specific entries from the DSL and instead configure such entries in a more appropriate dedicated generator configuration.

Short-form Constraints

In the grammar, constraints are defined as a `CallExpression` with a name and an argument. This was done to make the language more expressive, especially when using multiple constraints on the same type. To keep the language from becoming too verbose, we decided to allow omitting the name of the constraint as long as it is unambiguous. For example, `String(size(1..100))` is equivalent to `String(1..100)`.

6.2.4 Full Example Model

Listings 6.12 to 6.14 model an example greeter API which is similar to the ones in Section 6.1. Additional constraints and types were created to showcase more language features. This simple model does not showcase all possible language features, but includes additional constraints and types to demonstrate the essential features of the language.

greeter.samt

```
package samt.greeter

// define type alias for names
typealias Name = String( size(20..100) )

record GreetRequest {
  name: Name
}

record GreetResponse {
  message: String( pattern("[a-z]*") )

  @Description("
    This message should be destroyed once this date time is
       reached.
    If it is null, this response can live on forever.
  ")
  validUntil: DateTime?
}

service GreetService {
  greet(request: GreetRequest): GreetResponse
}
```

Listing 6.12: SAMT greeter core model

greeter-http.samt

```
package samt.greeter

provide GreeterHttpEndpoint {
  // The "{ greet }" is optional if all operations are
     implemented
  implements GreetService { greet }

  transport HTTP {
    serialization: "JSON"
    operations: {
      GreetService: {
        // More complex patterns and configuration is
           possible
        greet: "POST /greet"
      }
    }
  }
}
```

Listing 6.13: SAMT greeter provider model

greeter-consumer.samt

```
import samt.greeter.*

package samt.foo

consume GreeterHttpEndpoint {
  // The "{ greet }" is optional if all operations are used
  uses GreetService { greet }
}
```

Listing 6.14: SAMT greeter consumer model

Chapter 7

Architecture

This chapter documents the software architecture of the final product. The C4 model[1] is used to visualize the architecture.

7.1 System Context

SAMT has different stakeholders, potentially across different organizational boundaries (for a full list see Section 3.1). Figure 7.1 shows how different teams might interact with SAMT in the future, placing an emphasis on the way different teams have dependencies on each other. The service owner and business domain expert both use SAMT to model their service provider, for example, a central API for fetching account details. On the other end, team B wants to consume those services and uses SAMT to specify which parts of the API they are interested in. Both teams also use SAMT to generate code in the target programming language they like, for example, the provider code for Team A could be a Java HTTP server while the consumer code for Team B could be a Python HTTP client.

[1] `https://c4model.com`

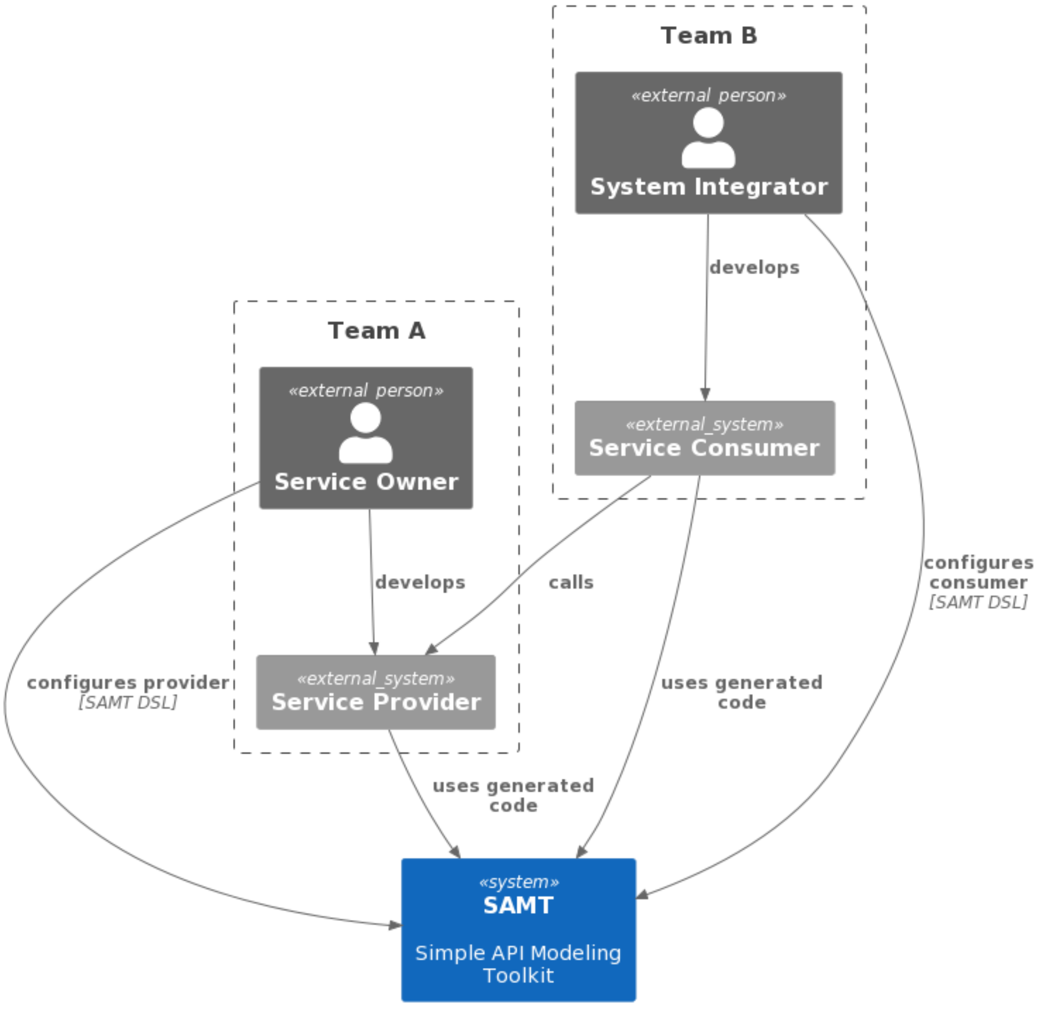


Figure 7.1: C4 system context diagram

7.2 Evaluation of Parser Variants

When implementing a parser one has to decide between using a parser generator or writing the parser by hand.

7.2.1 ANTLR

ANTLR[2] is a widely used parser generator.[3] To use ANTLR, one has to specify the desired grammar in ANTLR's grammar format. Subsequently, ANTLR emits the code for a lexer and parser that can recognize the desired grammar. Additionally, ANTLR generates classes that represent the AST class hierarchy along with traversal helper classes. Such code is often repetitive, which is why a parser generator can save time. The generated parser code represents the underlying state machine and can be difficult to read compared to a carefully handwritten parser. Furthermore, generated classes still require ANTLR's runtime library to function, which would run counter to our stated goal of minimal dependencies. Lastly, ANTLR's parse trees are a one-to-one representation of the grammar, whereas an abstract syntax tree (AST) can be easier to work with in the semantic steps. Thus, when using ANTLR, it may still be desirable to convert the parse tree to an abstract syntax tree.

7.2.2 Handwritten Recursive Descent Parser

Writing a parser by hand requires additional effort compared to using a generator, but gives full control over the parser's behavior. Handwritten code is often more readable than generated code, though one could argue that the readability of generated code is less important as it can be treated as a sort of black box. Recursive descent parsing refers to the implementation of a procedure for each non-terminal symbol in the grammar, which returns the corresponding node type in the syntax tree. This approach results in a code structure that mirrors the language's formal grammar, which makes it easy to understand for anyone familiar with the grammar. For example, if there is a rule called `IfStatement` in the grammar of a programming language, such as `IfStatement = "if", Expression, "then", Statement`, then there will be a method called `parseIfStatement` in the parser which returns an object of the type `IfStatementNode`. Listing 7.1 shows an example of what the parse method for this `IfStatement` might look like.

[2] https://www.antlr.org

[3] T. J. Parr and R. W. Quong, "ANTLR: A predicated-LL (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

```kotlin
abstract class StatementNode
class IfStatementNode(
    val condition: ExpressionNode,
    val statement: StatementNode
) : StatementNode
...

class Parser {
    ...
    fun parseIfStatement(): IfStatementNode {
        consume<IfToken>()
        val condition: ExpressionNode = parseExpression()
        consume<ThenToken>()
        val statement: StatementNode = parseStatement()
        return IfStatementNode(condition, statement)
    }
    ...
}
```
Listing 7.1: Excerpt of a recursive descent parser in Kotlin

Error handling can be fully customized to provide a helping hand to users facing syntax errors. Widely used compilers such as javac, as well as the C and C++ frontends in the GNU Compiler Collection (GCC) have handwritten recursive descent parsers.[4] GCC switched from generated parsers to handwritten ones – reported benefits were better parsing infrastructure, improved handling of extensions and a cleaner separation between parsing and semantic analysis.

[4]G. Li, *The comment of the class JavacParser is not appropriate*, 2021-06. [Online]. Available: https://github.com/openjdk/jdk/commit/b98e52a49191cfbb7d954646cd80a6711daeaca6 (visited on 2023-06-14); J. Myers, *New_C_Parser*, 2008-01. [Online]. Available: https://gcc.gnu.org/wiki/New_C_Parser (visited on 2023-06-14).

7.2.3 Decision

We decided to implement a handwritten recursive descent parser instead of relying on a parser generator based on the following criteria.

Maintainability While designing and building a handwritten parser introduces additional up-front development and design effort, maintaining and extending the handwritten parser becomes much easier than working with a parser generator. Although there are no formal scientific studies to support this claim, we have found enough anecdotal evidence on the Internet to justify this opinion.

Performance Older parser generators tend to produce slow performing code. One of our key NFRs is an improved performance over the existing Xtext based implementation. While modern parser generators like ANTLR easily outperform Xtext, a finely tuned handwritten parser will be even faster.

Independence Using a parser generator would permanently add another dependency to the project, and new engineers working on the project would have to familiarize themselves with the chosen library. By writing a custom parser, engineers only need to have a basic understanding of how a recursive descent parser works.

Experience Each team member has experience in compiler construction and has built a recursive descent parser by themselves at least once. We wanted to make good use of this knowledge and apply it to this project. We believe that if one cannot build something, one does not fully understand it, and we strive to fully understand every step of the SAMT code generation pipeline.

7.3 IDE Support

7.3.1 Requirements

The functional requirements for IDE support, shown in Section 2.2, are the following:

- FR-T-IDESYNTAX: Syntax Highlighting
- FR-T-IDEWARNERROR: Real time warnings and errors
- FR-T-IDEHINT: Code completion
- FR-T-IDEFORMAT: Code formatting
- FR-T-IDEREFACTOR: Code refactoring

7.3.2 Evaluation of Target IDE

IntelliJ and Visual Studio Code were considered as possible targets for an IDE plugin. Frameworks like JetBrains MPS[5] and Eclipse Xtext[6] also allow IDE integration, but

[5]https://www.jetbrains.com/mps/
[6]https://www.eclipse.org/Xtext/

would tie the entire language to a specific ecosystem, which is a problem of Zürcher Kantonalbank's current solution that we are trying to avoid.

IntelliJ

IntelliJ is the most widely used IDE at Zürcher Kantonalbank, which makes it an attractive target for IDE support, but it is also resource intensive and slow to start. It has a plugin API which supports the following features, among others:[7]

- Syntax highlighting
- Real time warnings and errors
- Code completion
- Code formatting
- Code refactoring

These features would suffice to fulfill our functional requirements for tooling.

The APIs require implementing IntelliJ's lexer and parser formats and work done on an IntelliJ plugin may not be easily transferred to other tools. The plugin APIs occasionally have breaking changes which may be a concern for the stability of the tooling.[8]

Visual Studio Code

Visual Studio Code is the most widely used editor according to the Stack Overflow survey[9] and has a plugin API. Additionally, it supports LSP which could be leveraged by SAMT to support VS Code and other editors with LSP support.

LSP was originally developed for Visual Studio Code, but many other tools support the protocol as well or have plugins which enable them to. It supports the following features among others:[10]

- Real time warnings and errors
- Code completion

[7] *Intellij Platform Plugin SDK - Custom Language Support*, JetBrains s.r.o, 2023. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/custom-language-support.html (visited on 2023-03-03).

[8] *Incompatible Changes in Intellij Platform and Plugins API*, JetBrains s.r.o, 2023. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/custom-language-support.html (visited on 2023-03-03).

[9] *Stack Overflow Developer Survey 2022*, Stack Overflow, 2022. [Online]. Available: https://survey.stackoverflow.co/2022/#most-popular-technologies-new-collab-tools-prof (visited on 2023-03-02).

[10] *Language Server Protocol Specification - 3.17*, Microsoft, 2022-05. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/ (visited on 2023-03-03).

- Code formatting
- Code refactoring

Syntax highlighting is not a part of the LSP, but Visual Studio Code supports syntax highlighting with TextMate grammars.[11] A combination of a language server and a TextMate grammar in a Visual Studio Code plugin could thus fulfill our functional requirements concerning IDE support.

Using a language server reduces the API surface to Visual Studio Code itself and the server could be reused to build plugins for other editors. One such candidate would be JetBrains Fleet[12], which is currently in public preview and does support LSP, although this is out of scope for this project.

7.3.3 Decision

We ultimately settled on implementing a VS Code plugin with a language server and a TextMate grammar summarized in the following Y-Statement:

In the context of IDE support for SAMT, facing usability and compatibility needs, we decided for implementing a VS Code plugin with a Language Server and a TextMate grammar and neglected implementing an IntelliJ plugin, to achieve easy portability to other tools, accepting downside no first class support in JetBrains IDEs.

Because IntelliJ also supports TextMate grammars, it may still be possible to create a simple syntax highlighting plugin for IntelliJ, although this is not a priority.

[11] *Visual Studio Code - Syntax Highlight Guide*, Microsoft, 2023-03. [Online]. Available: https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide (visited on 2023-03-03).

[12] https://www.jetbrains.com/fleet/

7.4 Technical Architecture

The technical architecture provides a framework for the implementation as part of this thesis as well as future development.

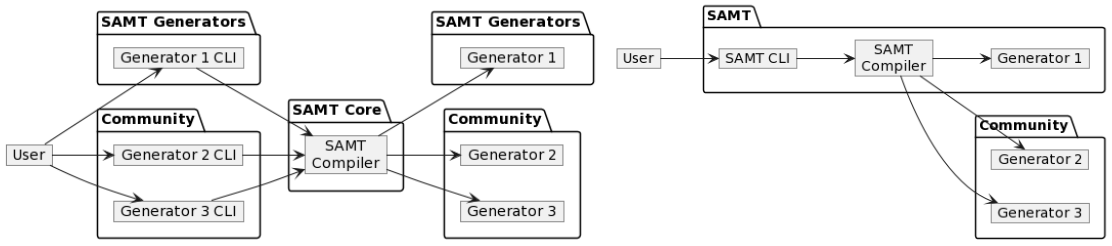
The following subsections paint a holistic picture of SAMT, including future enhancements that could not be implemented due to time constraints. Such elements are explicitly designated as out-of-scope for this project, but are nevertheless relevant due to their importance for the potential integration into Zürcher Kantonalbank.

7.4.1 Hard Architectural Problems

The following section explains hard architectural problems and how they are solved in SAMT.

Supporting Multiple Generators

A key feature of SAMT is the extensible support for different generators, allowing the community to add support for new programming languages with ease. One way to achieve this would be to implement an extensible architecture within SAMT and expect open-source contributions to add new generators into the public source code, but this would not work for enterprise customers like Zürcher Kantonalbank. Another idea would be to distribute SAMT as a library instead of an executable. However, this would have a negative impact on the usability of SAMT as shown in Figure 7.2.



(a) SAMT as a library (b) Single SAMT CLI

Figure 7.2: Comparing different CLI approaches

Version mismatches pose another problem, as a newer project might depend on a project using an older version and so on. This problem also exists in the opposite direction: what if a user wants to run a generator which was not built against the latest version of SAMT? Our final decisions are summarized in the following Y-statement:

In the context of core architectural decisions, facing extendability, simplicity and maintainability needs, we decided to create a single CLI and implementing a plugin-system for generators and neglected having standalone generators for every language. This was done to achieve a uniform and simple end-user experience, accepting downside of added implementation complexity and backwards compatibility handling.

Managing Dependencies Within SAMT Models

In the context of dependency resolution, facing programming language independence, ease-of-use and corporate needs, we decided for implementing plugin and package dependency resolution within the SAMT CLI and neglected integrating an existing package manager (e.g. Maven or Gradle). This was done to achieve a single entry-point for the end-user and ensure SAMT has minimal exposure to other tools, accepting downside of greatly increased implementation effort and additional server infrastructure costs.

SAMT itself provides a documented API to interact with the compiler, extract metadata and facilitate code generation. The installation and management of SAMT plugins and packages is not yet implemented.

Future-Proofing Generator Framework

Based on experience within Zürcher Kantonalbank, a new transport technology is introduced every few years. This means that SAMT and the community generators need to be able to adapt to new technologies at a rapid pace. Unfortunately, there is no way to predict what the next technology will be, so we need to make sure that the generator framework is as flexible as possible. This comes at the cost of increased complexity for generator developers, as they'll need to implement more features themselves. For example, if a transport protocol does not support the concept of packages (e.g. `tools.samt.v1.Person` and `tools.samt.v2.Person` can coexist), the generator must handle name collisions across different packages.

SAMT Configuration

In addition to the SAMT model, the user will also need to provide a configuration for SAMT itself. For example, the user has to specify which generator to use or where the generated code should be placed. Passing such configuration as command line arguments is not user-friendly, as the user would need to remember the arguments and type them out every time. For this reason, a configuration file called `samt.yaml` is used. This file is located in the root of the project and contains all the configuration options. Listing 7.2 shows an example configuration file.

```yaml
source: src

generators:
  - name: kotlin-ktor-provider
    output: ../ktor-server/src/main/kotlin
    options:
      removePrefixFromSamtPackage: tools.samt
      addPrefixToKotlinPackage: tools.samt.server.generated
      skipKtorServer: "true"
  - name: kotlin-ktor-consumer
    output: ../ktor-client/src/main/kotlin
    options:
      removePrefixFromSamtPackage: tools.samt
      addPrefixToKotlinPackage: tools.samt.client.generated
```
Listing 7.2: Example `samt.yaml` configuration file

In addition, a `.samtrc.yaml` for style configuration is planned. This file would be used to configure style and linter rule, for example, the maximum line length or the allowed number of parameters for an operation. The style configuration has not yet been implemented.

7.4.2 Container

With the different uses of SAMT in mind, the target architecture has to be both extensible and well-defined. In order to enable this, a detailed architecture was created, which is shown in Figure 7.4. SAMT Core represents a single git repository, but the contained components are still distributed separately. The Visual Studio Code Extension is in separate git repositories and released independently. The SAMT Wrapper only consists of a few script files and is not further discussed here, see Section 8.6 for more details.

Every SAMT User interfaces with SAMT through either Visual Studio Code or the SAMT Wrapper. The Visual Studio Code Extension enables the user to edit SAMT models with a good user experience. To enable this, the extension downloads the Language Server at runtime and communicates with it through LSP. Alternatively, the user can interact with the SAMT CLI, either through the SAMT Wrapper or the Visual Studio Code Extension.

Both the CLI and the Language Server rely on the compiler to do most of the heavy lifting. In addition, they also both depend on the SAMT Config to load configuration files. The CLI will forward the parsed model to the code-generation stage, which will then call the appropriate generator at runtime. A generator can also provide a custom transport technology parser, maximizing the flexibility of the generator framework.¨ Potential plugins are shown in Figure 7.3, illustrating the extensibility of the architecture.

The plugins with their respective generator and transport technology parsers will be loaded at runtime through a plugin loader in the future. Due to the project scope, the example generator is part of the Codegen component. Still, a dedicated Public API library is provided to enable the creation of custom generators against a binary-compatible API.

SAMT Core also contains the `Common` component, which was omitted here for brevity. It is used by most of the components and contains common types and utilities which are used across the SAMT ecosystem.

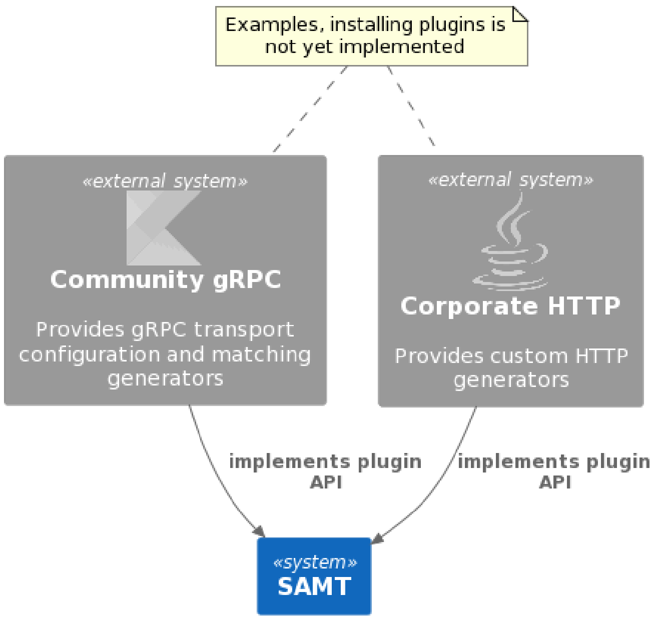

Examples, installing plugins is not yet implemented
«external system»
Community gRPC
Provides gRPC transport configuration and matching generators
«external system»
Corporate HTTP
Provides custom HTTP generators
implements plugin API
implements plugin API
«system»
SAMT

Figure 7.3: C4 container plugins diagram

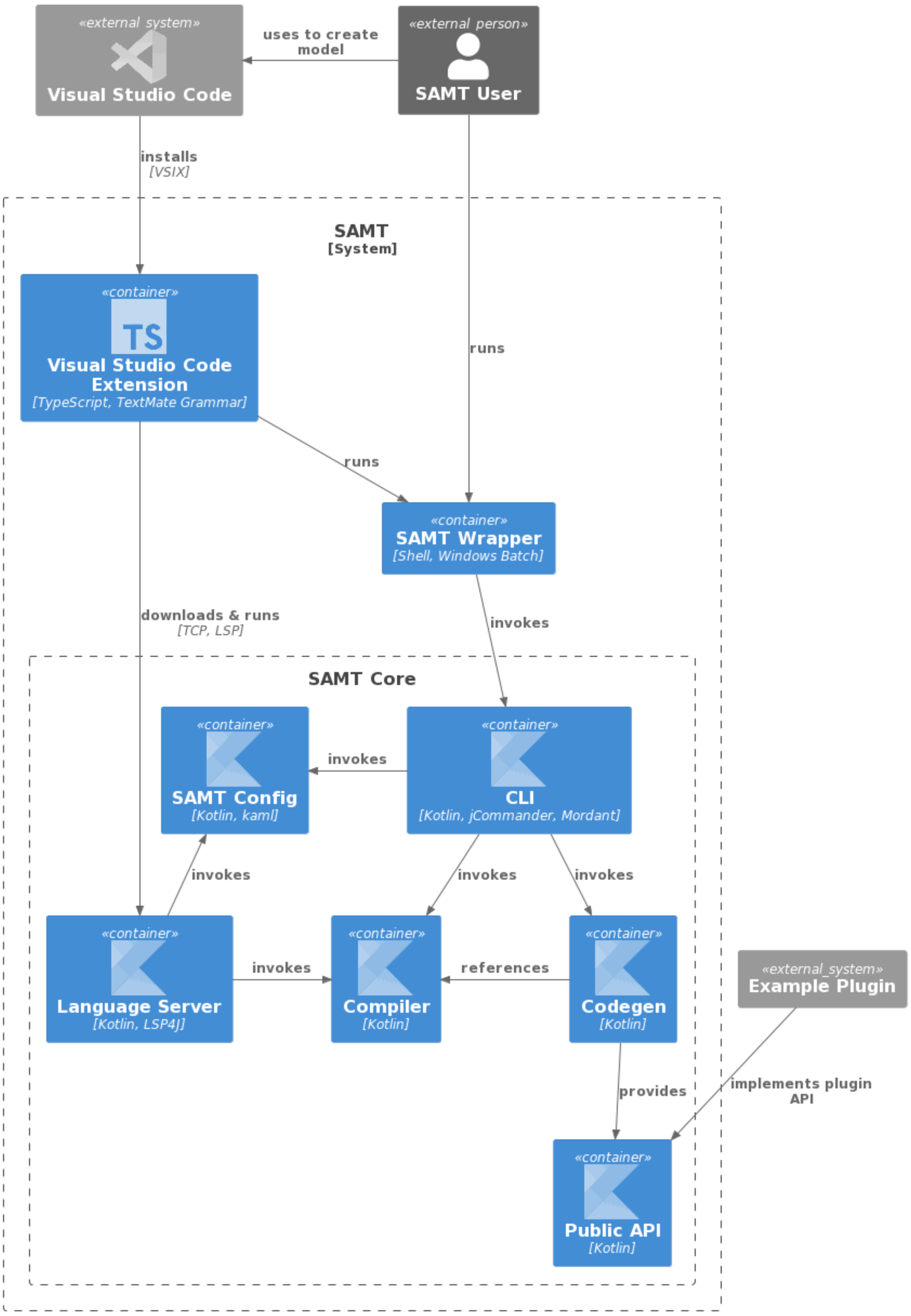


Figure 7.4: C4 container diagram

7.4.3 Component

An architectural overview of the Compiler, the most crucial part of SAMT, is shown in Figure 7.5. Full control over the input, output and error handling is obtained by the component that executed the Compiler. That is, the CLI, Language Server, or software depending on the library. For example, the CLI always parses all the files from start to finish, while the Language Server only parses the files which have been updated.

If we assume the CLI as the Compiler Client, the following steps are executed as part of the compilation process:

1. The CLI parses the command line arguments and loads the SAMT model from the file system.
2. The CLI calls the Lexer with the SAMT source code as input, which returns a token sequence.
3. The CLI calls the Parser with the token sequence as input, which returns an Abstract Syntax Tree (AST).
4. The CLI aggregates all the ASTs into a list.
5. The CLI calls the Semantic Checker with the list of ASTs as input, which returns the type information.

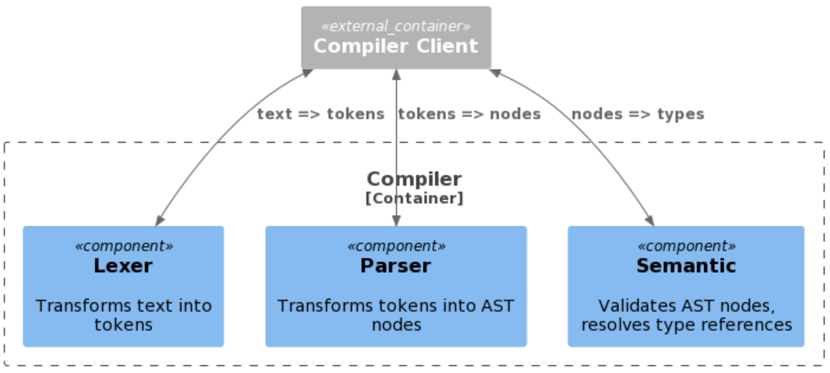


Figure 7.5: C4 component diagram for compiler

7.4.4 Compiler Architecture

This chapter focuses on the code structure of the compiler. The data models are ordered left-to-right as shown in Figure 7.5.

Lexer

The UML class diagram in Figure 7.6 shows the Token class hierarchy produced by the Lexer component. All `Token` subtypes are shown, except for the various `StaticToken` (representing language keywords) and `StructureToken` (representing punctuation characters such as braces and commas), which were omitted for brevity.

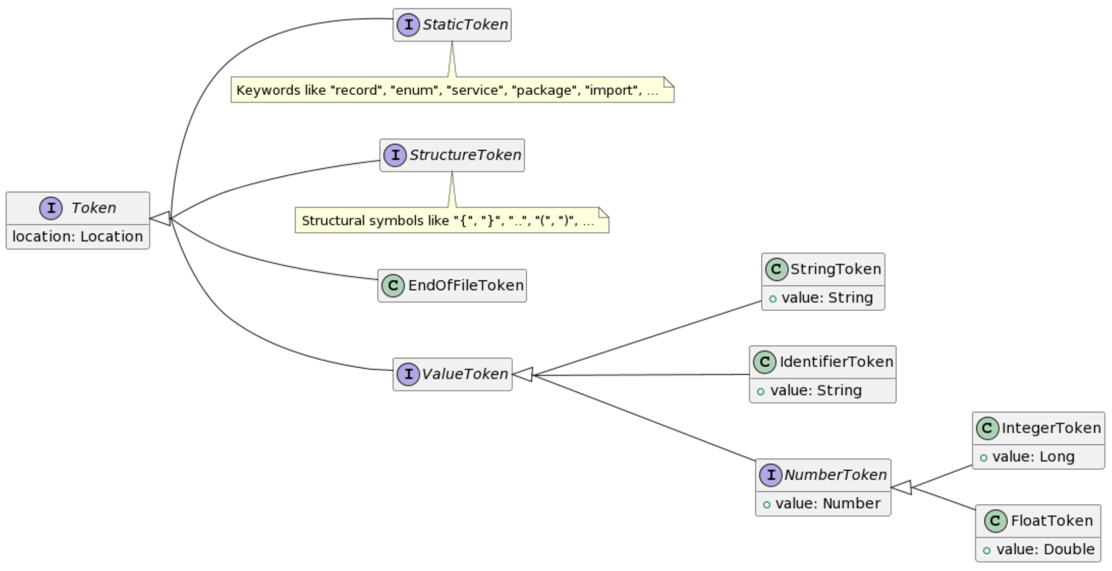
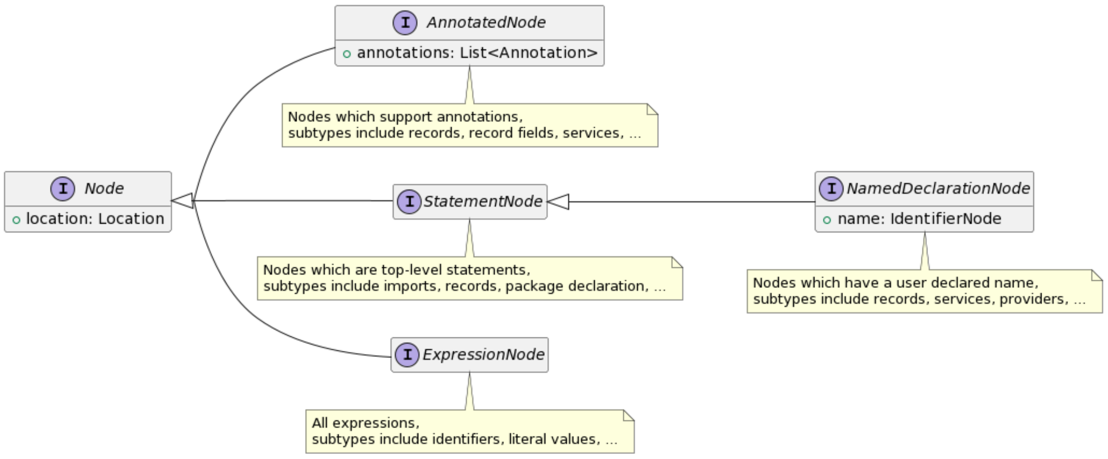


Figure 7.6: Tokens class diagram

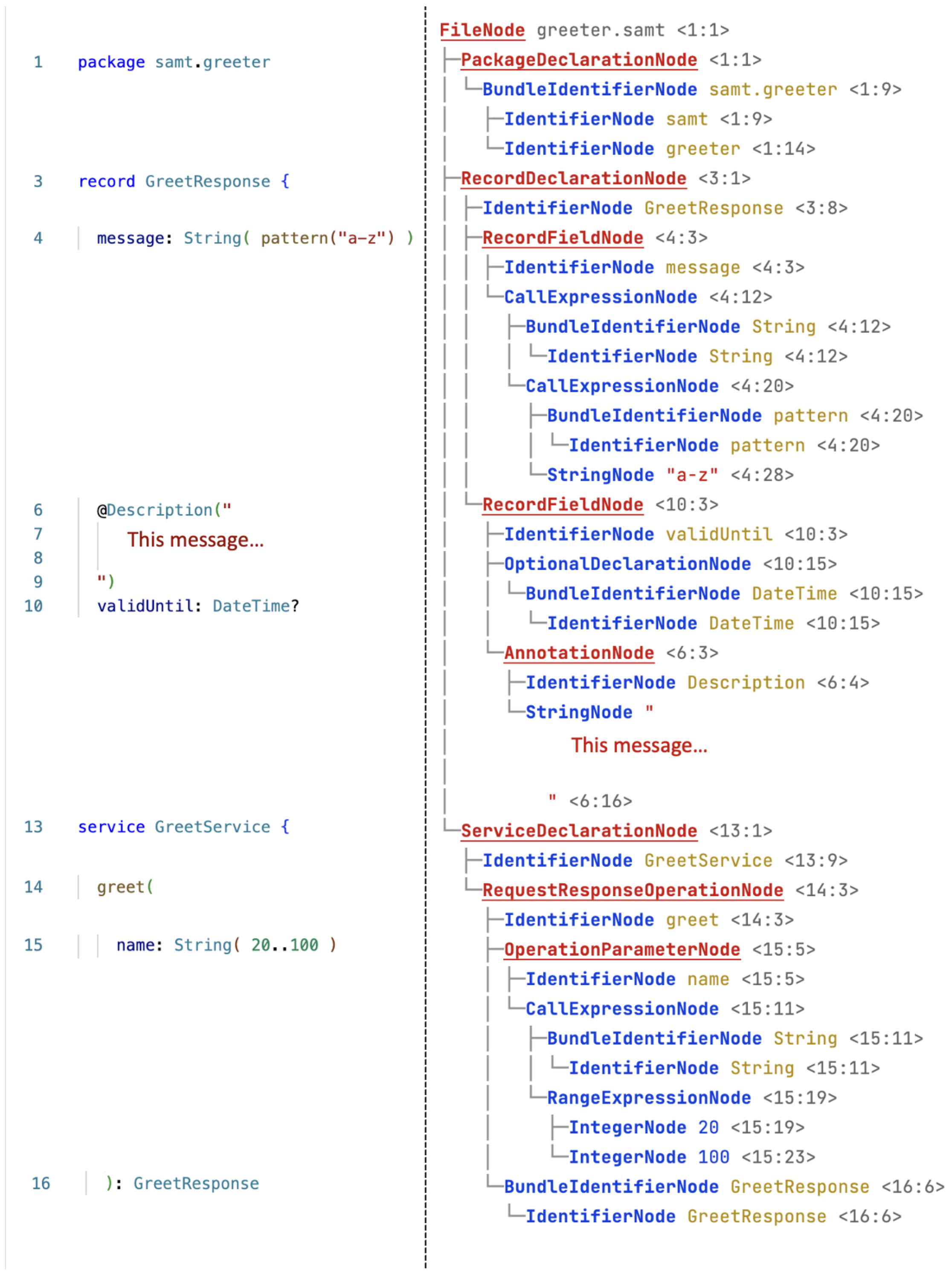
Parser

The simplified UML class diagram in Figure 7.7 shows the Node class hierarchy produced by the Parser component. Given that a full class diagram of the AST nodes would not fit on a single page, the AST's structure is shown in an example instead. Figure 7.8 shows a simple SAMT source file, Figure 7.9 enhances the previous figure to show each line of code side by side with the corresponding node of the AST.


AnnotatedNode
annotations: List<Annotation>
Nodes which support annotations,
subtypes include records, record fields, services, ...
Node
location: Location
StatementNode
NamedDeclarationNode
name: IdentifierNode
Nodes which are top-level statements,
subtypes include imports, records, package declaration, ...
Nodes which have a user declared name,
subtypes include records, services, providers, ...
ExpressionNode
All expressions,
subtypes include identifiers, literal values, ...
Figure 7.7: Nodes class diagram

```
package samt.greeter

record GreetResponse {
  message: String( pattern("a-z") )

  @Description("
    This message should be destroyed once this date time is reached.
    If it is null, this response can live on forever.
  ")
  validUntil: DateTime?
}

service GreetService {
  greet(
    name: String( 20..100 )
  ): GreetResponse
}
```

Figure 7.8: Example SAMT model



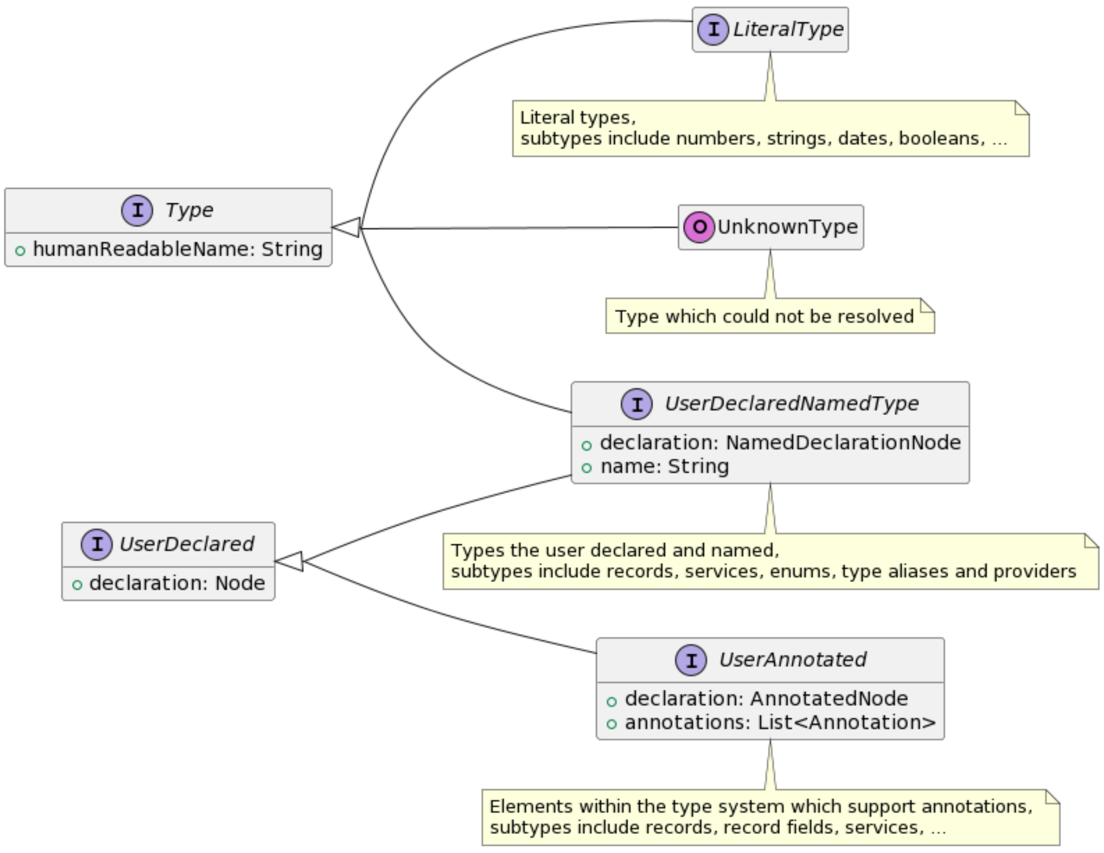
```
1   package samt.greeter

3   record GreetResponse {

4     message: String( pattern("a-z") )

6     @Description("
7       This message...
8
9     ")
10    validUntil: DateTime?

13  service GreetService {

14    greet(

15      name: String( 20..100 )

16    ): GreetResponse
```

```
FileNode greeter.samt <1:1>
├PackageDeclarationNode <1:1>
│ └BundleIdentifierNode samt.greeter <1:9>
│   ├IdentifierNode samt <1:9>
│   └IdentifierNode greeter <1:14>
├RecordDeclarationNode <3:1>
│ ├IdentifierNode GreetResponse <3:8>
│ ├RecordFieldNode <4:3>
│ │ ├IdentifierNode message <4:3>
│ │ └CallExpressionNode <4:12>
│ │   ├BundleIdentifierNode String <4:12>
│ │   │ └IdentifierNode String <4:12>
│ │   └CallExpressionNode <4:20>
│ │     ├BundleIdentifierNode pattern <4:20>
│ │     │ └IdentifierNode pattern <4:20>
│ │     └StringNode "a-z" <4:28>
│ └RecordFieldNode <10:3>
│   ├IdentifierNode validUntil <10:3>
│   ├OptionalDeclarationNode <10:15>
│   │ └BundleIdentifierNode DateTime <10:15>
│   │   └IdentifierNode DateTime <10:15>
│   └AnnotationNode <6:3>
│     ├IdentifierNode Description <6:4>
│     └StringNode "
│             This message...
│
│           " <6:16>
└ServiceDeclarationNode <13:1>
  ├IdentifierNode GreetService <13:9>
  └RequestResponseOperationNode <14:3>
    ├IdentifierNode greet <14:3>
    ├OperationParameterNode <15:5>
    │ ├IdentifierNode name <15:5>
    │ └CallExpressionNode <15:11>
    │   ├BundleIdentifierNode String <15:11>
    │   │ └IdentifierNode String <15:11>
    │   └RangeExpressionNode <15:19>
    │     ├IntegerNode 20 <15:19>
    │     └IntegerNode 100 <15:23>
    └BundleIdentifierNode GreetResponse <16:6>
      └IdentifierNode GreetResponse <16:6>
```

Figure 7.9: Example SAMT model side by side with AST produced by the SAMT Parser

Semantic

Finally, the UML class diagram in Figure 7.10 shows the Type class hierarchy produced by the Semantic component.



I LiteralType
Literal types,
subtypes include numbers, strings, dates, booleans, ...
I Type
humanReadableName: String
O UnknownType
Type which could not be resolved
I UserDeclaredNamedType
declaration: NamedDeclarationNode
name: String
Types the user declared and named,
subtypes include records, services, enums, type aliases and providers
I UserDeclared
declaration: Node
I UserAnnotated
declaration: AnnotatedNode
annotations: List<Annotation>
Elements within the type system which support annotations,
subtypes include records, record fields, services, ...

Figure 7.10: Types class diagram

7.4.5 Code Generator Architecture

This section describes various code generator architectures we considered, their advantages and disadvantages, and our final decision.

Existing Approach

The implementation of the existing language used at Zürcher Kantonalbank, shown in Figure 7.11, relies on a runtime library to interface with the required transport technology. A runtime library in their respective target language is responsible to serialize / deserialize messages and to provide a common interface for sending and receiving messages. Code generation tools emit calls into the runtime libraries to perform actual communication work.

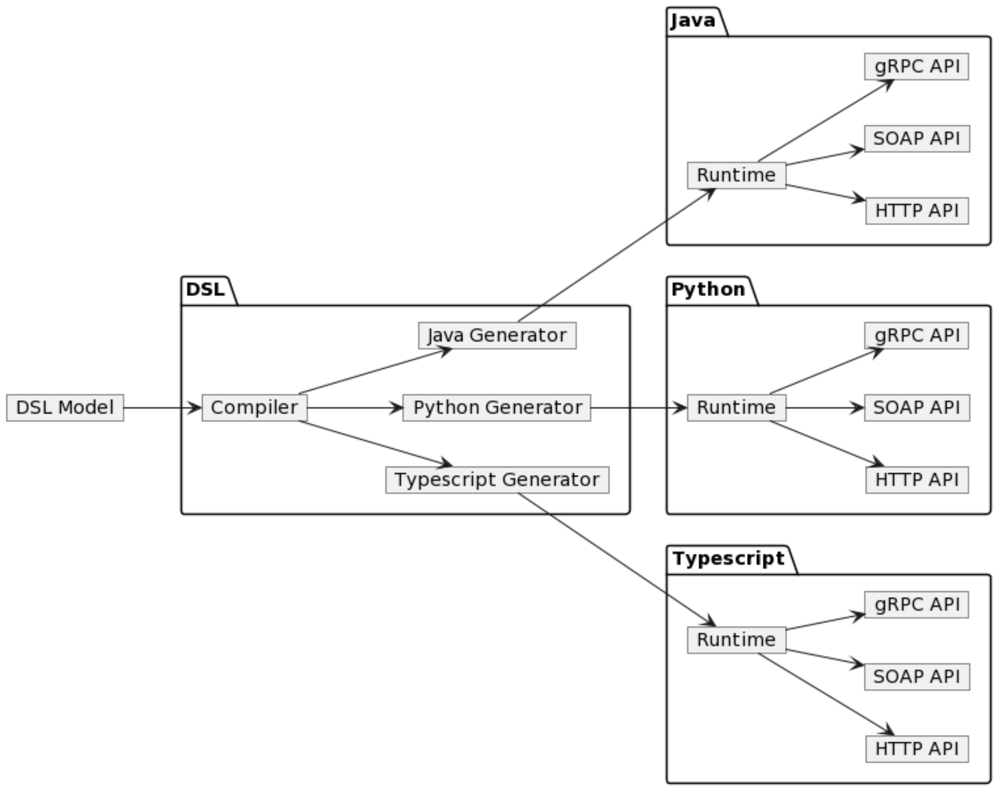


Figure 7.11: Generator architecture of the current DSL

Monolithic

With the monolithic architecture, shown in Figure 7.12, each generator implements a single combination of target language and transport technology. This generator is responsible for generating all the code required to interface with the transport technology. To implement support for a new target language or transport technology, a new generator must be written for that combination.

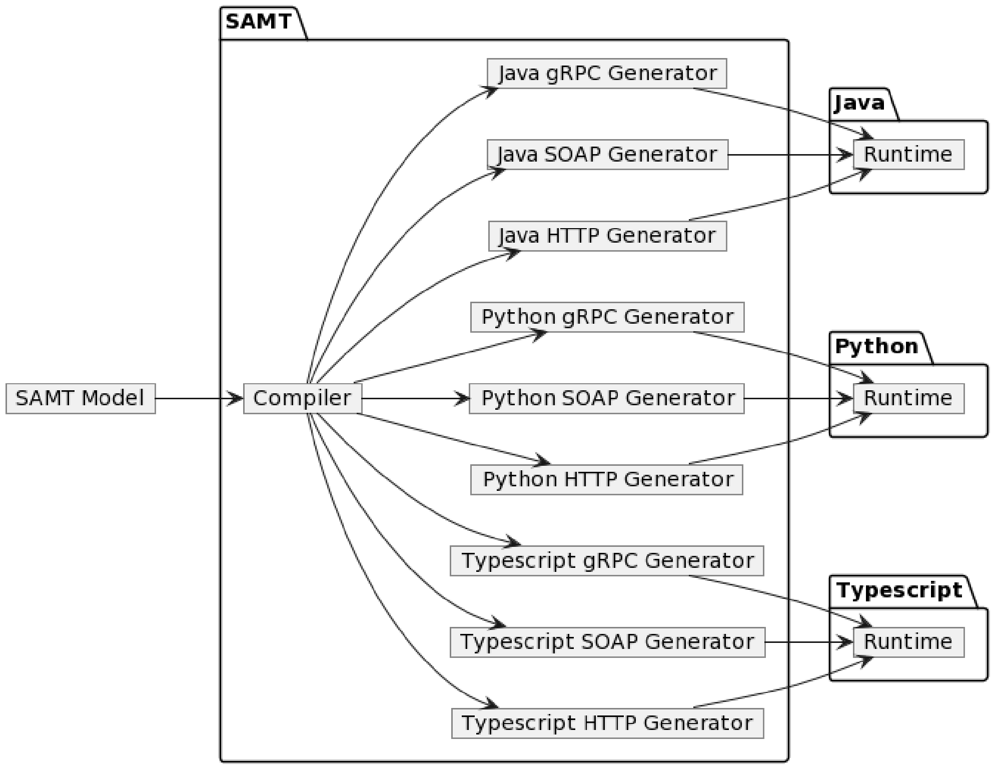


Figure 7.12: Monolithic generator architecture

Intermediate Representation

The intermediate representation (IR) architecture, shown in Figure 7.13, is based on the idea of a common intermediate representation that is shared among all generators. Instead of generating code from the SAMT model directly, each generator transforms the IR into the target language. This essentially relocates the complexity of the code generation from the generator to the IR. Due to the potential complexity introduced, this approach is by far the hardest to implement. It is also difficult to estimate the configurability of this approach, since each feature in the emitted code needs to be represented in the IR itself – whether as a dedicated node or as a collection of other primitive nodes.

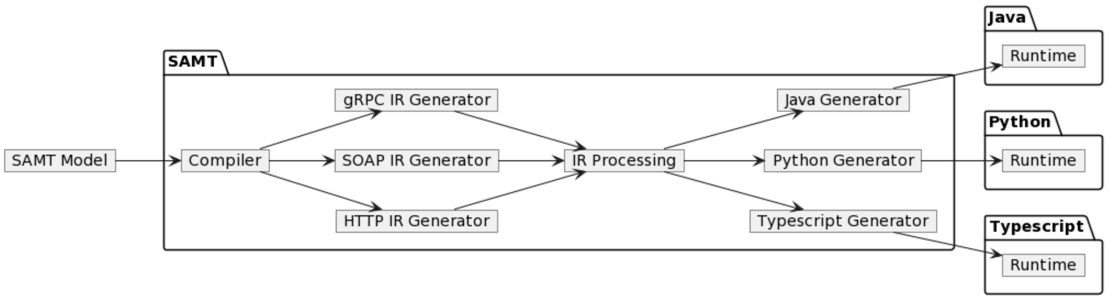


Figure 7.13: Intermediate representation (IR) generator architecture

Transport Buddy

The “Transport Buddy” approach, shown in Figure 7.14, is based on the idea of a separate actor that handles all transport related work. Generators emit code that interfaces with this actor to perform any network operations and serialization work. This approach is similar to the “Existing Approach” of Section 7.4.5 in the regard that language generators refrain from handling transport specifics and instead rely on the runtime. Conversely, the runtime is no longer part of a library which is implemented for each platform. Instead, a separate entity is implemented once, which enables new language generators to immediately gain access to all previously supported transport protocols. Additionally, new transport protocols become available to all previously supported languages.

The exact implementation of the transport buddy is left unspecified, but possible implementations include a separate worker process or a background thread. It is unclear how this approach would work in an environment that does not support communication between external processes and threads, such as a web browser. The configurability problem present with the IR approach becomes apparent as each possible transport-related feature relies on an implementation in the transport buddy.

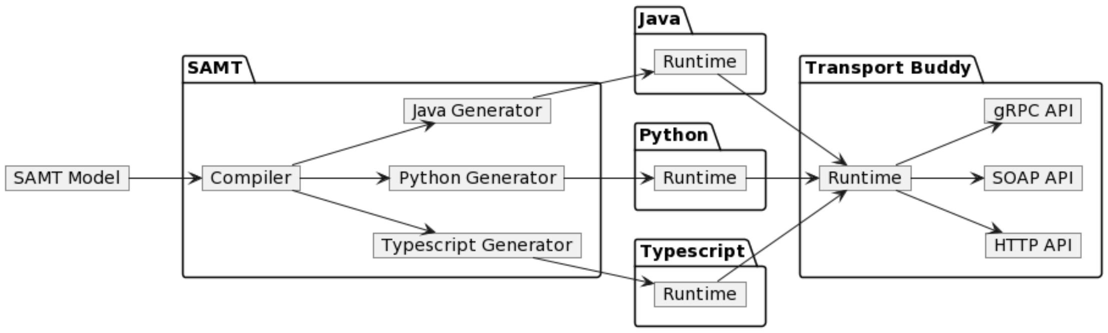


Figure 7.14: Transport buddy generator architecture

Decision

The different approaches listed above all have their own advantages and disadvantages. For our purpose, we decided to implement a proof-of-concept monolithic architecture, shown in Figure 7.15. Our rationale is summarized in the following Y-statement: In the context of the code generator architecture, facing the need for an easy implementation we decided for a proof of concept monolithic architecture and neglected the IR and transport-buddy approaches to achieve a functional proof of concept accepting that this decision will have to be revisited at a later date. Additionally, choosing the simplest approach allowed us to focus on other issues. This architecture is not intended to be final and a suitable replacement must be evaluated in the future. The implementation of this architecture is described in Section 8.2.

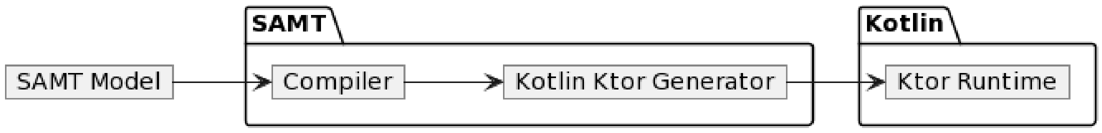


Figure 7.15: Proof of concept generator architecture

Chapter 8

Implementation

This chapter documents the final state of the product.

8.1 Compilation Internals

The Compiler as described in Section 7.4.3 consists of three components: The Lexer, Parser, and Semantic. Since the Codegen component finalizes this compilation process, the entire process can be thought of as a pipeline, where the output of one stage corresponds to the input of the subsequent stage.

Figure 8.1 shows the simplified compilation process with a focus on the data flow between the components. The SAMT model is provided by the SAMT CLI, the SAMT Visual Studio Code Extension or a custom frontend. Similarly, the code generator could be a custom generator. This section focuses on the middle part of the compilation pipeline, where no dynamic components are involved.

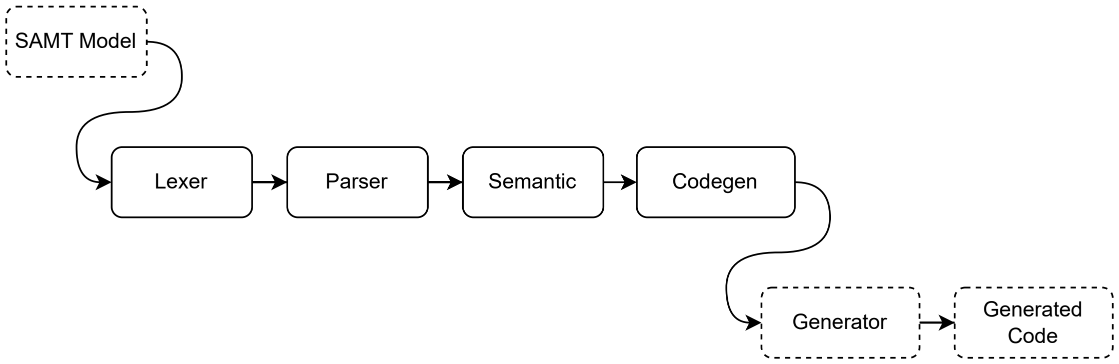


Figure 8.1: Simplified SAMT compilation pipeline

8.1.1 Lexer

The Lexer is the first stage of the compilation process. Each source file is split into a sequence of tokens, representing the smallest meaningful units of the language. The Lexer performs some basic source processing, such as expanding escape sequences within strings, removing comments and whitespace, and checking for unclosed strings and invalid characters. This has the intended side effect of making the language whitespace insensitive and therefore provides developers more freedom to format their code.

The Lexer generally reads character by character from the source file with the exception of the range syntax. When the Lexer encounters a period after a number (`1.`), it looks ahead one additional character and decides whether to produce a token corresponding to a floating point number literal (`1.2`) or the start of a range (`1..2`).

8.1.2 Parser

The Parser takes in the sequence of tokens produced by the Lexer and constructs an Abstract Syntax Tree (AST). The AST is a tree-like data structure that represents the syntactic structure of the source code. Each node in the tree represents a different construct, such as a string literal or a type declaration. The Parser checks for syntax errors pertaining to the order of statements, invalid grammatical constructs, and missing tokens.

The Parser is implemented as a recursive-descent parser, which is a type of top-down parser. Only a single token is consumed at a time, making the code relatively simple and easy to understand. The implementation itself mostly follows the structure of the grammar in Section 6.2.1. Since type information is not yet available during the parsing phase, the Parser is unable to check for semantic errors. Transport configuration objects are parsed as general dictionaries of values and are not yet validated. The complete AST is then passed to the semantic analysis stage.

8.1.3 Semantic Analysis

The semantic analysis stage is responsible for checking the AST for semantic errors and inconsistencies. An initial pass checks for duplicate elements, parameters and other errors that can be detected without type information. Subsequent passes resolve type declarations, type references, and aliases. During this stage, all semantic rules laid out in Section 6.2.2 are enforced. If the model is absent of any errors, this stage yields a fully resolved SAMT model that can be passed to the configured code generator.

8.1.4 Code Generation

SAMT can be configured to run with any generator that supports the public API. These generators can implement support for any target language and transport technology. Each generator can provide custom transport configuration parsers, which consume the configuration objects provided by the model. Multiple generators can be invoked at once, allowing one generator to build on top of the work of the previous one. As mentioned in Section 7.4.5, we decided to implement a proof of concept architecture based on the monolithic approach. This means that the generator is responsible for generating all the code for the chosen target language and transport protocol.

8.2 Ktor Generators

We built three separate generators for the Kotlin programming language, implementing support for the Web framework Ktor.[1]

KotlinTypesGenerator Emits type and interface declarations

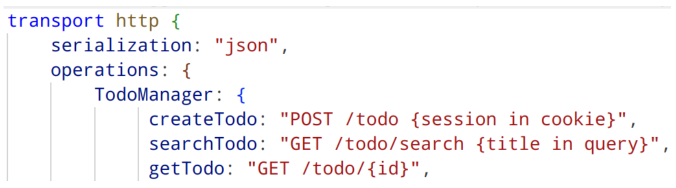
KotlinKtorProviderGenerator Emits a Ktor server application

KotlinKtorConsumerGenerator Emits a Ktor client application

The generators all share their core structure. Each generator iterates over all the packages that are provided by the model, extracts the relevant providers and consumers and emits the required client and server bindings. The HTTP transport configuration described in Section 8.3 is used to configure the Ktor server, such as HTTP methods, path and type of parameters.

8.3 HTTP Transport Configuration Parser

As part of the generator built for the Web framework Ktor, a transport configuration parser for HTTP was implemented. The parser is responsible for extracting the HTTP method, path and parameters of each operation. Parameters can be declared either as inline path parameters, being passed as part of the URL, or as query, header, cookie or body parameters. Non-path parameters are declared via the {`parameterName in parameterType`} syntax. Multiple parameters can be declared by separating their names with commas.



```
transport http {
    serialization: "json",
    operations: {
        TodoManager: {
            createTodo: "POST /todo {session in cookie}",
            searchTodo: "GET /todo/search {title in query}",
            getTodo: "GET /todo/{id}",
```

Figure 8.2: Screenshot of an example HTTP transport configuration inside Visual Studio Code

Figure 8.2 shows an example transport configuration with three operations. If no configuration entries are present for an operation, the parser will default to HTTP POST and a unique path based on the service and operation names.

[1]https://ktor.io/

8.4 SAMT CLI

As decided in Section 7.4.1, a CLI interface to SAMT has been developed. The CLI can be used to initialize, inspect, and compile projects. It is usually invoked using the SAMT Wrapper described in Section 8.6.

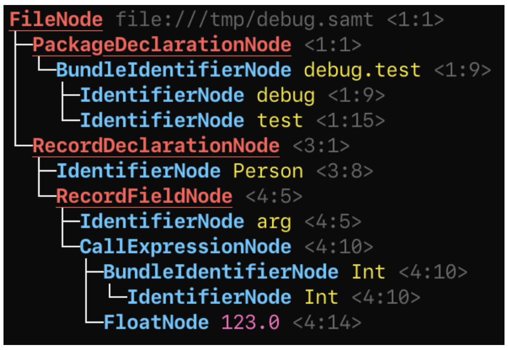
8.4.1 Commands

The following commands are available in the CLI:

compile
: Used to compile a SAMT project and generate the corresponding target files. If any errors occur during compilation, they will be displayed in the CLI.

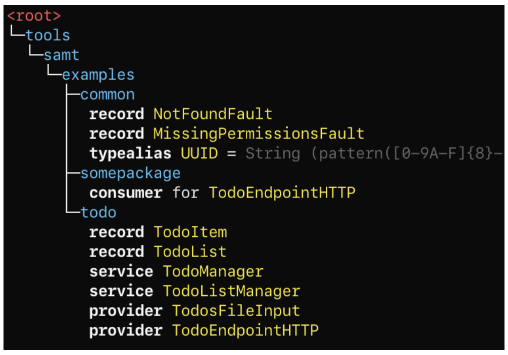
dump
: Used to dump various types of debug information, such as the parsed AST and the type hierarchy, as shown in Figures 8.3 and 8.4.

wrapper
: Used to initialize an instance of the SAMT wrapper in the current directory. An optional flag can be set to control the location from which the initial “samtw”, “samtw.bat” and “samt-wrapper.properties” files will be downloaded from.



```
FileNode file:///tmp/debug.samt <1:1>
├─PackageDeclarationNode <1:1>
│ └─BundleIdentifierNode debug.test <1:9>
│   ├─IdentifierNode debug <1:9>
│   └─IdentifierNode test <1:15>
└─RecordDeclarationNode <3:1>
  ├─IdentifierNode Person <3:8>
  └─RecordFieldNode <4:5>
    ├─IdentifierNode arg <4:5>
    └─CallExpressionNode <4:10>
      ├─BundleIdentifierNode Int <4:10>
      │ └─IdentifierNode Int <4:10>
      └─FloatNode 123.0 <4:14>
```

Figure 8.3: Visual representation of the AST

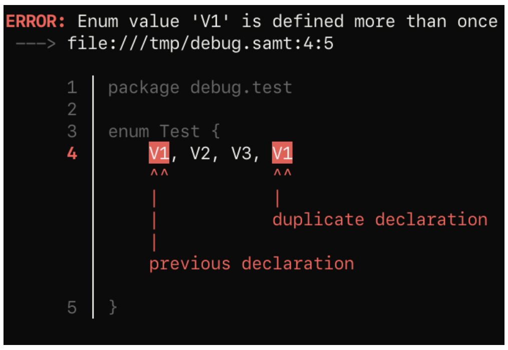


```
<root>
└─tools
  └─samt
    └─examples
      ├─common
      │ record NotFoundFault
      │ record MissingPermissionsFault
      │ typealias UUID = String (pattern([0-9A-F]{8}-
      ├─somepackage
      │ consumer for TodoEndpointHTTP
      └─todo
        record TodoItem
        record TodoList
        service TodoManager
        service TodoListManager
        provider TodosFileInput
        provider TodoEndpointHTTP
```

Figure 8.4: Type structure of an example todo application

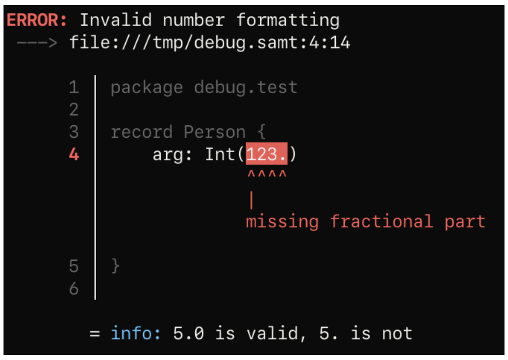
8.4.2 Message Formatting

Error messages shown via the CLI interface should be just as intuitive and easy to read as the ones displayed in an IDE. For this purpose, a custom error formatter was written that can highlight individual sections of a source file and draw accompanying error messages with a small red arrow pointing them toward the location in the source code. Because the file location is URI encoded, the link can be clicked to open the file in the default editor.



```
ERROR: Enum value 'V1' is defined more than once
 ---> file:///tmp/debug.samt:4:5

   | package debug.test
   |
   | enum Test {
   |     V1, V2, V3, V1
   |     ^^          ^^
   |     |           |
   |     |           duplicate declaration
   |     |
   |     previous declaration
   |
   | }
```

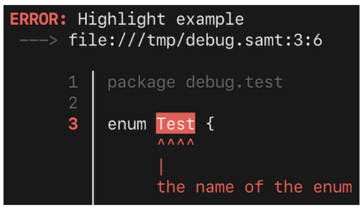
Figure 8.5: Error message containing two highlighted code sections



```
ERROR: Invalid number formatting
 ---> file:///tmp/debug.samt:4:14

   | package debug.test
   |
   | record Person {
   |     arg: Int(123.)
   |              ^^^^
   |              |
   |              missing fractional part
   |
   | }
   |

  = info: 5.0 is valid, 5. is not
```

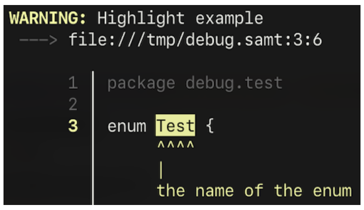
Figure 8.6: Error message with info and help line

The diagnostic formatter draws each error with colorized highlights of the relevant source code and corresponding messages. The surrounding code is printed in a gray color to provide context but not distract too much. Figure 8.5 shows how a duplicate declaration of an enum value shows up in the CLI. Figure 8.6 shows an info line displayed below the highlighted source code, providing the user with further information about the error.



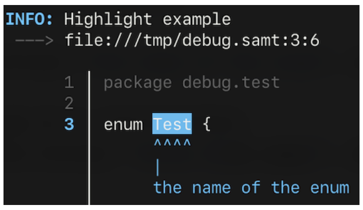
```
ERROR: Highlight example
 ---> file:///tmp/debug.samt:3:6

   | package debug.test
   |
   | enum Test {
   |      ^^^^
   |      |
   |      the name of the enum
```

Error



```
WARNING: Highlight example
 ---> file:///tmp/debug.samt:3:6

   | package debug.test
   |
   | enum Test {
   |      ^^^^
   |      |
   |      the name of the enum
```

Warning



```
INFO: Highlight example
 ---> file:///tmp/debug.samt:3:6

   | package debug.test
   |
   | enum Test {
   |      ^^^^
   |      |
   |      the name of the enum
```

Info

Figure 8.7: Different types of diagnostic output types

Messages can be displayed in three different types, depending on their severity, as shown in Figure 8.7. Errors block the compilation process until the problem is resolved. Warnings can be ignored but provide useful suggestions. Info messages contain log output from different stages.

8.5 Extending SAMT

A key feature of SAMT is its extensibility. It was designed to be easily extensible by third-party developers without requiring them to modify the SAMT source code. There are three intended ways to extend SAMT:

1. Add new generators
2. Add new transport technology
3. Custom SAMT frontend

8.5.1 Adding SAMT as a Dependency

The first step to extend SAMT is to add it as a dependency to your project. To enable this, different SAMT artifacts are published to the Maven Central Repository.[2] The available artifacts are listed in Table 8.1.

Group ID	**Artifact ID**	**Description**
tools.samt	public-api	Public API of SAMT, used when creating a SAMT plugin
tools.samt	compiler	Library for compiling any SAMT source files
tools.samt	samt-config	Library to parse SAMT project configuration files
tools.samt	common	Library containing common utility functions and classes for SAMT
tools.samt	codegen	Library for invoking the appropriate code generators

Table 8.1: Descriptions of the packages published to Maven Central

8.5.2 Adding New Generators

The easiest way of extending SAMT is to build a new generator. Generators are classes that implement the `Generator` interface from the `public-api` package. A generator receives the root SAMT package as an input and returns a list of generated files as an output.

The model types within the `public-api` are designed to be easily usable for code generation. A simplified meta model is shown in Figure 8.8. Some details have been omitted for brevity, but the general structure is still visible. A Java generator might loop over all the record instances and generate a Java class for each. For usages of types, for example the type of a field, a `TypeReference` is used. While every `Type` only exists once, there can be multiple `TypeReference` instances pointing to the same `Type`.

[2] https://central.sonatype.com/namespace/tools.samt

When generating network code, the `TransportConfiguration` instances can be used to generate the appropriate code for the selected transport technology. The generator simply checks whether the `TransportConfiguration` is an instance of the desired transport technology and generates the appropriate code. Currently only one transport technology is implemented, `SamtHttpTransport`. The process of adding a new transport technology is described in Section 8.5.3. A more detailed explanation of how to apply the meta model is given in the SAMT Authoring Generators guide on GitHub.[3]

[3] `https://github.com/samtkit/core/wiki/Authoring-Generators`

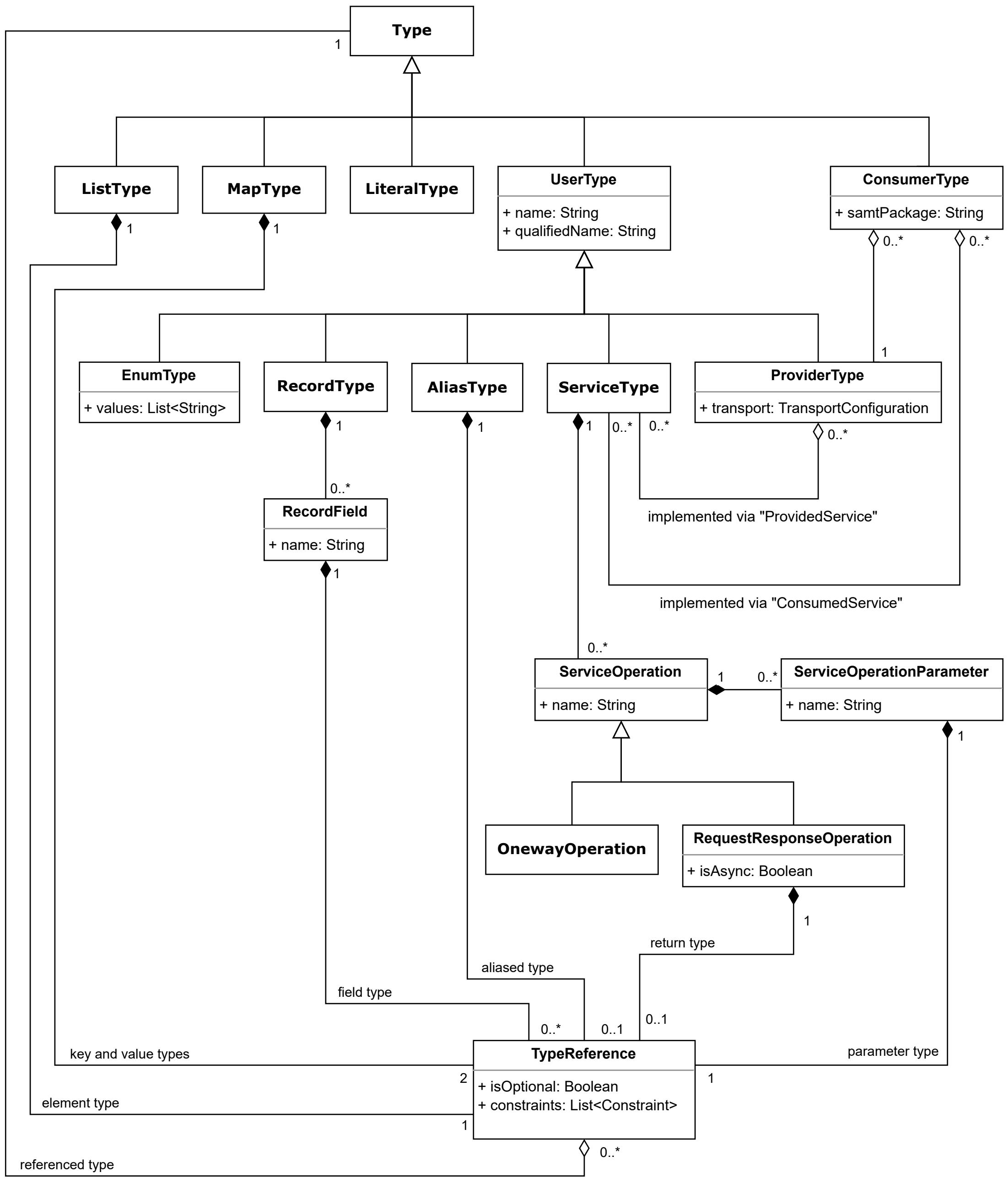


Figure 8.8: Class diagram of the SAMT meta model

8.5.3 Adding New Transport Technology

Configuration parsers for new transport technologies can be added to SAMT by implementing the `TransportConfigurationParser` interface from the `public-api` package. A transport configuration parser receives the transport configuration data from a Provider as input and returns a `TransportConfiguration` instance as output. The configuration data is wrapped in a `ConfigurationObject` instance from the `public-api` package. The purpose of this wrapper is to provide a unified interface for accessing configuration data, independent of the format that the previous stages consume (e.g. an AST).

8.5.4 Custom SAMT Frontend

If you want to extend SAMT in any other way, for example, by providing an API portal, a web-based editor, or a custom CLI, you can use the published packages to do so. The public interfaces of every module are designed to be easily usable by third-party developers from any JVM language, such as Java, Kotlin, Scala, or Groovy. As a caller of the SAMT Compiler component, you also get full control over error handling, dependency management, and so on.

8.6 SAMT Wrapper

SAMT is supposed to be simple and usable. This means that it should be easy to get started with SAMT, without having to install a lot of dependencies. The SAMT CLI described in Section 8.4 is a good example of this approach, requiring only the download of a single JAR file to run SAMT. However, downloading a JAR file and placing it in the correct directory is not very user-friendly. Another approach would have been to create a SAMT installer and distribute the SAMT CLI as a full-featured application, which is not very user-friendly either. Instead, we decided to use a more simple approach used by other popular build tools in the Java ecosystem.

8.6.1 Wrapper Script

We decided to follow the path of other build tools and implemented a wrapper script. Prominent examples of this approach are the Gradle Wrapper[4] and Maven Wrapper[5]. The wrapper is a way to automatically download and run the required binaries without requiring the user to install any additional software. In our case, the wrapper is implemented using a shell script for Linux and macOS and a batch script for Windows. They are about 40 and 60 lines of code respectively and should rarely need to be changed.

[4]`https://docs.gradle.org/current/userguide/gradle_wrapper.html`

[5]`https://maven.apache.org/wrapper/index.html`

8.6.2 Distribution

The wrapper needs a defined location to download the SAMT CLI from. We decided to use GitHub releases[6] for this purpose. Whenever a release is created, the CI pipeline automatically builds and attaches the necessary files to the release. To allow users to easily upgrade to a newer SAMT version, they can simply run the `./samtw wrapper --version <version>` command. Alternatively, the `./samtw wrapper` command is used to automatically fetch the latest stable version from GitHub.

All resource locations are configurable, so users can choose where to download the CLI from if they are behind a corporate proxy. If an organization uses SAMT internally, they can also host the CLI on their own servers and configure the wrapper to download it from there.

8.6.3 SAMT Template

To get started with the `SAMT Wrapper`, an initial set of the start script files is required. Based on personal experience, we decided to create a template repository[7], which contains a basic set of files to get started with SAMT. The template repository contains the following set of starter files:

```
.
├── .samt
│   ├── .gitignore ........................................ Ignore SAMT binaries
│   └── samt-wrapper.properties ................. SAMT version & download url
├── .samtrc.yaml .................................... SAMT style configuration
├── README.md
├── samt.yaml ...................................... SAMT project configuration
├── samtw ................................................... Linux & MacOS
├── samtw.bat ................................................... Windows
└── src
    ├── greeter.samt ........................................ Example model
    ├── greeter-provider.samt
    └── greeter-consumer.samt
```

A user can click the *Use this template* button within GitHub to not only fork this template, but also run a cleanup script. This script will remove all unnecessary files and replace the SAMT version in the `samt-wrapper.properties` file with the latest stable version. Using this approach allows users to get started with SAMT in a matter of seconds without having to manually edit a version in a "Getting Started" guide.

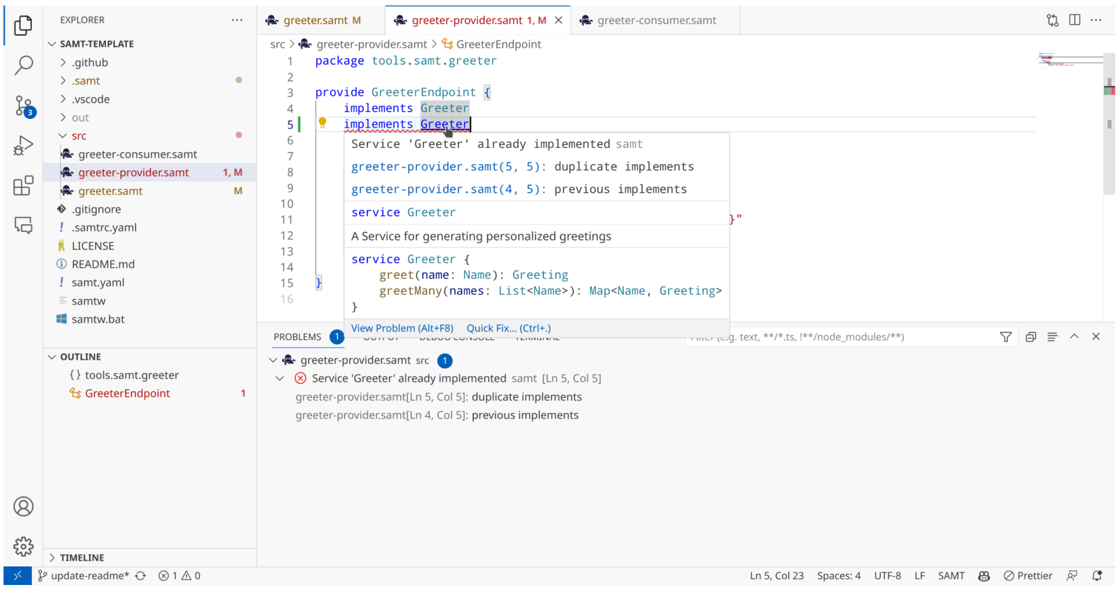
When a user runs any `./samtw` command, the wrapper will download the SAMT CLI and place it in the `.samt` directory. This will create the following temporary files:

[6] https://github.com/samtkit/core/releases
[7] https://github.com/samtkit/template

```
.samt
└── wrapper
    ├── version.txt .......... Installed version, used to prevent duplicate download
    └── cli-shadow ........................................ Extracted SAMT files
        ├── bin
        │   ├── cli ......................................... Linux & MacOS
        │   └── cli.bat ..................................... Linux & MacOS
        └── lib
            └── samt-cli.jar ................................ SAMT CLI
```

8.7 SAMT Visual Studio Code Extension

As decided in Section 7.3.3, we implemented a Visual Studio Code extension for SAMT which has been published on the official marketplace.[8] Figure 8.9 shows the extension in use with syntax highlighting, hover information, diagnostics, the document outline and a preview from the Goto Definition feature.



```
package tools.samt.greeter

provide GreeterEndpoint {
    implements Greeter
    implements Greeter
```

```
Service 'Greeter' already implemented samt
greeter-provider.samt(5, 5): duplicate implements
greeter-provider.samt(4, 5): previous implements
service Greeter
A Service for generating personalized greetings
service Greeter {
    greet(name: Name): Greeting
    greetMany(names: List<Name>): Map<Name, Greeting>
}
View Problem (Alt+F8)    Quick Fix... (Ctrl+.)
```

```
greeter-provider.samt src 1
  Service 'Greeter' already implemented samt [Ln 5, Col 5]
    greeter-provider.samt[Ln 5, Col 5]: duplicate implements
    greeter-provider.samt[Ln 4, Col 5]: previous implements
```

Figure 8.9: Screenshot of the SAMT Visual Studio Code Extension in use

8.7.1 Syntax Highlighting

An expected feature of a language extension is syntax highlighting to help the developer discern different elements of the language. The syntax highlighting is implemented using the TextMate grammar[9] format. This format is supported not only by Visual Studio

[8] https://marketplace.visualstudio.com/items?itemName=samt.samt
[9] https://macromates.com/manual/en/language_grammars

Code, but also by many other editors and IDEs, meaning that the TextMate grammar could be repackaged in an extension for other tools or included manually by a developer.

8.7.2 Snippets

Visual Studio Code snippets allow the developer to reduce typing by providing a quick way to insert a piece of code, such as a record declaration, with predefined customization points (such as its name) that can be tabbed through. There are snippets available for records, enums, type-aliases, services, providers and consumers.

8.7.3 Task Provider

The task provider allows the developer to run the SAMT compiler and code generation through Visual Studio Code's "Run Build Task" command. If the SAMT Wrapper, documented in Section 8.6, is located in the opened folder, the task provider will automatically use it to run the SAMT compiler.

8.7.4 Language Server

VS Code supports LSP, which allows implementing IDE features in a separate process. The main benefit of a separate process is that we are not limited to the Node.js runtime that Visual Studio Code uses and can instead use the JVM, where the SAMT compiler library is available. An additional benefit is that the language server could be used in other editors and IDEs that support the LSP. To minimize manual setup the extension attempts to automatically detect a Java installation on the system, which can also be overridden in the extension's settings. The language server itself is then automatically downloaded from GitHub and started in the background. The following paragraphs describe the various features that the language server supports at this time.

Diagnostics The language server forwards all diagnostics from the SAMT compiler to Visual Studio Code. These diagnostics are the same as the ones shown in the SAMT CLI, but are updated in real time as the user types. Diagnostics from the code generation step, like an invalid HTTP configuration, are not displayed.

Document Synchronization The language server supports document synchronization, which allows it to keep track of changes to the document and update its internal model accordingly. Up-to-date diagnostics can be displayed to the user as they are typing, because Visual Studio Code sends changes even before a file is saved to disk.

File Watching Changes might not only be made through the editor, but also through other means such as a version control system. To keep up with such changes, the language server registers a file watcher for all SAMT files in the workspace.

Semantic Highlighting While syntax highlighting is handled mainly through the TextMate grammar, its regex-based approach lacks the context required to properly highlight type system dependent elements of the language. For this purpose, the language server supports so called Semantic Tokens requests. The IDE uses these to query the language server for certain context-sensitive labels to apply to each syntax token. Depending on the chosen theme, a record identifier can be highlighted differently than an enum identifier.

Goto Definition & Find References To facilitate navigating around a complex data model, it is essential to be able to quickly jump between related elements. To go from a type usage to its definition, we implemented the Goto Definition request. Users can simply Ctrl+Click on a type to jump to the definition of that symbol.

Conversely, users can find all references to a symbol by Ctrl+Clicking on its definition. For this purpose, the language server supports Find References requests, which use the same code for the lookup as the Goto Definition implementation.

Hover When hovering over a symbol, the user expects to see a tooltip with additional information about the symbol. For this purpose, the language server supports Hover requests, which display a code snippet of the declaration and a description if one was provided with the `@Description` annotation. Visual Studio Code also applies syntax highlighting to the code snippet, using the TextMate grammar described in Section 8.7.1.

Document Symbols To give an overview of a file's contents and to allow quick navigation, the language server supports Document Symbols requests. Visual Studio Code then displays the symbols in the outline view and in the breadcrumb view at the top of the editor as seen in Figure 8.9.

8.7.5 SAMT Configuration

The SAMT configuration files introduced in Section 7.4.1 are written in YAML and are not part of the SAMT language. However, Visual Studio Code has support for YAML files thanks to the popular YAML extension.[10] To provide a better user experience when editing YAML configuration files, a JSON Schema[11] can be used to provide auto-completion and validation. In order to make the user experience seamless, this schema can be embedded in the JSON Schema Store[12], which is automatically referenced by the YAML extension. As SAMT has two YAML configuration files, a matching JSON Schema was created for both the SAMT configuration file[13] and the SAMT style configuration file.[14]

[10] https://marketplace.visualstudio.com/items?itemName=redhat.vscode-yaml
[11] https://json-schema.org/
[12] https://www.schemastore.org/
[13] https://json.schemastore.org/samt.json
[14] https://json.schemastore.org/samtrc.json

8.8 Requirements Coverage

This section givens an overview of the status of the requirements.

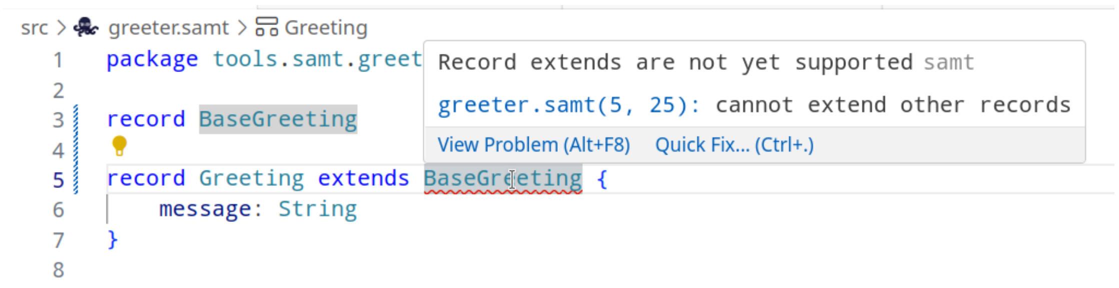
8.8.1 Functional Requirements

In Chapter 2 we defined a set of functional requirements for SAMT. Because it seemed unlikely to meet all functional requirements in the given time frame, we set one of three priorities "must have", "should have" and "could have". In the end, all "must have" requirements were implemented, as well as a majority of "should have" requirements and some "could have" requirements.

Comments on Notable Requirements

This section gives an overview of notable requirements that were not met or only partially met.

Inheritance The inheritance feature was implemented only in the parser, but is not supported in the type system. Users are informed about this limitation by an error message, as shown in Figure 8.10.



```
package tools.samt.greet

record BaseGreeting

record Greeting extends BaseGreeting {
    message: String
}

```

```
Record extends are not yet supported samt
greeter.samt(5, 25): cannot extend other records
```

Figure 8.10: Error message when using inheritance

Faults Faults are ways of modeling what kind of errors can occur during an operation. The fault feature was implemented in the parser and operations in the type system have their faults set, but some semantic validation is missing. In addition, there was supposed to be a `Fault` type from which to extend from, but this is currently not possible due to the lack of an inheritance feature. Because of this, faults were omitted from the Public API explained in Section 8.5.1, meaning that generators do not know about them.

File Separation To ensure separation of concerns, the existing Xtext based solution provides multiple DSLs for different purposes, ensuring that they are kept in different files. SAMT, on the other hand, is a single language, which means that everything could be written in a single file. We decided to implement this feature as a compiler warning, but only warn about models of a certain size to allow simple examples and unit tests to keep their code in a single file. Zürcher Kantonalbank would prefer to emit the warning unconditionally, which could easily be achieved by changing a single line of code. As a team, we feel that this would be too restrictive and raise the barrier of entry for new users. When configurable linting rules are implemented in the future, the threshold will be made configurable on a per-project basis.

IDE Features The more advanced IDE features, namely context-aware autocompletion, formatting and refactoring were not implemented. These features are supported by LSP, which means that they can be implemented in the future.

In contrast, additional features that were not specified in the requirements were implemented, such as Hover and Document Symbols. Thanks to this solid foundation, implementing the missing IDE features should be relatively easy. For example, refactorings would be straightforward to implement, as finding all references to a symbol is already implemented for the Find References feature.

Full Overview

Table 8.2 gives an overview of the status of all functional requirements.

Requirement	**Priority**	**State**
FR-LF-PRIMITIVETYPES	Must	Done
FR-LF-RECORD	Must	Done
FR-LF-ENUM	Must	Done
FR-LF-NULLABILITY	Must	Done
FR-LF-TYPECONSTRAINTS	Must	Done
FR-LF-LIST	Must	Done
FR-LF-ALIAS	Must	Done
FR-LF-SERVICE	Must	Done
FR-LF-OPERATION	Must	Done
FR-LF-PROVIDER	Must	Done
FR-LF-CONSUMER	Must	Done
FR-LF-PACKAGE	Must	Done
FR-LF-SEPARATIONOFFILES	Must	Done
FR-T-PARSERLIB	Must	Done
FR-T-CODEGENLIB	Must	Done
FR-T-IDESYNTAX	Must	Done
FR-LF-DOCUMENTATION	Should	Done
FR-LF-ANNOTATION	Should	Done
FR-T-IDEWARNERROR	Should	Done
FR-LF-INHERITANCE	Should	Mixed
FR-T-IDEHINT	Should	Not Done
FR-T-IDEFORMAT	Should	Not Done
FR-LF-MAP	Could	Done
FR-LF-ONEWAYOPERATION	Could	Done
FR-LF-FAULT	Could	Mixed
FR-LF-ASYNCOPERATION	Could	Mixed
FR-LF-MULTIPLEINHERITANCE	Could	Not Done
FR-LF-CONSTANTS	Could	Not Done
FR-LF-MODULARCONSTRAINTS	Could	Not Done
FR-LF-RECORDCONSTRAINTS	Could	Not Done
FR-LF-DEPENDENCIES	Could	Not Done
FR-T-IDEREFACTOR	Could	Not Done

Table 8.2: Overview of all functional requirements with their priority and state

8.8.2 Non-Functional Requirements

This section contains a review of the non-functional requirements defined in Chapter 3.

Usability

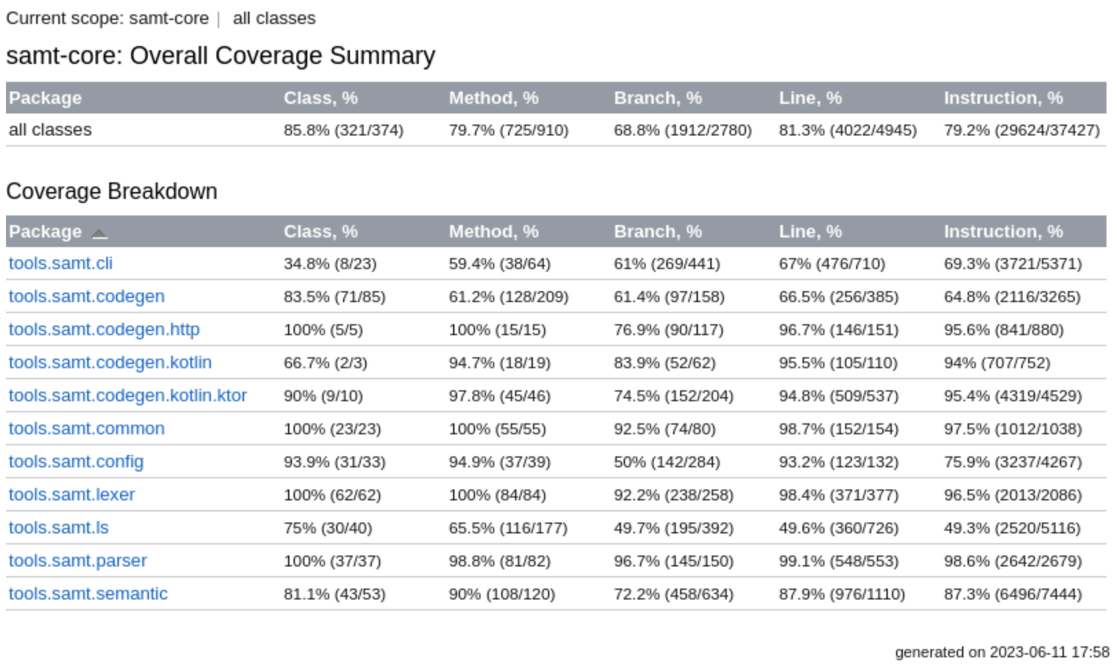
Each stakeholder archetype, namely the Service Owner, System Integrator and Domain Expert, has a specific usability requirement defined for that role. For verification, user testing was performed with four different developers at Zürcher Kantonalbank, who were not familiar with the project. The results of this testing are discussed in Appendix C.6. Unfortunately, all four developers would fit the Service Owner role, which means that the NFRs for the System Integrator and Domain Expert stakeholder were not tested.

Compatibility

To verify operating system compatibility, a Continuous Integration pipeline was configured to run on the latest versions of Windows, Ubuntu and macOS. As SAMT Core runs on Java 17, it should be compatible with any operating system that has an implementation of Java 17 available, but this was not explicitly verified. However, basic functionality has been tested on all three operating systems, as evidenced by the data in Figure 8.12.

Maintainability

For maintainability, a line coverage target of 80% was set, which was narrowly exceeded in the SAMT Core project as shown in Figure 8.11. In the SAMT Visual Studio Code Extension, a few tests were written to protect against certain oversights in the TextMate grammar, but the TypeScript code was left untested. However, the most complex IDE functionality is implemented in the Language Server, which is a part of the well-tested SAMT Core project. To put the size of the two codebases into perspective, the SAMT Core project has close to 5000 lines of code, while the Visual Studio Code extension has less than 600 lines of code.


Current scope: samt-core | all classes

samt-core: Overall Coverage Summary

Package	Class, %	Method, %	Branch, %	Line, %	Instruction, %
all classes	85.8% (321/374)	79.7% (725/910)	68.8% (1912/2780)	81.3% (4022/4945)	79.2% (29624/37427)

Coverage Breakdown

Package	Class, %	Method, %	Branch, %	Line, %	Instruction, %
tools.samt.cli	34.8% (8/23)	59.4% (38/64)	61% (269/441)	67% (476/710)	69.3% (3721/5371)
tools.samt.codegen	83.5% (71/85)	61.2% (128/209)	61.4% (97/158)	66.5% (256/385)	64.8% (2116/3265)
tools.samt.codegen.http	100% (5/5)	100% (15/15)	76.9% (90/117)	96.7% (146/151)	95.6% (841/880)
tools.samt.codegen.kotlin	66.7% (2/3)	94.7% (18/19)	83.9% (52/62)	95.5% (105/110)	94% (707/752)
tools.samt.codegen.kotlin.ktor	90% (9/10)	97.8% (45/46)	74.5% (152/204)	94.8% (509/537)	95.4% (4319/4529)
tools.samt.common	100% (23/23)	100% (55/55)	92.5% (74/80)	98.7% (152/154)	97.5% (1012/1038)
tools.samt.config	93.9% (31/33)	94.9% (37/39)	50% (142/284)	93.2% (123/132)	75.9% (3237/4267)
tools.samt.lexer	100% (62/62)	100% (84/84)	92.2% (238/258)	98.4% (371/377)	96.5% (2013/2086)
tools.samt.ls	75% (30/40)	65.5% (116/177)	49.7% (195/392)	49.6% (360/726)	49.3% (2520/5116)
tools.samt.parser	100% (37/37)	98.8% (81/82)	96.7% (145/150)	99.1% (548/553)	98.6% (2642/2679)
tools.samt.semantic	81.1% (43/53)	90% (108/120)	72.2% (458/634)	87.9% (976/1110)	87.3% (6496/7444)

generated on 2023-06-11 17:58

Figure 8.11: Code coverage report of the SAMT Core project

JetBrains Qodana was chosen as our static code analysis tool. Qodana is configured as part of our GitHub Actions CI pipeline. All pull requests are required to pass the Qodana analysis before merging, which ensures that the code quality does not degrade over time. In conclusion, we consider the maintainability requirements to be fulfilled.

Security

There are two main aspects of security for SAMT, namely the security of SAMT and its dependencies, and the security of the generated code. To reduce the risk of supply chain vulnerabilities, the SAMT libraries do not have any dependencies besides the Kotlin Standard Library. The CLI has a few dependencies for command line parsing and output formatting, while the Language Server uses the LSP4J library to handle protocol concerns. Gradle dependencies are submitted to the GitHub dependency submission API, which checks them for known vulnerabilities.

In the SAMT Visual Studio Code Extension, the GitHub Dependabot[15] checks all dependencies for known vulnerabilities. A sample GitHub repository with generated code was created[16] and equipped with GitHub code scanning to check for vulnerabilities in the generated code. The code scanning did not find any vulnerabilities in the generated code. Of course, this does not mean that there are no vulnerabilities, as such tools can only be used to assert the presence of vulnerabilities, not their absence.

[15] https://docs.github.com/en/code-security/dependabot
[16] https://github.com/samtkit/ktor-demo

Given these results, we consider all security requirements to be fulfilled.

Extensibility

An extensibility guide was written and published on the SAMT GitHub wiki.[17] Due to time constraints, we were unable to verify the measure if an experienced developer could implement a new code generator in a work week. The code generator we built was finished in a couple of days. This however is not a fair comparison, as we were already intimately familiar with the codebase.

Performance

To assess the performance of SAMT, benchmark files with the size and complexity specified in the performance NFR defined in Section 3.6 were created.[18] The benchmark consists of running the SAMT compilation 100 times with the help of the benchmarking tool Hyperfine.[19] The benchmark was run on three different machines:

- MacBook Pro with Apple M1 Max running macOS 13.4
- Desktop PC with AMD Ryzen 7 7700X running Windows 11
- Desktop PC with Intel Core i7 6700K running Linux 6.3

The results of the benchmark are shown in Figure 8.12.

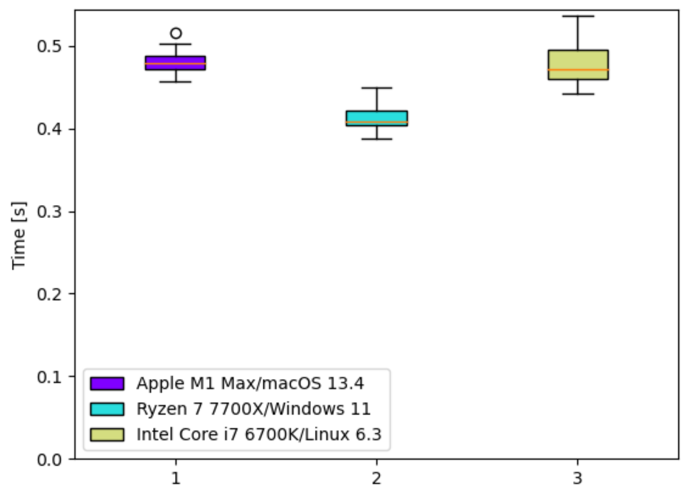


Figure 8.12: Whisker chart of the performance benchmark

[17] https://github.com/samtkit/core/wiki
[18] https://github.com/samtkit/core/tree/main/specification/benchmarks/todo-service
[19] https://github.com/sharkdp/hyperfine

The maximum run time was under 0.6 seconds, which is well below the 5 seconds "Outstanding" landing zone specified in the NFR. In addition, variance between the slowest and the fastest run on each machine was less than 0.1 seconds, indicating a high level of consistency. Overall, we consider the performance requirements to be fulfilled.

Full Overview

Table 8.3 gives an overview of the status of all non-functional requirements.

Requirement	**Priority**	**State**
NFR-C-OS	High	Done
NFR-M-TESTS	High	Done
NFR-P-COMPILATION	High	Done
NFR-S-CODEGEN	High	Done
NFR-S-VULNERABILITIES	High	Done
NFR-U-SERVICEOWNER	High	Done
NFR-E-EXTENSIBILITY	High	Mixed
NFR-U-SYSTEMINTEGRATOR	High	Mixed
NFR-U-DOMAINEXPERT	High	Mixed
NFR-M-QUALITY	Medium	Done

Table 8.3: Overview of all non-functional requirements with their priority and state

Chapter 9

Summary and Outlook

This chapter provides a comprehensive summary of the results achieved, followed by an outlook on future work.

9.1 Summary

The project started with defining the requirements that set the objectives and scope and will serve as the foundation for the subsequent development phases. To address these requirements, we created a language specification that outlined the syntax, structure, and key features of SAMT. We iteratively refined this specification to ensure its effectiveness in addressing the requirements. Many trade-offs were considered and evaluated, which resulted in a language design that provides a balance between simplicity and expressiveness.

Furthermore, we developed an architecture that provides a robust and flexible framework to support the language implementation and subsequent development phases. Based on the language specification and architecture, a compiler was implemented from scratch, including a lexer, a parser, and a semantic analyzer. The decision to not rely on existing tools and libraries was challenged several times during the development process, however, based on the outcome of the evaluations performed, this choice was eventually deemed justified.

To improve the developer experience and increase their productivity, we implemented a Visual Studio Code extension that was published to the Visual Studio Marketplace. This extension provides several features to help developers model in SAMT, such as syntax highlighting and snippets. In addition, we implemented a language server to enable advanced code editing features such as code navigation and real-time error checking. A dedicated language server simplifies development, separates compiler and editor concerns, and enables seamless integration with other IDEs that support LSP.

A sample Ktor generator demonstrates the extensibility and versatility of the internal

SAMT model for one of the most complex transport protocols, HTTP. The implementation of this generator further highlighted the need for a refinement of the generator API and architecture, in order to achieve widespread adoption.

We conducted usability testing to evaluate the effectiveness and usability of the language, compiler, and associated tools. Valuable feedback helped to identify areas for improvement and fine-tuning, improving the overall quality and usability of the tools developed.

To provide the required extensibility to Zürcher Kantonalbank, and potentially to the open-source community, we published the SAMT libraries to Maven Central. These libraries provide access to the functionality offered by SAMT and allow developers to extend or integrate the SAMT compiler into their own projects. In addition, we created extensive documentation and implemented a sample plugin in Java to make contributions as easy as possible.

Finally, all deliverables of this thesis were produced and published under an open-source license wherever required by the project guidelines. SAMT was also presented at a developer conference within Zürcher Kantonalbank, where it received positive feedback and interest from the audience.

9.2 Outlook

The future of SAMT is influenced by the interest of the open-source community and the needs of Zürcher Kantonalbank. It requires extensive further development to be applicable in a large organization. Zürcher Kantonalbank will evaluate the use of SAMT, which will determine whether they will either continue the development of SAMT, create a fork for their specific needs, or use another solution altogether. Regardless of the future of SAMT, we hope that the results from this thesis will be valuable for Zürcher Kantonalbank.

Independent of its actual use in Zürcher Kantonalbank or else, there are several feature enhancements that could be developed in the future. They are briefly explained in the following paragraphs:

Fault Handling Implement fault handling for operations to handle errors and exceptions that may occur during their execution. This is similar to explicit exception handling in Java, but with a focus on the communication of errors between services. User feedback from usability testing has shown that this is a desired feature, however, the details of this implementation are still unclear (e.g. some users prefer the terminology exceptions instead of faults).

Security Support a security configuration within SAMT models, for example to limit a certain operation to a subset of users.

API Versioning Add API versioning for models written in SAMT to alleviate the need to maintain multiple versions of the same API.

Inheritance Support Introduce support for inheritance, allowing to create hierarchies to model complex data structures.

Improved Parser Error Recovery Improve the parser error recovery mechanism to provide clearer error messages and better strategies for recovering from syntax errors.

Expanded Built-in Constraints Expand the set of built-in constraints for standard library types to meet specific requirements such as number precision, date time zones, and string encoding.

Finalized Generator Architecture Finalize the generator API and architecture, resolving any outstanding issues identified during development to enhance code generation. Possible solutions are discussed in Section 7.4.5.

Plugin Mechanisms Implement plugin installation and loading mechanisms to allow modular extensions and integration with third-party libraries. The interfaces for these mechanisms have already been defined, but the implementation is still missing. Particularly, the distribution of plugins is a challenge, as it requires a central repository for plugins.

Model Dependencies Establish dependencies between models, enabling the referencing of types from other models and facilitating the creation of more complex systems. The main challenge here is to define a mechanism for resolving these dependencies, given that the compiler needs to know where to find the referenced models. Possible solutions are discussed in Section 7.4.1.

Extend Language Server By implementing more interfaces provided by the Language Server Protocol (LSP), such as code completion and code formatting, the development experience can be greatly improved.

API Portal Provide an API portal for SAMT to centralize documentation and resources related to published APIs. This is a unique opportunity for SAMT, as the API portal can be automatically generated from the SAMT model and provide a single source of truth for a broad range of APIs. Whereas other API portals are often limited to a single technology, the SAMT API portal can provide a unified view of all APIs within an organization.

In case SAMT is adopted for production use within Zürcher Kantonalbank, the following steps are needed specifically for a large organization:

- Implement SAMT Plugins to support the required transports and frameworks specific to the organization's needs.
- Analyze the possibility of developing an IntelliJ IDEA Plugin as a complementary tool to the SAMT Visual Studio Code Extension.

- Roll out the SAMT Visual Studio Code Extension to all developers, ensuring widespread adoption and usage within the organization.
- Distribute the required SAMT binaries to a private repository, ensuring secure access and distribution of the associated tools.
- Create a custom SAMT Template for internal use, enabling rapid development of projects based on predefined patterns and best practices.

In conclusion, this thesis has established a foundation for further development and utilization of SAMT, both within Zürcher Kantonalbank and potentially by the broader developer community. The implemented requirements demonstrate the viability and potential impact of the proposed language and associated tools. With the outlined future steps, SAMT could evolve into a powerful and widely adopted solution for efficient model-based development workflows.

Appendices

Appendix A

Task Assignment

This chapter contains the task assignment as originally written by the thesis advisor.

Aufgabenstellung Bachelorarbeit
Pascal Honegger, Marcel Joss, Leonard Schütz

Neuentwicklung einer erweiterbaren Schnittstellen-Modellierungssprache

1. Auftraggeber und Betreuer

Diese Bachelorarbeit wird in Zusammenarbeit mit der Firma Zürcher Kantonalbank (ZKB) durchgeführt.

Ansprechpartner (extern):

André Lehner, Projektleiter seitens ZKB, andre.lehner@zkb.ch

Betreuer (OST):

Prof. Dr. Olaf Zimmermann, OST Dept. I, Institut für Software, olaf.zimmermann@ost.ch

2. Ausgangslage

Der Industriepartner verwendet eine intern entwickelte Modellierungssprache für die Definition von Schnittstellen zwischen Applikationen. Ein Kernaspekt dieser Modellierungssprache ist die Plattform- und Technologieunabhängigkeit, welche bestehende Tools via OpenAPI nicht bieten. Die technische Plattform für die Sprach- und Werkzeugentwicklung, das auf Eclipse basierende DSL-Framework Xtext, stellt aufgrund seiner Komplexität und Zukunftsunsicherheit allerdings ein Risiko für die Weiterentwicklung und Pflege dieser Schnittstellen-Modellierungssprache dar.

3. Ziele der Arbeit und Liefergegenstände

In der Bachelorarbeit geht es um die Erarbeitung und Entwicklung einer neuen Modellierungssprache und dazu passendes Tooling. Ziel ist es, die bestehende Abhängigkeit von Xtext mit einer Neuentwicklung zu beseitigen. Die neu zu erstellende Modellierungssprache muss die bestehende funktional ersetzen, kann aber technisch frei entwickelt werden. Zudem muss eine entsprechende Sprachunterstützung in einem etablierten Editor (Bsp. IntelliJ oder Visual Studio Code) sichergestellt werden. Beispiele der existierenden Modellierungssprache und Anforderungen an das erwartete Produkt der Arbeit sind vorhanden.

Liefergegenstände. Die zentralen Deliverables der Arbeit sind:

- Sprachspezifikation (Grammatik) sowie Source Code und zugehörige Dokumentation (z.B. Tutorials, Beispiele, Starthilfe für die Nutzung, API- bzw. Programmierreferenz)
- IDE-Plugin
- BA-Bericht

Kritische Erfolgsfaktoren. Für die Bewertung der Arbeit sind folgende Kriterien besonders relevant:

- Angemessener, priorisierter Funktionsumfang von Sprache und Werkzeugen (Bsp. muss, kann, soll)
- Benutzbarkeit und Lernaufwand
- Wartbarkeit und Betreibbarkeit der Lösung, insbesondere Erweiterbarkeit (zukünftige Integrierbarkeit, z.B. Generatoren) und Zukunftssicherheit
- Software Engineering-Hygienefaktoren wie Versionsverwaltung, automatisierte Builds und Tests sowie zielgruppengerechte Dokumentation

4. Unterstützung

Die erwartete und effektiv erhaltene Unterstützung wird durch die Studierenden protokolliert.

5. Zur Durchführung

Mit dem Betreuer finden in der Regel wöchentlich Besprechungen statt (Treffen an der OST, beim Industriepartner oder Videokonferenz). Zusätzliche Besprechungen sind nach Bedarf zu veranlassen.

Alle Besprechungen, bei denen eine Vorbereitung durch den Betreuer nötig ist, sind von den Studierenden mit einer Traktandenliste vorzubereiten. Beschlüsse sind in einem Protokoll zu dokumentieren.

Für die Durchführung der Arbeit ist ein Projektplan zu erstellen. Dabei ist auf einen kontinuierlichen und sichtbaren Arbeitsfortschritt zu achten. Arbeitszeiten sind zu dokumentieren. Sofern nicht in dieser Aufgabenstellung vorgeben, sind die Studierenden für die Auswahl und korrekte Anwendung Ihrer Hilfsmittel (also Werkzeuge, Libraries, Frameworks, SaaS-Angebote, etc.) selbst verantwortlich. Auf Wunsch kann eine an der OST gehostete virtuelle Maschine zur Verfügung gestellt werden.

Die Spezifikation der Anforderungen geschieht durch die Studierenden in Absprache mit dem Betreuer. Bei Disputen entscheidet der Betreuer in Rücksprache mit den Studierenden über die definitiv für die Bachelorarbeit relevanten Anforderungen.

Vorstudie, Anforderungsdokumentation und Architekturdokumentation sollten im Laufe des Projektes mittels Milestone mit dem Auftraggeber und dem Betreuer in einem stabilen Zustand abgenommen werden. Zu den abgegebenen Arbeitsresultaten wird ein vorläufiges Feedback abgegeben. Eine definitive Beurteilung erfolgt auf Grund der am Abgabetermin abgelieferten Dokumentation.

Die Rechte an den Ergebnissen der Bachelorarbeit werden in einer separaten Vereinbarung definiert (Bericht öffentlich, Implementierung open source; kein Non-Disclosure Agreement erforderlich).

6. Dokumentation

Über diese Arbeit ist eine Dokumentation gemäss den Richtlinien des Studiengangs Informatik zu verfassen. Die zu erstellenden Dokumente sind im Projektplan festzuhalten. Alle Dokumente sind nachzuführen, d.h. sie sollten den Stand der Arbeit bei der Abgabe in konsistenter Form dokumentieren.

Bei der Projektdokumentation und deren Abgabe sind die allgemeinen Informationen zu Studien- und Bachelorarbeiten sowie insbesondere die "Orientierungshilfe für die Dokumentation einer Studien- oder Bachelorarbeit" und der "Leitfaden für Studien- und Bachelorarbeiten" des Studiengangs zu beachten.

7. Termine

Die Termine wurden vom Sekretariat des Studiengangs Informatik in MS Teams veröffentlicht im Dokument "Bachelorarbeiten: Termine Frühjahrssemester 2023". Sie sollen in einem Sitzungsprotokoll dokumentiert werden.

Erste Semesterwoche, beginnend mit dem 20.02.23	Beginn der Bachelorarbeit, Ausgabe der Aufgabenstellung durch die Betreuer
16.06.2023	Hochladen aller verlangten Dokumente auf https://avt.i.ost.ch/ Abgabe des Berichts an den Betreuer/die Betreuerin bis 17.00 Uhr.
bis 31.08.2023	Mündliche BA-Prüfung

8. Beurteilung

Eine erfolgreiche Bachelorarbeit zählt 12 ECTS-Punkte pro Studierenden. Für 1 ECTS-Punkt ist eine Arbeitsleistung von 30 Stunden budgetiert.

Es gelten die Bestimmungen des Studiengangs Informatik zur Durchführung und Bewertung von Bachelorarbeiten (siehe "Leitfaden für Studien- und Bachelorarbeiten").

Rapperswil, 22. 02. 2023

Prof. Dr. Olaf Zimmermann
Institut für Software
OST

Appendix B

Development Instructions

This chapter contains instructions for developing and maintaining SAMT.

B.1 SAMT Core

The `samtkit/core` repository[1] hosts the code for the SAMT libraries, the CLI, and the language server. Only members of the Simple API Modeling Toolkit GitHub organization[2] have write access to this repository. The architecture of the codebase itself is explained in more detail in Section 7.4.

B.1.1 Development IDE

So far all development has been done in IntelliJ IDEA, so it is recommended to use this IDE for development. Using another IDE is not recommended because JetBrains Qodana is used as part of the CI pipeline. Qodana reports most of the same issues as IntelliJ IDEA, so using another IDE may cause surprises when running the CI pipeline. The Community Edition of IntelliJ IDEA is sufficient for development.

B.1.2 Build System

SAMT Core is a Gradle project which uses the Gradle wrapper. Because of this, a local Gradle installation is not necessary to build the project, instead you can run the Gradle wrapper script (`./gradlew.bat` on Windows and `./gradlew` on other platforms) to execute tasks. If you prefer to use a GUI, you can also choose tasks from a list in IntelliJ IDEA. The most commonly used build tasks are the following:

`assemble` Assembles the JAR files, which is useful if you need to run the CLI or language server outside of an IDE

[1] `https://github.com/samtkit/core`
[2] `https://github.com/samtkit`

`check` Runs tests

`koverVerify` Runs tests and verifies that code coverage goals are met

`koverHtmlReport` Runs tests and generates a code coverage report in HTML format, which is useful for finding untested code

`publishToMavenLocal` Publishes the SAMT libraries to the local Maven repository, which is useful to see what would be published in a release

The project uses Gradle toolchains[3], which will automatically find a Java 17 installation on the system or download it if necessary. You still need to have a Gradle compatible Java installation on your system for this to work, but you do not need to install Java 17 specifically for this project.

B.1.3 Continuous Integration

The repository uses GitHub Actions for continuous integration. The CI pipeline runs on every push to the `main` branch and on every pull request. The following workflows exist:

Build and Test Builds the project, runs tests and verifies that code coverage goals are met. Runs on Ubuntu, Windows and macOS.

Qodana Runs static analysis of the Kotlin code using JetBrains Qodana.

Security Scanning Performs GitHub CodeQL analysis to find security vulnerabilities and quality issues. This task is also run weekly on the `main` branch in case of new security issues.

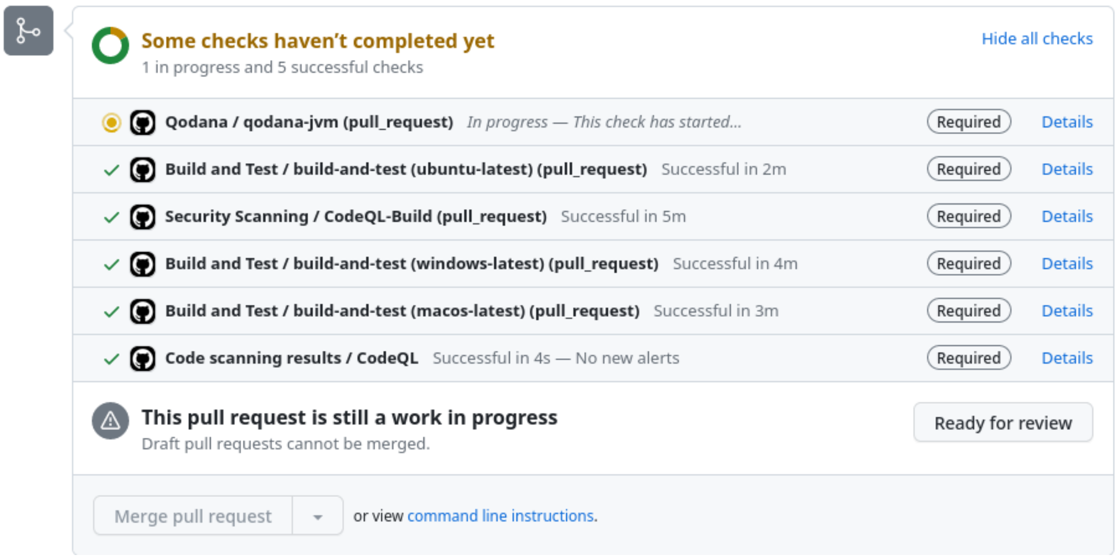
Gradle Dependency Submission Submits the Gradle dependencies to GitHub, which allows viewing them in the GitHub UI and getting notified about security issues in dependencies. This task runs only on the main branch because the repository can have only one set of dependencies.

Publish Snapshot Publishes a snapshot version of the SAMT libraries to the Sonatype OSSRH repository. This worflow runs only on the main branch.

Publish Release Publishes a draft release of SAMT on GitHub with the packaged versions of the CLI and language server and publishes the SAMT libraries to the Sonatype OSSRH staging repository. This task runs only after pushing a tag of the form `vX.Y.Z`.

Pull requests can only be merged after a successful run of the CI pipeline and at least one approval of another project member. Figure B.1 shows the CI pipeline in progress on a pull request.

[3] https://docs.gradle.org/current/userguide/toolchains.html


Some checks haven't completed yet
1 in progress and 5 successful checks
Hide all checks

Qodana / qodana-jvm (pull_request)	*In progress — This check has started...*	Required	Details
✓ Build and Test / build-and-test (ubuntu-latest) (pull_request)	Successful in 2m	Required	Details
✓ Security Scanning / CodeQL-Build (pull_request)	Successful in 5m	Required	Details
✓ Build and Test / build-and-test (windows-latest) (pull_request)	Successful in 4m	Required	Details
✓ Build and Test / build-and-test (macos-latest) (pull_request)	Successful in 3m	Required	Details
✓ Code scanning results / CodeQL	Successful in 4s — No new alerts	Required	Details

This pull request is still a work in progress
Draft pull requests cannot be merged.
Ready for review

Merge pull request ▾ or view command line instructions.

Figure B.1: Continuous integration pipeline in GitHub's UI

B.1.4 Releasing

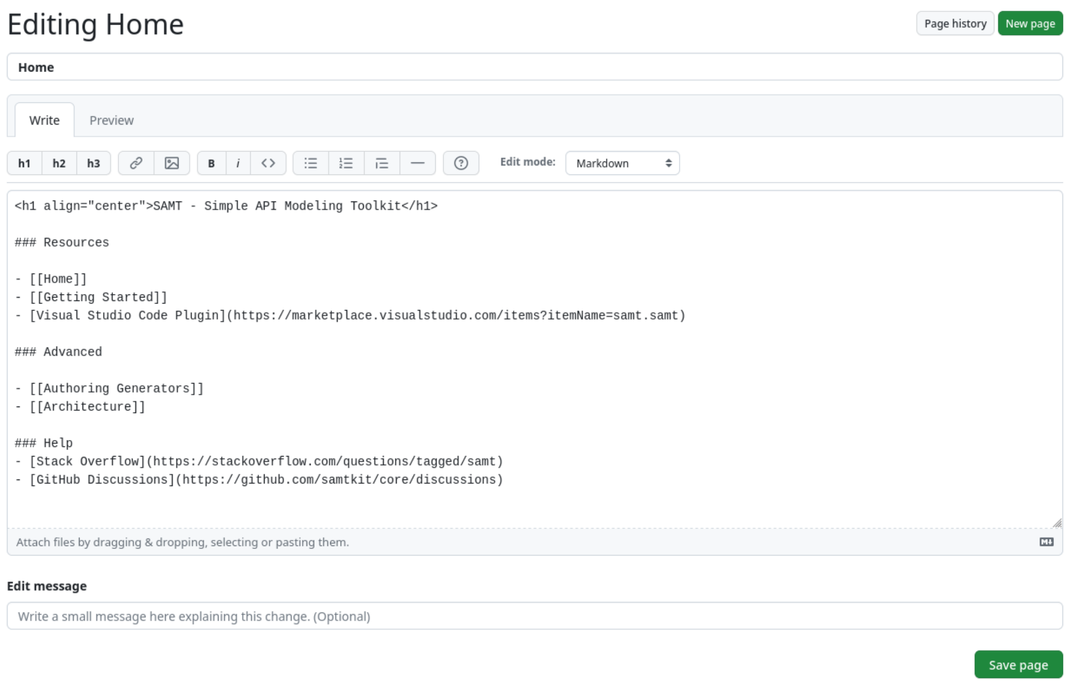
GitHub Because the Publish Release task only creates a draft release on GitHub, the release must be published manually. To do this, open the created draft release on GitHub, edit it, and click the *Publish release* button. By default, the release notes mention all pull requests which have been merged since the last release, but they can also be adjusted manually.

Sonatype OSSRH The Publish Release task only publishes the SAMT libraries to the Sonatype OSSRH staging repository.[4] To release the libraries to Maven Central, they must be promoted from the staging repository to the release repository.

B.1.5 Documentation

The documentation is kept in the wiki of the SAMT Core repository, where it can be edited in a Markdown format with a live preview as shown in Figure B.2.

[4]https://s01.oss.sonatype.org/


Editing Home
Page history
New page
Home
Write
Preview
Edit mode: Markdown
Attach files by dragging & dropping, selecting or pasting them.
Edit message
Write a small message here explaining this change. (Optional)
Save page

```
<h1 align="center">SAMT - Simple API Modeling Toolkit</h1>

### Resources

- [[Home]]
- [[Getting Started]]
- [Visual Studio Code Plugin](https://marketplace.visualstudio.com/items?itemName=samt.samt)

### Advanced

- [[Authoring Generators]]
- [[Architecture]]

### Help
- [Stack Overflow](https://stackoverflow.com/questions/tagged/samt)
- [GitHub Discussions](https://github.com/samtkit/core/discussions)
```

Figure B.2: Homepage of the SAMT wiki in editing view

B.2 SAMT Visual Studio Code Extension

The `samtkit/vscode` repository[5] hosts the code for the SAMT Visual Studio Code Extension. Its implementation is explained in more detail in Section 8.7.

B.2.1 Development IDE

The development of the extension is done with Visual Studio Code itself, because it makes it easy to start a separate instance with the developed extension active. To launch the extension open the *Run and Debug* panel and select the *Run Extension* configuration.

B.2.2 Package Manager

The pnpm[6] package manager is used to manage dependencies and build the extension. It is recommended to enable Corepack[7] on a development machine to automatically use the same version of pnpm as other developers. If you have Node.js installed, simply run `corepack enable` in a terminal. If this fails, you may have to uninstall any existing Yarn or pnpm installations first.

[5] https://github.com/samtkit/vscode
[6] https://pnpm.io/
[7] https://nodejs.org/api/corepack.html

B.2.3 Scripts

The `package.json` file contains scripts which can be run with the `pnpm run` command. The most important ones are the following:

`format` Formats the TypeScript source files using the Prettier[8] code formatter.

`lint` Checks if the files are formatted correctly according to Prettier and runs ESLint[9] to find problems in the code.

`test` Runs the `lint` script and unit tests.

`build` Builds the extension for development.

`package` Packages the extension into a VSIX file, which can be installed locally into Visual Studio Code or manually uploaded to the marketplace.

`publish` Publishes the extension to the marketplace. Requires an Azure DevOps personal access token with the *Publish* permission.

B.2.4 Continuous Integration

Just like the `core` repository, the `vscode` repository uses GitHub Actions for continuous integration and imposes the same rules for merging pull requests. Workflows are the following:

Build and Test Builds the extension and runs the test script. Runs on Ubuntu, Windows and macOS.

Security Scanning Performs GitHub CodeQL analysis to find security vulnerabilities and quality issues. This workflow is also run weekly on the `main` branch in case of new security issues.

B.2.5 Releasing

Prerequisites

First an owner of the SAMT publisher must add you to the publisher, to do that they need your User Id. To find your User Id open the Visual Studio Marketplace[10] and move the mouse cursor over your name at the top of the screen as seen in Figure B.3.

[8] `https://prettier.io`
[9] `https://eslint.org`
[10] `https://marketplace.visualstudio.com`

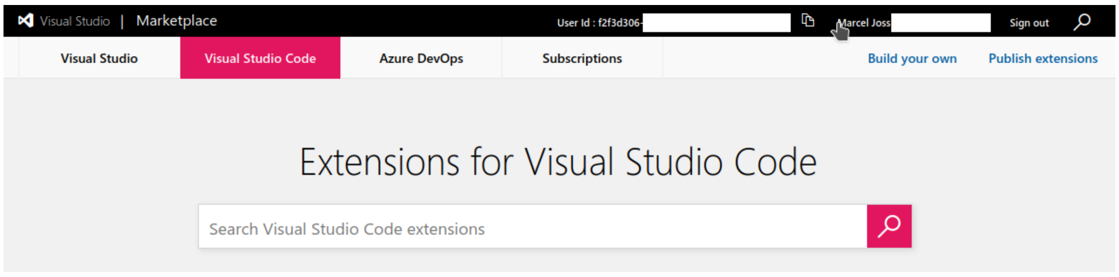

Visual Studio | Marketplace
User Id : f2f3d306[redacted]
Marcel Joss[redacted]
Sign out
Visual Studio
Visual Studio Code
Azure DevOps
Subscriptions
Build your own
Publish extensions
Extensions for Visual Studio Code
Search Visual Studio Code extensions

Figure B.3: User Id in the Visual Studio Marketplace

If you want to publish directly from the command line, you need an Azure DevOps personal access token. To access Azure DevOps, you can click on your name in the Visual Studio Marketplace. If you want to create a token, you need to be a member of an organization, which you can create for free. Note that the actual token is not tied to the organization, you only need an organization because you cannot access the token creation page without one.

On your organization page click on the *User Settings* in the top right corner and then on *Personal access tokens* as shown in Figure B.4.

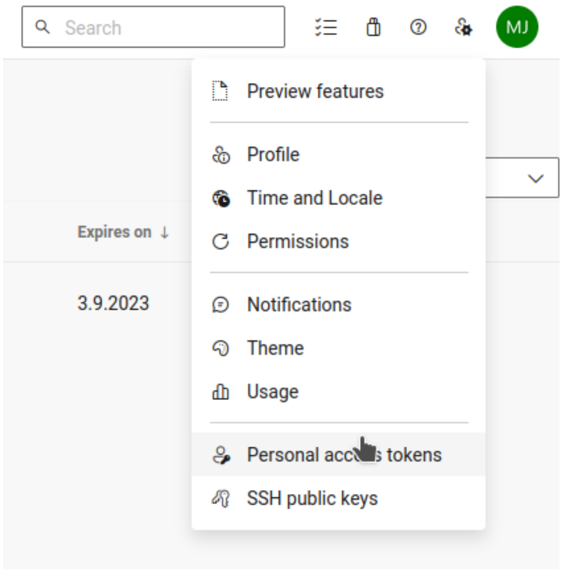
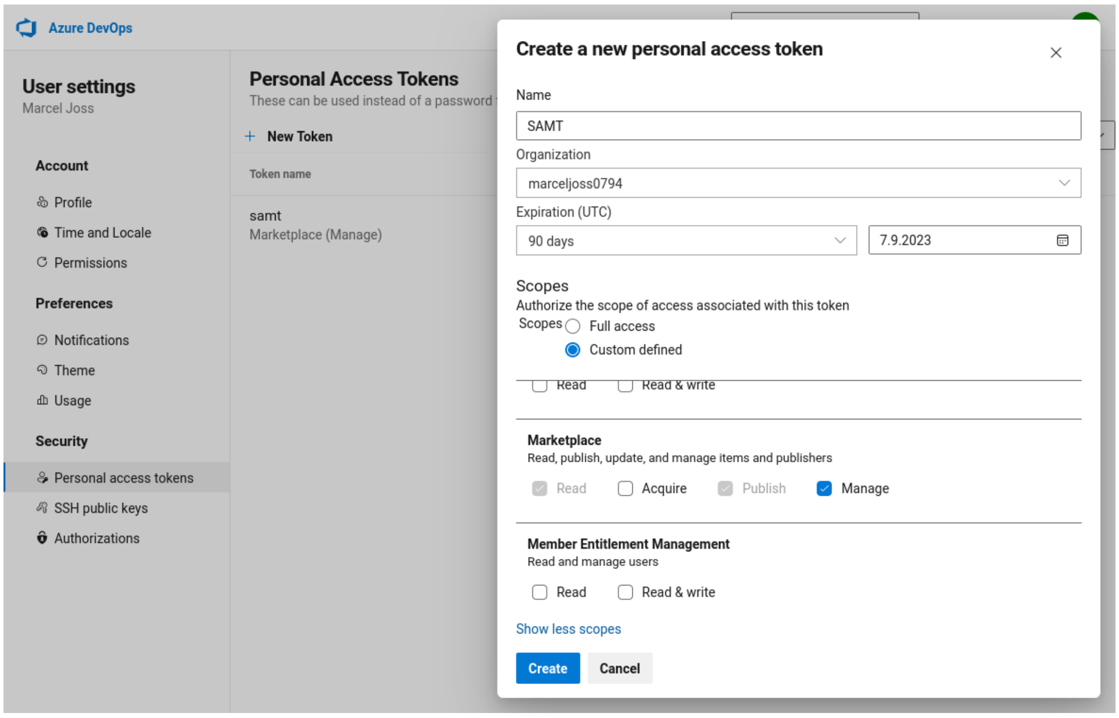


Figure B.4: Azure DevOps user settings

On the personal access tokens page click on *New Token*, under scopes select *Custom defined*, click *Show all scopes*, scroll to *Marketplace* and choose *Manage* as shown in Figure B.5. After clicking *Create* you can copy the token and use it to publish the extension. The token cannot be viewed again after closing the page, so make sure to save it somewhere, preferably a credentials manager.


Create a new personal access token

Name: SAMT

Organization: marceljoss0794

Expiration (UTC): 90 days — 7.9.2023

Scopes
Authorize the scope of access associated with this token
Scopes: Full access [✓ Custom defined]

Read / Read & write

Marketplace
Read, publish, update, and manage items and publishers
[✓ Read] Acquire [✓ Publish] [✓ Manage]

Member Entitlement Management
Read and manage users
Read / Read & write

Personal Access Tokens
These can be used instead of a password

Token name
samt Marketplace (Manage)

Figure B.5: Azure DevOps personal access token

Publishing

Before publishing the extension you need to update the version number in the `package.json` file and document the changes and additions in the `CHANGELOG.md` file. Now run `pnpm run publish` in the terminal, which will prompt you for your personal access token if you are running it for the first time. Afterwards it will take a few minutes for the new version to appear in the marketplace, because extensions have to pass an automated validation process. Once that is done you will be notified by email.

B.2.6 Documentation

The user documentation of the extension is the `README.md` file in the repository, which is also displayed on the marketplace. Additional documentation for developers is kept in the wiki of the repository, where it is also explained how to debug the SAMT Language Server.[11]

[11] https://github.com/samtkit/vscode/wiki/Debugging-the-Language-Server

Appendix C

User Feedback

This chapter documents the complete user feedback collected during the testing phase. Participants were given 30 minutes to implement a simple service in SAMT and provide feedback on the language and tooling. The feedback and our observations have been translated into English, categorized by topic, and anonymized, but otherwise left unchanged.

Every statement is also categorized into one of the following categories:

+ Positive feedback
− Negative feedback
= Neutral feedback
• Observation

C.1 Developer 1

Developer 1 works at Zürcher Kantonalbank and modeled different APIs in the existing infrastructure.

IDE Features

+ Word autocompletion works well
+ Snippets are very useful

= Variable rename would be nice

- Mostly copy-pasted example code and modified it
- Was happy with autocompletion, even though it is just the standard VS Code functionality with no language specific intelligence

Data Types

= Generate enum with default value to enable backwards compatibility

- Created dedicated data types for every request and response to enable backwards compatibility

Syntax

– Does not like the possibility to have both regular and short-form constraints, would only allow one to prevent mixed usage

– Did not find the `alias Foo: Type` syntax intuitive

Technical Issues

– Error messages for incomplete range syntax (e.g. `String(5..)`) are not very helpful

– Find references for enum does not work

Other

= Would be interested in how interceptors, proxy objects and other features are implemented

C.2 Developer 2

Developer 2 works at Zürcher Kantonalbank and has been working on integrating gRPC into the existing infrastructure.

IDE Features

– Autocomplete for types of record field is missing

• Did not use snippets (not aware of them)

Data Types

– Decimal precision specification is missing (e.g. `Decimal(10,2)`)

• `String(5)` was assumed to mean a string with exactly 5 characters (equal to `String(5..5)`)

Syntax

+ Record keyword analogous to modern Java concept

– Question mark placed before type instead of after (e.g. `?String`, how it is in other tools within Zürcher Kantonalbank)

= Maybe rename query to queryParam to avoid confusion

• Tried to write `throws` instead of `raises` to indicate an error

• Read the first part of the transport configuration (e.g. `header: foo`) as the parameter name

Error Handling

– Why is it called fault and not exception?

• A snippet or tooling assistance for faults would be nice

Technical Issues

– Why not implement it with ANTLR to make maintenance easier?

• Language server crashed in some edge-cases and did not correctly restart

Other

= Not much better, not worse either, just a different language

C.3 Developer 3

Developer 3 works at Zürcher Kantonalbank and has been using the existing infrastructure for a long time.

IDE Features

+ Word completion is nice
− No formatter for SAMT
• Spent a lot of time writing import statements by hand
• Did not use snippets (not aware of them)
• Initially interpreted the name after `consume` as the consumer name

Data Types

− Trailing comma in transport configuration causes error
= Tried to use `String[]` to indicate an array of strings, but quickly learned the correct syntax
= Naming best practices for data types would be nice (e.g. ending with `Service` for services)
• Initially tried to write types in lower case (e.g. `string` instead of `String`)

Syntax

= Maybe use a different syntax for the transport configuration (e.g. `foo, bar as header`)
= The ability to specify the return encoding would be nice (e.g. `application/pdf` for binary, or send a string in a header)

Technical Issues

• Error messages for incomplete range syntax (e.g. `String(5..)`) are not very helpful

Other

= Would not recommend trying to support every possible use case, but rather focus on the most common ones and use transport specific solutions for the rest

C.4 Developer 4

Developer 4 works at Zürcher Kantonalbank and works in the environment of the existing integration infrastructure.

IDE Features

– Wanted to autocomplete all available data types, which is not possible

Data Types

- Tried to add a `: void` to mark a operation as not returning anything
- Initially tried to write types in lower case (e.g. `boolean` instead of `Boolean`)
- `String(5)` was assumed to mean a string with exactly 5 characters, but after some thought it would make more sense to interpret it as a max length (equal to `String(0..5)`)

Syntax

+ Initially placed question mark before type, but actually finds our version with the question mark after the type more readable
+ Approves of the generic `List<T>` syntax which is more flexible than `T[]` (e.g. `Map<K, V>`)

– The usage of `consume` and `provide` seems inconsistent compared to `implements` and `uses`

– Strongly is against the use of `*` in range syntax (e.g. `String(5..*)`) as it is not clear what it means

– Would not allow the use of star at the beginning of length constraints as it makes no sense (e.g. `String(*..10)` should be replaced with `String(0..10)`)

– Does not like that the package is placed after the import statements, it gives the wrong impression that more than one package per file can be specified

– Code documentation should be easy to write and read, but the current syntax is quite verbose when documenting an operation parameter, would prefer a system like JavaDoc

– Does not like the use of the `=` in the type alias syntax, would prefer a more verbose version (e.g. `type Foo as String`)

- Tried to add second service implementation comma separated, which is not supported (e.g. `implements A, B`)

Technical Issues

– Created the same operation name `hi` in two services which are implemented by the same provider, which resulted in the same runtime URL, which should have caused at least a warning

Other

+ The existing infrastructure has a weird model-package to java-package behavior, which is now fixed

C.5 Internal Developer Conference

Züricher Kantonalbank performs internal developer conferences several times a year to share their latest experiences. Fortunately, SAMT was allocated one of the four presentation slots on June 13, 2023, which was used to present SAMT to 130 attendees.

Show of Hands During the presentation, the audience was asked to raise their hands in response to the questions listed in Table C.1. In summary, the majority had used the existing integration infrastructure, but relatively few had enjoyed the process.

Question	Response rate
Who has used the existing integration infrastructure before?	60%
Who used the official IDE to do so?	40%
Who enjoyed the process?	20%

Table C.1: Conference feedback by show of hands

Feedback Form At the end of the presentation, the audience was asked to fill out a short questionnaire to gather feedback for all presentations. Of the 130 attendees, 57 filled out the presentation questionnaire with overwhelmingly positive feedback. As these responses do not focus on SAMT itself, they are not included in this report.

In addition, 30 attendees left a comment, both focusing on the SAMT language and the presentation itself. All comments which had any reference to SAMT as a project were translated into english and listed here:

+ Very interesting presentation and interesting project
+ Goooo SAMT
+ I like SAMT
+ I really like that SAMT is open-source
+ Great project and interesting topic for a bachelor thesis
+ Great topic and also solution for a bachelor thesis. It would be a pity if it is not developed further. Very well presented.
+ Very awesome, wold like to use it immediately!
+ Very nice! Just added a star on GitHub and hope that we can use it here soon.
+ Once again great presentation! Please continue this SAMT project.

= Will keep an eye on SAMT

− Besides the context of a Bachelor's thesis it would be interesting to ask: why a custom solution instead of using an existing one?

C.6 Summary

This section summarizes the feedback from all participants along with the actions we have taken. We selected the following feedback as the most important, each of which is discussed in more detail in the following paragraphs.

Lack of IDE Features The lack of certain IDE features was the most common complaint. The most important missing features are listed in order of priority:

1. Context-aware autocompletion
2. File formatting
3. Global rename
4. Add import for missing types

All of these features can be implemented in the SAMT Visual Studio Code Extension, but were not implemented due to time constraints. Especially the lack of context-aware autocompletion was mentioned by almost every participant, which is why it was added as a high priority in the outlook section of the report.

Range Syntax The use of the `*` character to define open-ended ranges can be confused with cardinality indicators. A participant proposed to just leave out the upper or lower bound to indicate an open-ended range. This was discussed and rejected in the design phase, both because of syntax ambiguity and visual incompleteness. However, the feedback is still valid and should be considered in future iterations.

One improvement was implemented to reduce the ambiguity of the range syntax. The lower bound for sizes now reports a warning if the `*` character is used, as it is always identical to `0`. For example, `String(*..5)` should now be modeled as `String(0..5)`.

Minor Syntax Inconsistencies Some feedback was given on minor syntax inconsistencies, such as the use of `=` for type alias which is not used anywhere else in the language. Another example is the use of `provide` and `consume` instead of `provides` and `consumes`. These inconsistencies would be easy to fix, but require further discussion to ensure that the language is consistent throughout. Depending on the outcome of the discussion, these changes may be implemented in future iterations and require a breaking change.

Advanced REST Features One participant expressed the desire to model more advanced HTTP endpoints, such as an operation which accepts a JSON and returns a PDF. This is currently not possible in SAMT, as the request and response body must be of the same type. Luckily, this is a problem with the implementation of the SAMT HTTP transport configuration parser and not the language itself. This means that a community provided plugin could also implement this feature, which is why it was not prioritized.

Improved Error Handling On many occasions, participants caused errors in their SAMT code which were not anticipated by the team. One reoccurring example is the improper use of the `?` operator in front of the type instead of after. This specific error is now reported with a helpful error message, instead of a generic syntax error.

Order of Package A participant suggested to change the order of the package declaration to be the first element in a SAMT file. This way, it is easier to see which package a file belongs to, especially since every file must be in a package. It would also be consistent with Java, which shares many other syntax elements with SAMT surrounding package declarations and imports. The implementation of this change is trivial, but requires updating all existing examples and documentation. Because of this, the change was not implemented in this iteration due to time constraints.

Appendix D

Dependencies

This chapter lists all of the direct dependencies.

D.1 SAMT Core

This section lists the direct dependencies of the SAMT Core project. For a full list of all dependencies including transitive ones, you can run `./gradlew` *`project`*`:dependencies` for each subproject.

Libraries

Listing D.1 lists the libraries used in the SAMT Core project, with the project's website and Maven Central page linked.

```
Kotlin Standard Library JDK8 Extension
https://kotlinlang.org/api/latest/jvm/stdlib/
https://central.sonatype.com/artifact/org.jetbrains.kotlin/
   kotlin-stdlib-jdk8/1.8.22

jCommander
https://jcommander.org
https://central.sonatype.com/artifact/com.beust/jcommander
   /1.82

Mordant
https://ajalt.github.io/mordant/
https://central.sonatype.com/artifact/com.github.ajalt.
   mordant/mordant/2.0.0-beta13

kotlinx.serialization-json
https://github.com/Kotlin/kotlinx.serialization
https://central.sonatype.com/artifact/org.jetbrains.kotlinx/
   kotlinx-serialization-json/1.5.1

Kaml
https://github.com/charleskorn/kaml
https://central.sonatype.com/artifact/com.charleskorn.kaml/
   kaml/0.54.0

Eclipse LSP4J
https://projects.eclipse.org/projects/technology.lsp4j
https://central.sonatype.com/artifact/org.eclipse.lsp4j/org.
   eclipse.lsp4j/0.21.0

kotlin.test
https://kotlinlang.org/api/latest/kotlin.test/
https://central.sonatype.com/artifact/org.jetbrains.kotlin/
   kotlin-test/1.9.0-Beta
```

Listing D.1: Libraries used in SAMT Core

Plugins

Listing D.2 lists the Gradle plugins used in the SAMT Core project, with the project's website and Gradle Plugin Portal or Maven Central page linked.

```
Gradle Shadow
https://github.com/johnrengelman/shadow
https://plugins.gradle.org/plugin/com.github.johnrengelman.
   shadow

Kover - Kotlin Code Coverage Tool
https://kotlin.github.io/kotlinx-kover/gradle-plugin/
https://central.sonatype.com/artifact/org.jetbrains.kotlinx.
   kover/org.jetbrains.kotlinx.kover.gradle.plugin/0.7.1

Gradle Git Versioning Plugin
https://github.com/qoomon/gradle-git-versioning-plugin
https://plugins.gradle.org/plugin/me.qoomon.git-versioning
```

Listing D.2: Gradle plugins used in SAMT Core

D.2 SAMT Visual Studio Code Extension

Listing D.3 lists the direct dependencies of the SAMT Visual Studio Code Extension, using the output of `pnpm run list --depth 0 --long` stripped of local paths. To get a full tree of all depedencies including transitive ones, run `pnpm run list --depth Infinity`.

```
dependencies:
@microsoft/vscode-file-downloader-api 1.0.1
  Wrapper package for the VS Code File Downloader extension
  git+https://github.com/microsoft/vscode-file-downloader-
     api.git
  https://github.com/microsoft/vscode-file-downloader-api#
     readme
axios 1.4.0
  Promise based HTTP client for the browser and node.js
  git+https://github.com/axios/axios.git
  https://axios-http.com
get-port 7.0.0
  Get an available port
  git+https://github.com/sindresorhus/get-port.git
  https://github.com/sindresorhus/get-port#readme
jdk-utils 0.4.6
  JDK related utils for Java related development.
  git+https://github.com/Eskibear/node-jdk-utils.git
  https://github.com/Eskibear/node-jdk-utils#readme
vscode-languageclient 8.1.0
  VSCode Language client implementation
  git+https://github.com/Microsoft/vscode-languageserver-
     node.git
  https://github.com/Microsoft/vscode-languageserver-node#
     readme
which 3.0.1
  Like which(1) unix command. Find the first instance of an
     executable in the PATH.
  git+https://github.com/npm/node-which.git
  https://github.com/npm/node-which#readme

devDependencies:
@types/mocha 10.0.1
  TypeScript definitions for mocha
  git+https://github.com/DefinitelyTyped/DefinitelyTyped.git
  https://github.com/DefinitelyTyped/DefinitelyTyped/tree/
     master/types/mocha
@types/node 16.18.34
  TypeScript definitions for Node.js
  git+https://github.com/DefinitelyTyped/DefinitelyTyped.git
  https://github.com/DefinitelyTyped/DefinitelyTyped/tree/
     master/types/node
@types/vscode 1.78.1
  TypeScript definitions for Visual Studio Code
  git+https://github.com/DefinitelyTyped/DefinitelyTyped.git
  https://github.com/DefinitelyTyped/DefinitelyTyped/tree/
     master/types/vscode
@types/which 3.0.0
  TypeScript definitions for which
  git+https://github.com/DefinitelyTyped/DefinitelyTyped.git
  https://github.com/DefinitelyTyped/DefinitelyTyped/tree/
     master/types/which
@typescript-eslint/eslint-plugin 5.59.9
  TypeScript plugin for ESLint
  git+https://github.com/typescript-eslint/typescript-eslint
     .git
  https://github.com/typescript-eslint/typescript-eslint#
     readme
@typescript-eslint/parser 5.59.9
  An ESLint custom parser which leverages TypeScript ESTree
  git+https://github.com/typescript-eslint/typescript-eslint
     .git
  https://github.com/typescript-eslint/typescript-eslint#
     readme
@vscode/test-electron 2.3.2

  git+https://github.com/Microsoft/vscode-test.git
  https://github.com/Microsoft/vscode-test#readme
@vscode/vsce 2.19.0
  VSCode Extension Manager
  git+https://github.com/Microsoft/vsce.git
  https://code.visualstudio.com
esbuild 0.17.19
  An extremely fast JavaScript and CSS bundler and minifier.
  git+https://github.com/evanw/esbuild.git
  https://github.com/evanw/esbuild#readme
eslint 8.42.0
  An AST-based pattern checker for JavaScript.
  git+https://github.com/eslint/eslint.git
  https://eslint.org
eslint-config-prettier 8.8.0
  Turns off all rules that are unnecessary or might conflict
      with Prettier.
  git+https://github.com/prettier/eslint-config-prettier.git
  https://github.com/prettier/eslint-config-prettier#readme
glob 10.2.7
  the most correct and second fastest glob implementation in
      JavaScript
  git://github.com/isaacs/node-glob.git
  https://github.com/isaacs/node-glob#readme
mocha 10.2.0
  simple, flexible, fun test framework
  git+https://github.com/mochajs/mocha.git
  https://mochajs.org/
prettier 2.8.8
  Prettier is an opinionated code formatter
  git+https://github.com/prettier/prettier.git
  https://prettier.io
typescript 5.1.3
  TypeScript is a language for application scale JavaScript
     development
  git+https://github.com/Microsoft/TypeScript.git
  https://www.typescriptlang.org/
```
Listing D.3: NPM packages used in the SAMT Visual Studio Code Extension

D.3 GitHub Actions

Listing D.4 lists the GitHub Actions used by both respositories alongside their GitHub respositories.

```
actions/checkout@v3
https://github.com/actions/checkout

actions/setup-java@v3
https://github.com/actions/setup-java

gradle/gradle-build-action@v2
https://github.comm/gradle/gradle-build-action

mikepenz/gradle-dependency-submission@main
https://github.com/mikepenz/gradle-dependency-submission

github/codeql-action/init@v2
https://github.com/github/codeql-action

github/codeql-action/analyze@v2
https://github.com/github/codeql-action

github/codeql-action/upload-sarif@v2
https://github.com/github/codeql-action

softprops/action-gh-release@v1
https://github.com/softprops/action-gh-release

actions/setup-node@v3
https://github.com/actions/setup-node

actions/upload-artifact@v3
https://github.com/actions/upload-artifact

pnpm/action-setup@v3
https://github.com/pnpm/action-setup
```
Listing D.4: GitHub Actions used by SAMT

Glossary

ANTLR is a popular parser generator tool.

API Application Programming Interface.

AST Abstract Syntax Tree.

C4 is a framework used in visualizing the architecture of a software system. It stands for Context, Container, Component and Code..

CI Continuous Integration.

CLI Command-Line Interface.

DSL Domain-Specific Language.

Extended Backus–Naur form (EBNF) is a notation to express a context-free grammar.

FR Functional Requirement.

Gradle is a build automation tool used primarily for Java projects.

gRPC is a high-performance open-source framework developed by Google that enables server-side communication..

HTTP Hyper Text Transport Protocol.

IDE Integrated Development Environment.

IR Intermediate Representation.

JAR Java Archive.

JVM Java Virtual Machine.

Kafka is an open-source distributed event streaming platform for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications..

Kotlin is an open-source framework for building servers and clients in connected systems using the Kotlin programming language.

Ktor is an open-source framework for building servers and clients in connected systems using the Kotlin programming language.

Lexer performs lexical analysis. It takes a stream of characters as input and converts it into a stream of tokens.

LSP Language Server Protocol.

Maven is a build automation tool used primarily for Java projects.

MDSL Microservice Domain Specific Language.

NFR Non Functional Requirement.

OWASP Open Web Application Security Project.

Parser performs syntactic analysis. It takes a stream of tokens as input and converts it into an abstract syntax tree.

RPC Remote Procedure Call.

SAMT Simple API Modeling Toolkit.

Semantic Analyzer performs semantic analysis. It takes an abstract syntax tree as input and performs semantic checks, such as type checking.

SOAP Simple Object Access Protocol.

TextMate is a text editor for macOS.

URI Uniform Resource Identifier.

Vertical Prototype focuses on implementing a single subsystem or function throughout all layers.

VSCode Visual Studio Code.

WSDL Web Services Description Language.

Xtext is a framework for the development of domain-specific languages..

List of Figures

List of Tables

Listings

Bibliography

[1] C. Dietrich, *Call to action: Secure the future maintenance of xtext*, 2020-03. [Online]. Available: https://github.com/eclipse/xtext/issues/1721 (visited on 2023-06-11).

[2] D. Miller, J. Whitlock, M. Gardiner, M. Ralphson, R. Ratovsky, and U. Sarid, *OpenAPI Specification v3.1.0*, 2022-02. [Online]. Available: https://spec.openapis.org/oas/v3.1.0 (visited on 2023-06-14).

[3] O. Zimmermann, M. Stocker, D. Lübke, U. Zdun, and C. Pautasso, *Patterns for API design* (Addison-Wesley Signature Series (Vernon)). Boston, MA: Addison Wesley, 2023-01.

[4] R. Chinnici, J.-J. Moreau, A. Ryman, and S. Weerawarana, "Web services description language (wsdl) version 2.0 part 1: Core language," *W3C recommendation*, vol. 26, no. 1, p. 19, 2007.

[5] K. Varda, "Protocol Buffers: Google's Data Interchange Format," 2008-07. [Online]. Available: https://opensource.googleblog.com/2008/07/protocol-buffers-googles-data.html (visited on 2023-06-14).

[6] F. Montesi, C. Guidi, and G. Zavattaro, "Composing services with JOLIE," in *Fifth European Conference on Web Services (ECOWS'07)*, IEEE, 2007, pp. 13–22.

[7] *AsyncAPI Specification*, AsyncAPI Initiative, 2023-02. [Online]. Available: https://www.asyncapi.com/docs/reference/specification/v2.6.0 (visited on 2023-06-14).

[8] T. J. Parr and R. W. Quong, "ANTLR: A predicated-LL (k) parser generator," *Software: Practice and Experience*, vol. 25, no. 7, pp. 789–810, 1995.

[9] G. Li, *The comment of the class JavacParser is not appropriate*, 2021-06. [Online]. Available: https://github.com/openjdk/jdk/commit/b98e52a49191cfbb7d954646cd80a6711daeaca6 (visited on 2023-06-14).

[10] J. Myers, *New_C_Parser*, 2008-01. [Online]. Available: https://gcc.gnu.org/wiki/New_C_Parser (visited on 2023-06-14).

[11] *Intellij Platform Plugin SDK - Custom Language Support*, JetBrains s.r.o, 2023. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/custom-language-support.html (visited on 2023-03-03).

[12] *Incompatible Changes in Intellij Platform and Plugins API*, JetBrains s.r.o, 2023. [Online]. Available: https://plugins.jetbrains.com/docs/intellij/custom-language-support.html (visited on 2023-03-03).

[13] *Stack Overflow Developer Survey 2022*, Stack Overflow, 2022. [Online]. Available: https://survey.stackoverflow.co/2022/#most-popular-technologies-new-collab-tools-prof (visited on 2023-03-02).

[14] *Language Server Protocol Specification - 3.17*, Microsoft, 2022-05. [Online]. Available: https://microsoft.github.io/language-server-protocol/specifications/lsp/3.17/specification/ (visited on 2023-03-03).

[15] *Visual Studio Code - Syntax Highlight Guide*, Microsoft, 2023-03. [Online]. Available: https://code.visualstudio.com/api/language-extensions/syntax-highlight-guide (visited on 2023-03-03).