

Institute of Software Technology

University of Stuttgart  
Universitätsstraße 38  
D-70569 Stuttgart

Bachelorarbeit

## SKiL Language Server

Johannes Schäufele

<b>Course of Study:</b>	Informatik
<b>Examiner:</b>	Prof. Dr. Erhard Plödereder
<b>Supervisor:</b>	Dipl.-Inf. Felix Krause, Dr. Timm Felden
<b>Commenced:</b>	April 9, 2018
<b>Completed:</b>	October 9, 2018



## **Abstract**

Language analysis features offered by integrated development environments (IDEs) can ease and accelerate the task of writing code, but are often not available for domain-specific languages. The Language Server Protocol (LSP) aims to solve this problem by allowing language servers that support these features for a certain programming language to be used portably in a number of IDEs. A language server for Serialization Killer Language (SKilL) was implemented that supports a multitude of language features including automatic formatting, completion suggestions, and display of references and documentation associated with symbols. This thesis presents how the language server was implemented and discusses associated challenges that arose due to the nature of the SKilL and LSP specification.

## **Kurzfassung**

Sprachunterstützungen von IDEs können Programmierern ihre Arbeit deutlich erleichtern, jedoch werden diese oft nicht für domänenspezifische Sprachen unterstützt. Das LSP erlaubt Unterstützungen für eine Sprache durch einen Language Server zu implementieren, der anschließend in einer Vielzahl an IDEs nutzbar ist. Es wurde ein Language Server für SKiL implementiert, mit dem unter anderem Code automatisch formatiert, Vervollständigungen vorgeschlagen sowie Verweise und Dokumentation angezeigt werden kann. In dieser Ausarbeitung wird das Vorgehen und dabei aufgekommene Herausforderungen durch die SKiL und LSP Spezifikation diskutiert.

# Contents

<b>1</b>	<b>Introduction</b>	<b>7</b>
1.1	Motivation . . . . .	7
1.2	Outline . . . . .	7
<b>2</b>	<b>The Language Server Protocol</b>	<b>9</b>
2.1	Overview . . . . .	9
2.2	Language features . . . . .	10
2.3	Language and client support . . . . .	11
2.4	Limitations . . . . .	13
<b>3</b>	<b>Serialization Killer Language</b>	<b>15</b>
3.1	Overview . . . . .	15
3.2	Specification . . . . .	20
<b>4</b>	<b>Implementation</b>	<b>23</b>
4.1	Managing source files . . . . .	23
4.2	Analysis . . . . .	24
4.3	Language features . . . . .	29
<b>5</b>	<b>Evaluation</b>	<b>39</b>
5.1	Testing . . . . .	39
5.2	Performance . . . . .	42
5.3	Task fulfillment . . . . .	43
<b>6</b>	<b>Conclusions and future work</b>	<b>45</b>
6.1	Conclusions . . . . .	45
6.2	Future work . . . . .	46
	<b>Bibliography</b>	<b>57</b>



# 1 Introduction

## 1.1 Motivation

IDEs improve programmers' user experience and can increase their productivity by offering a wide array of language analysis features, ranging from displaying associated documentation over jumping to definitions to auto-completion. Traditionally, these inherently language-specific features have to be implemented for every possible IDE and language pair to ensure complete coverage.

The amount of work associated with this is impractical and thus especially smaller, domain-specific languages are often not supported in many IDEs and less popular IDEs only support language analysis features for a few languages at most. This motivates a way of implementing language analysis features independent of the supported IDE.

The LSP [Mic16a] aims to solve this problem by defining an interface between IDEs and language servers, allowing language servers that support language features for a programming language to be implemented independently of specific IDEs, enabling them to be available in all IDEs that support the LSP.

The domain-specific language SKiL [Fel17], used for platform and language independent serialization, faced a similar problem as was described previously: An eclipse-based IDE "skilled" [Stu15] was developed, but the support for SKiL language features was limited to this IDE and could not easily be made available in other IDEs.

Getting used to the concepts and workflow of a specific IDE can cost time, and once one is used to working with a certain IDE, the transfer of switching to a different one may incur much of that time investment a second time [KS05], making language support in IDEs that users are already proficient in all the more valuable.

Editing SKiL files in an IDE that supports the LSP also comes at the added benefit of being able to view both the specification and the generated code for target languages in the same IDE while being able to use language features for both.

## 1.2 Outline

A language server for SKiL was implemented that supports most important language features and is usable in many IDEs through the LSP. This additionally serves as a case study to evaluate the viability of implementing language features for domain specific languages on the basis of the LSP in general.

In chapter 2, the fundamental concepts of the LSP are described in more detail and the current state of the protocol and its implementations with their advantages and shortcomings are presented. Following this, chapter 3 lays the foundation for SKiLL and clarifies the specification written against. Chapter 4 then details the language server's implementation, touching on the approach taken and difficulties encountered as well as the current state of the implementation. Finally, chapter 6 summarizes the results of this paper, draws conclusions based on them, and presents an outlook on possible future work.



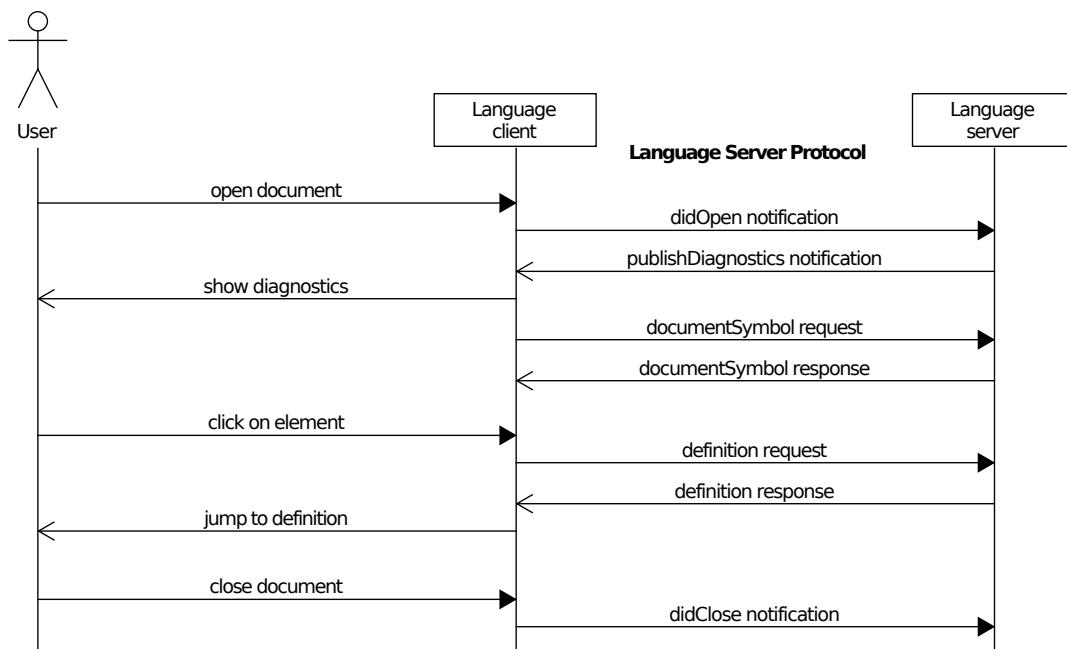
## 2 The Language Server Protocol

### 2.1 Overview

The Language Server Protocol (LSP) [Mic16a] defines a set of request, command, and message types that allow the decoupling of IDEs and their features from language support features by splitting responsibilities across language clients and language servers. This independence enables any IDE implementing a language client to use all existing language servers for various languages and a newly written language server to be compatible with many existing IDEs, at least in principle.

Without such a protocol, a new IDE would have to implement language support for every language from scratch and new programming languages would have to be integrated into a multitude of IDEs to achieve the same coverage, resulting in a vast and often infeasible amount of work.

Language clients are usually part of an IDE or available as a plugin for one and language servers are implemented as separate applications for each supported language.



**Figure 2.1:** Exemplaric messages sent between the language client and server, based on the first figure in [Mic16d]. A more detailed overview of exchanged messages including concrete parameters can be found on the LSP Inspector page [Mic16c].

The protocol itself is based on JSON remote procedure call (JSON-RPC), where messages are serialized to JavaScript Object Notation (JSON) before transmission.

A typical session might begin with the user opening a file in their IDE, causing it to start the appropriate language server and then send it messages including initialization information defining a workspace, a notification for the file that was opened, and requests for a list of all symbols and actions associated with the document. The server would generally analyze all relevant files, send diagnostics including warnings and errors to the client, and respond appropriately to all requests made by the client, such as symbol and action requests, using an internal representation of the analyzed files.

### 2.2 Language features

Language servers can implement a variety of features then available for use in the client. Some of the currently specified features include

- **Diagnostics:** Messages with varying level of severity, such as hint, info, warning, and error, can be sent from the server to the client. This is usually done when a document is first seen or whenever it is modified.
- **Definition:** One can jump to the definition of symbols from all of their references. The same is possible exclusively for types using **type definition**.
- **Hover:** Upon hovering over symbols, associated information, such as parts of the definition, its type, and documentation can be displayed.
- **Highlight:** Identifiers which refer to the same symbol can be marked as such through information supplied by the server, and subsequently be highlighted simultaneously on selection.
- **References:** For a position in a document, all references to the symbol at that position in the workspace can be shown.
- **Symbols:** All symbols in a document, along with their type, such as a class, an enum, or a field, and optionally the surrounding structure they were declared in, as well as their declaration position can be requested by the client.
- **Rename:** Symbols can be renamed, causing all of their occurrences in the workspace to be replaced accordingly.
- **Completion:** Incomplete prefixes in the code can be completed with a list of suggestions, alongside associated documentation and type information, based on the surrounding context.
- **Formatting:** An entire file can be formatted according to certain rules, often based on a style guide which mandates certain indentation, usage of spacing, newline placement, and maximum line lengths, amongst other things. Additionally, **range formatting** can be used to format a selected subset of the document and **on type formatting** can be used to format text as it is typed in by the user.
- **Workspace symbols:** One can list all symbols in the current workspace (partially) matching a search term, alongside its documentation and type.

- **Folding range:** Larger elements, such as import groups, multi-line comments, and class or method bodies, can be shown and hidden in their entirety using toggleable folds.
- **Code lens:** Code lenses can be displayed alongside a line of code, that could indicate things such as the number of references or how many of the tests associated with a method at that point were successful. They can also be associated with a command, such as rerunning all tests for the current method.
- **Code action:** Code actions can be associated with a range of code, allowing errors or warnings to be fixed automatically or automatic refactorings to be performed, such as inlining or extracting a method.
- **Commands:** Commands can be invoked in the client to execute arbitrary actions on the server. This could range from inserting a piece of code or formatting the document in a certain way to compiling the current document or clearing the server's cache.
- **Implementation:** Can be used to jump to the implementation of abstract methods. Not relevant for SKiL, as it does not feature methods.
- **Signature help:** Shows the signatures for callable symbols. Not relevant for SKiL, as there are no callables in SKiL.

Other features include link and color support, which are less relevant to SKiL and were not implemented.

Language server implementations can choose to only support a subset of these features. All supported features and chosen options, such as whether to transmit changed document contents in full or just the changed ranges, are sent to the client during initialization.

Several unofficial extensions to the protocol that define additional language features also exist and are available in a smaller subset of IDEs and language servers. Current experimental features in development include semantic highlighting and type hierarchy features missing in the protocol. More fundamental extensions to the protocol allow for it to be used without a shared file system between the client and server or add metadata to symbols that is then available through extended versions of the definition, references, and workspace symbol features.

Clients may specify a workspace root when starting the server. This workspace directory marks the base for all files viewed in the IDE, similar to the concept of a project, and is used to determine which source files are eligible for workspace features like the workspace symbol search or even listing references. Whether the workspace encompasses all subdirectories recursively or only those directly in it is left up to language servers. Currently only one workspace root can be specified at startup time, proposed extensions to the protocol allow setting multiple workspace roots simultaneously.

## 2.3 Language and client support

Though at the time of writing the LSP is still a fairly recent development, a sizable amount of programming languages, ranging from major languages to more domain specific ones, are supported through language servers. A list of available language servers can be found at [Mic16b] and the

unofficial [Sou16b] which includes an overview of the supported features. This does not seem to leave much more to be desired when working with somewhat well-known languages other than the protocol’s limitations themselves.

On the client side, some of the major IDEs, but far from all of them, implement the LSP with varying degrees of functionality and usability. A list of available language clients can be found at [Mic16f] and on [Sou16b].

While part of this effort comes from the IDEs’ developers themselves, a portion of it is user-driven and delivered in the form of plugins or similar extension mechanisms.

In particular, Visual Studio Code (VSCode), for which the LSP was initially developed and which serves as a reference implementation for language clients, unsurprisingly is the most complete and usable implementation for language clients followed by Eclipse, whereas the user-driven implementations often trail behind in feature completeness and usability.

Given enough time, the level of acceptance and support of the LSP in IDEs may rise, alleviating the current shortage of complete and robust language client implementations and lead to coverage for all major IDEs. This trend could be hinted at by a number of IDEs that either already consider adding such support or even have ongoing efforts to do so.

	VSCode	Eclipse	JetBrains IDEs	Sublime Text 3
Essential features	X	X	X	
All features	X			
Officially maintained	X	X		
Out of the box client	X			
Out of the box servers		X	X	X
Robust implementation	X	X		X

**Table 2.1:** Feature matrix comparing different language client implementations currently available in IDEs. A ticked cell indicates the language client implementation possessing the respective capability or feature.

Table 2.1 summarizes the feature completeness and usability of language clients that were tested in the process of the work presented in this thesis. Here, essential features refer to the diagnostics, formatting, and completion features, to give a simple baseline of features to compare against. It is indicated whether the language client implementation is maintained officially by the developer of the IDE and whether the language client ships with all installations of the IDE, referred to as “out of the box client”. There is also an indication of whether executable language servers can be used in the IDE through configuration mechanisms, but without having to write any additional code for integration, referred to as “out of the box servers”. Finally, “robust implementation” refers to IDEs that did not show any noticeable bugs or impairments when using their features.

## 2.4 Limitations

Though the LSP supports most common language features, syntax highlighting stands out as not being supported despite the feature being very wide-spread, useful, and often requested to be supported by the protocol. There are ongoing efforts to add syntax highlighting to the LSP, which suggests that it will be supported eventually. In the mean time, the de facto standard way of supporting syntax highlighting in IDEs remains to be through TextMate Language Grammars [Mac], which are supported by many IDEs, or generators with additional targets, such as Iro [Con17], which allows the definition of all syntactical elements in a stack-based matching language. This definition can then be used to generate syntax descriptions as TextMate Language Grammars and the equivalent for many other IDEs, such as Atom, Sublime Text 3, and others, increasing the amount of supported IDEs to include some that do not support TextMate Language Grammars.

Another common feature that was not supported when efforts to implement a language server for SKiLL began was code folding, which allows snippets of code, often enclosed in braces, to be hidden or displayed through folding. Support for this functionality was recently added in version 3.10 of the protocol. Otherwise, the LSP is reasonably feature complete, supporting many features for language analysis, barring some more involved and less frequently used or more language-specific features, such as features specific to functional languages.

The protocol does also not specify certain aspects necessary for implementation of functioning language clients and servers, such as the communication channel used between the client and server (popular choices include standard in/out and TCP connections), specific implementation of features, such as how commands are invoked from the IDE or how configuration options are set, and how language servers are acquired, installed, and started.

This leads to implementation differences in different language clients and servers which can result in the very incompatibilities the LSP was meant to solve. In particular, the installation of language servers being unclear results in very different procedures having to be taken for IDEs, sometimes necessitating additional IDE specific wrapper plugins around the language server to be written for individual IDEs. These complications are far from ideal and a step back towards IDE dependent behavior, thus should be addressed by the specification itself. Despite this, the effort required for such integration is often very limited and dwarfs that of implementing language features for different APIs of IDEs, thus still results in less incompatibilities and additional effort than what would have been caused by solutions without the LSP.



## 3 Serialization Killer Language

Serialization Killer Language (SKiL) [Fel17] is a domain specific language that allows platform and language independent serialization of data. SKiL files are specifications that describe the data structure layout used for serialization. Based on a specification, a SKiL compiler can generate source code for bindings in various target languages, that allow serialization and deserialization of the described data structures natively in the target language.

### 3.1 Overview

Specifications can contain single-line head comments beginning with #, usually containing a general description of the specification file's purpose and functionality.

Following this, other specifications' declarations can be imported using `include`, or alternatively with statements that specify the specification's typically relative path. The include hierarchy may be recursive, specifications imported multiple times are treated the same as if they were only included a single time. Conceptually, included specifications are treated as if their declarations were inserted before all of those in the current file.

---

```
# Example skill specification
```

```
include "other.skill"
```

---

**Listing 3.1:** Example of a specification file with a head comment and an include

#### 3.1.1 Declarations

The main part of specification files consists of a list of declarations. Declarations are the primary artifacts of specifications and are used to build up all data structures later used for serialization. Each declaration may be preceded by a Javadoc [Ora14] / Doxygen-style [Hee16] comment that will be included in the compiled form and generated code to be available to all tools using the specification or one of its derivatives. Most declaration types may also be annotated with type annotations, namely restrictions and hints, that can restrict the allowable behavior for a type and give hints potentially useful for optimization to generators respectively.

### 3 Serialization Killer Language

---

There are several different types of declarations. The most simple type of declaration is a user type declaration. It consists of an identifier, optionally followed by a list of types it inherits from, and then a brace-enclosed list of fields.

---

```
/**
 * Example user type
 *
 * @version 1.0
 */
Child : Parent with Interface {
  // Fields
}
```

---

**Listing 3.2:** Example code for a user type inheriting from another user type and an interface

Syntactically similar declarations are those for interfaces. They are prefixed by the keyword `interface`, but otherwise follow the same structure as user type declarations.

---

```
interface J {
}

interface I : A with J {
  // Fields
}
```

---

**Listing 3.3:** Simple interface example, A represents a user type

User types and interfaces may transitively inherit from arbitrarily many interfaces, but only from a single user type at most. The inheritance hierarchy must also be acyclic. Inheritance covers all fields.

Enumerations are prefixed by the keyword `enum`, followed by an identifier for the type and a brace-enclosed, non-empty list of enum values and a potentially empty list of fields. Unlike user types and interfaces, enums may not inherit from other types and they may also not have any hints or restrictions.

---

```
enum Color {
  RED,
  GREEN,
  BLUE;

  // Fields
}
```

---

**Listing 3.4:** Example code with an enumeration of colors



---

Finally, typedefs may be used to alias type names with additional type annotations. They follow the C-style order of typedefs with potential type annotations interposed.

---

```
typedef Strings set<string>;

typedef Month @range(1, 12) i8;
```

---

**Listing 3.5:** Examples of type definitions. Here, range is a restriction, restricting the range of possible values to [1,12]

### 3.1.2 Built-in types

SKILL features a number of built-in types, such as integral types of varying lengths, floating point types, a boolean type, a string type, and a pointer type called annotation, not to be confused with type annotations. Additionally, containers or so-called compound types, namely lists, sets, maps, and arrays can be formed on top of types, however containers may not be nested. This limitation can not be circumvented using typedefs.

---

```
BuiltIn {
  // 8-bit integer, i16, i32, and i64 analogous
  i8 integer;

  // Variable width integer
  v64 variableWidth;

  // Floating point types
  f32 singlePrecision;
  f64 doublePrecision;

  string[] strings;
  list<string> collection;

  map<string, string> lookupTable;

  bool boolean;

  annotation pointer;
}
```

---

**Listing 3.6:** Example of various built-in types

### 3.1.3 Fields

Similar to declarations, fields may be prefixed by Java-doc/Doxygen-style comments and type annotations, however some type annotations are exclusive to declarations and some to fields.

### 3 Serialization Killer Language

---

A basic, data field consists of a type identifier signifying the field's type, followed by a field identifier. All fields are terminated by semicolons. The `const` keyword can be prepended to signify a constant field, and an initial value must be assigned. Only integer type constants may exist.

---

```
Example {
  string name;

  const i32 version = 1;
}
```

---

**Listing 3.7:** Example of a user type with a string field and a constant integer field

View fields, denoted by the `view` keyword can be used to rename or retype fields directly declared or inherited from parents. For retyping to be legal, all instances of the new type must also be instances of the previous type.

---

```
A {
  A a;
}

B : A {
  view A.a as
  B b;
}
```

---

**Listing 3.8:** Examples of renaming and retyping using a view field

Finally, custom fields are a way of injecting language-specific functionality into specifications that can be interpreted by the generators for the specified target language.

---

```
Custom {
  custom java
  !import "java.lang.Object"
  !modifier "public synchronized"
  "Object" any;
}
```

---

**Listing 3.9:** Example of a custom field targeting the java generator, excerpt taken from tests specifications used to test [Fel13]

### 3.1.4 Type annotations

Type annotations can be used to denote special properties of types that allow generation of more efficient code for these special cases or limit possible values of a type to those that are meaningful for what the type models, making invalid values out of range detectable statically. There are two kinds of type annotations: hints that are passed to generators and can improve the code generated for target languages and restrictions that restrict the set of values a type can hold. Restrictions are prefixed with @ and hints with ! respectively. Type annotations may take arguments, such as integers, floats, strings, or identifiers, enclosed in brackets.

---

```
!monotone
@singleton
Annotated {
    !ignore
    @min(0)
    i32 value;
}
```

---

**Listing 3.10:** Example of a user type with type annotations and a field with type annotations

Type annotations can be applied to both declarations and fields, though some type annotations can only be applied to either declarations or fields exclusively, others can be applied to both.

Following are the descriptions of some, but far from all, type annotations to give an idea of the type of effects achievable with them.

#### Restrictions

The range restriction can be applied to integer and floating-point types and restricts the values possible for the annotated type to the specified range. It takes the lower and upper bounds as arguments and can optionally take strings which identify whether these bounds are inclusive or not. By default, both bounds are inclusive. Similarly, min and max restrictions exist, which restrict values to a minimum or maximum.

---

```
RangeExample {
    @range(1, 7)
    i8 dayOfWeek;

    @range(0, 31, "exclusive", "inclusive")
    i8 dayOfMonth;

    @min(0.0F, "exclusive")
    f32 positive;
}
```

---

**Listing 3.11:** Example of using the range and related restrictions to limit the possible set of values for fields

### 3 Serialization Killer Language

---

Another restriction type, `oneOf`, can be used to restrict the possible instances of annotation and user types to a limited number of types from a list.

---

```
Restricted {
  @oneOf(Integer, Float)
  Number value;
}

Number {
  annotation value;
}

Integer : Number {}
Float : Number {}
Double : Number {}
```

---

**Listing 3.12:** Example usage of the `oneOf` restriction

#### Hints

The `hide` hint can be used to hide field declarations from the public API. This can be lifted using renaming view fields. It also implies another hint for the field called `onDemand` which causes this field to be deserialized lazily, only when requested.

A `readOnly` hint can be used on user type declarations that does not recursively inherit from any other user types and makes it impossible to modify instances of the type. All subtypes of a type that is `readOnly` are also marked as `readOnly`.

#### 3.1.5 Name resolution

In SKiLL, names are treated as equivalent if after converting all their characters to lower case, their names are equal. Using `CamelCase` for identifiers allows generators to convert names to the established casing variant used for target languages, such as `snake_case` for Ada. The SKiLL specification also mandates that every front-end for the language must provide an option to treat underscores `'_'` as word separators instead, effectively switching from `CamelCase` to `snake_case` and ignoring single underscores between words when evaluating name equivalence. In this mode, a double underscore `'__'` can be used to escape single underscores.

### 3.2 Specification

The implementation of the SKiLL language server makes additional assumptions not stated in [Fel17]. To clearly define what specification the SKiLL language server was written against, the following marks all changes to the base specification in [Fel17] made to arrive at the specification used for the implementation.

### 3.2.1 Language grammar

- An arbitrary amount of ASCII code white spaces, tabulators, and newlines may be inserted between all syntactical elements without affecting semantics.
- The SKilL grammar defined in [Fel17] borrows integer and float literals and comments directly from the C11 standard [ISO11] and also uses C11-style identifiers and strings enriched by additional Unicode characters. In practice, the SKilL compiler [Fel13] only recognizes a subset of these elements defined by the C11 standard and these relevant subsets are used to define these elements, as they are used in practice.
  - Integer literals, unlike C11-style integer literals, do not include octal integers, but only a decimal and hexadecimal version, which may only start with `0x` and not `0X`, and also do not allow any of the `u`, `U`, `l`, `L`, `ll`, `LL` suffixes, as in the reference SKilL compiler.
  - Floating-point literals, unlike C11-style floating pointer literals, do not support hexadecimal floats or binary exponents `p`, `P` or the `l`, `L` suffixes, but additionally support the `d`, `D` suffixes.
  - String literals, unlike C11-style string literals, do not support the encoding prefixes `u8`, `u`, `U`, `L` and do also not support any escape sequences, but instead allow the backslash character `\` to be used directly as part of string literals.
- Section 4.4.5 and Appendix A of [Fel17] state contradictory definitions for custom fields. To resolve this, the grammatical definition for custom fields in Appendix A is changed to

$$\langle \text{custom} \rangle ::= \text{custom} \langle \text{ID} \rangle ('!' \langle \text{ID} \rangle (\langle \text{STRING} \rangle? | '(' \langle \text{STRING} \rangle* ')'))* \langle \text{STRING} \rangle \langle \text{ID} \rangle$$

and the definition in section 4.4.5 to

```
'custom' <id>
('!' <id> (<string>? | '(' <string>* ')'))*
<string> <id> ';'

```

accordingly, following the revision of Timm Felden. This reflects the implementation in the reference SKilL compiler.

### 3.2.2 Extended specification

As some edge cases are not accounted for in the SKilL specification, their behavior is specified here explicitly.

- The behavior for type annotations on typedefs aliasing built-in types is undefined.
- Type annotations applied to arrays, lists and sets through typedefs apply to their elements. Behavior for type annotations, other than `constantLengthPointer`, on map types is undefined.
- View fields may not alter the type of fields with a compound type.
- The behavior for a type with more than one type annotation of the same type is undefined.
- The `flat` hint may only be applied to user types and not through typedefs.

- The style guide is amended by the following guidelines, based on the style of the examples given in the specification:
  - Opening braces should be on the same line as the preceding identifier, closing braces should reside on their own line with no additional empty lines before the closing brace.
  - Type declarations should be separated by a single blank line.
  - The usage of white space is preferred over the usage of tabulators. One tabulator is equivalent to two white spaces for indentation purposes.
  - View fields should be split up into two lines: the first line containing everything from the view to the `as` keyword and the second line containing the new type and field name.
  - Custom fields should feature the `custom` keyword and generator name on the first line, followed by each option on its own line, and finally the type and field identifiers on the last line.
  - Unless otherwise specified, there should only be a single space between keywords, as well as the syntactic element `:`, and identifiers. There should also only be a single space between type names and identifiers. A single white space should precede opening brackets. Commas in container type names and type annotation arguments should be followed by a single white space.
  - Unless otherwise specified, or beneficial to readability or clarity, there should be no superfluous white space, tabs, or new lines.

#### 3.2.3 Clarifications

Some parts of the specifications may differ significantly from what one is used to for similar vocabulary in other languages or not be clear enough. The following clarifications attempt to help with their understanding.

- Type annotations encompass hints and restrictions and are unrelated to the annotation type.
- No implicit type conversions exist, meaning that integer literals may not stand in for floating-point literals. This also implies that fields with boolean type can not be assigned a true default value.
- A base type is a user type that does not recursively inherit from any other user type.
- Types inheriting from a type with a `default` restriction may be erroneous, if the default value is no longer an instance of the inheriting type.
- Tools are a concept outside of the language's type hierarchy.
- `removeRestrictions` only lifts restrictions acquired through deserialization from usually older revisions of the declaration.

## 4 Implementation

A language server for SKiL was implemented that supports most language features and is usable in many IDEs. This chapter describes the approach taken, challenges that arose during implementation, and details of the resulting language server.

Java was chosen as the programming language used to implement the language server. While this is by no means the only reasonable choice, languages capable of running on the Java virtual machine (JVM) have the advantage of being able to easily interface with the existing SKiL compiler written in Scala and being able to use LSP4J [Ecl16] as a binding for the LSP. A list of bindings can be found at [Mic16e]. The choice of Java over alternatives such as Scala is mostly down to personal preference and experience with the language.

### 4.1 Managing source files

In order for the LSP to function properly, the language client and server must share a file system. An unofficial extension that allows usage of the protocol without a shared file system also exists [Sou16a].

Files are identified by their Uniform Resource Identifier (URI) [TM05], however the URI associated with a specific file can differ between different clients and servers, such differences include the URI beginning with either `file://localhost/` or `file:///`, the existence of dot-segments, and possible capitalization differences under Windows systems.

To allow comparisons and lookups based on file URIs, URIs are first normalized to a unique representation for each file before performing the aforementioned operations. Should the language client have referred to that file using a specific URI previously, that URI is stored and used to identify the file in all responses to the client in order to avoid possible errors in the client that could arise by returning a different, normalized URI.

Generally, files are owned and managed by the language client, which monitors the file for external changes and sends the file's contents to the server when necessary. In certain cases, the server has to load files not opened in or managed by the client, such as when one file imports another file in the workspace not opened in the client or features operating on all documents in the workspace, such as a workspace symbol search, require it. In these cases, the server has to determine the content of files itself, which introduces the additional burden of determining the files' encoding, and dealing with external modifications. Here, the encoding of all files managed by the language server is assumed to be UTF-8. The LSP allows language clients to send notifications to the server whenever files in the current workspace are created, changed, or deleted, alleviating the burden of watching over modifications, given the client supports this feature.

As diagnostics should only be reported for files managed by the client, not for those managed by the server, the ownership must be saved as a property for the file.

When the client opens a file previously managed by the server, ownership must be transferred, which includes updating the file's URI and content according to the view of the client followed by reanalyzing the file.

### 4.2 Analysis

Using the LSP has certain implications for front-ends of language servers:

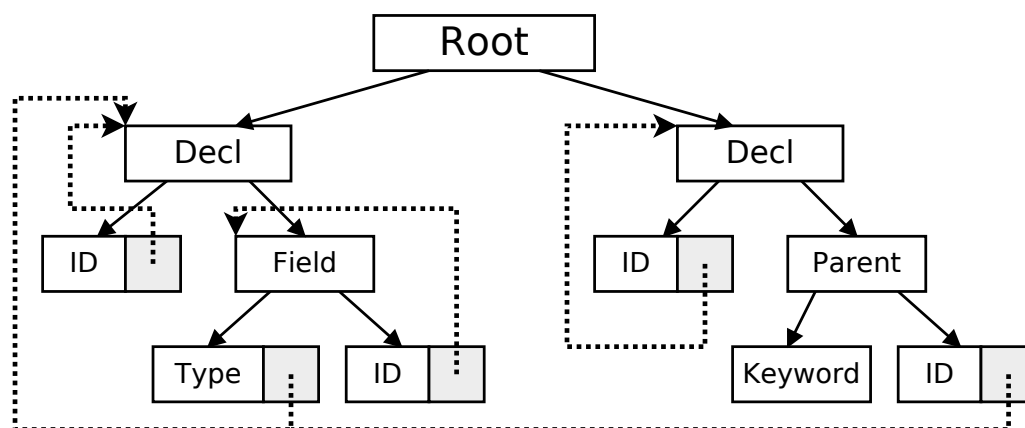
- Positions in a document are based on lines and characters, rather than an index into the file's buffer.
- For each position, the construct at that position and associated semantic information must be easily computable, to support features such as definition or hover.
- Vice-versa, for each construct, the extent as a range in the document with a starting and exclusive end position must be easily computable, to support features such as definition or references.
- The front-end should fail gracefully for incomplete or incorrect inputs. In particular, analysis should progress after errors rather than immediately terminating, meaningful error messages with associated ranges in the source should be given for each error, and incomplete structures should still be recognized and analyzed as far as possible, to enable language features such as completion.
- Due to performance considerations, the complete analysis of a document should only need to occur once ahead of time and allow efficient execution of all language features without needing to recompute syntactic or semantic information.

It is far from unusual to use the front-end of an existing compiler or analysis tool to implement a language server, as this saves development time and duplication for the analysis. This has been done for many existing language servers, including those for mainstream languages such as Java and C++ and domain-specific languages, like Ballerina or Julia. Though a front-end for SKILL is already available as part of the existing compiler, it terminates upon encountering the first syntactic error with a generic error message and does not record positional information for syntactic elements, neither can it be easily adapted to do so. It also records semantic information separate from the abstract syntax tree (AST), making the implementation of language features more difficult.

As this front-end is unsuitable for usage in a language server due to not fulfilling the mentioned requirements and can not be easily adapted to do so, a new front-end, tailored to the needs of a language server, was implemented.

The aforementioned requirements motivate an AST annotated with associated semantic information as the primary data structure, where every node on all levels also stores its extent as a range. This data structure allows efficient lookup of nodes at a position in the document by recursively descending in the AST based on node's ranges as follows:





**Figure 4.1:** Schematic annotated AST with **D**eclarations, fields, and **I**Dentifiers. Arrows indicate children, dotted arrows represent semantic annotations.

---

**Algorithm 4.1** Determine the AST node at a given position in a document

---

```

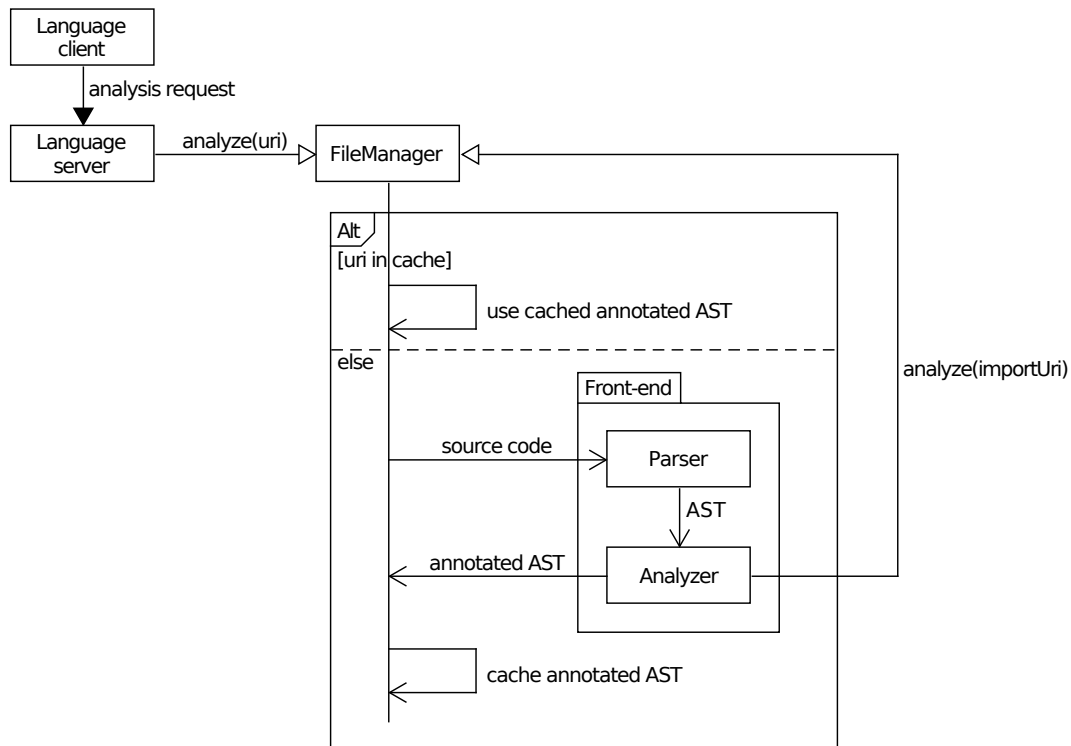
1: function GETNODEAT(rootNode, position) : Node
2:   currentNode  $\leftarrow$  rootNode
3:   lastNode  $\leftarrow$  currentNode
4:
5:   while currentNode.children  $\neq$   $\emptyset$  and lastNode  $\neq$  currentNode do
6:     lastNode  $\leftarrow$  currentNode
7:     l  $\leftarrow$  0
8:     r  $\leftarrow$  children.length - 1
9:     while l < r do
10:      index  $\leftarrow$   $\lfloor \frac{l+r}{2} \rfloor$ 
11:      node  $\leftarrow$  children[index]
12:      if position < node.startPosition then
13:        r  $\leftarrow$  index - 1
14:      else if position  $\geq$  node.endPosition then
15:        l  $\leftarrow$  index + 1
16:      else
17:        currentNode  $\leftarrow$  childNode
18:        break
19:      end if
20:    end while
21:  end while
22:
23:  return currentNode
24: end function

```

---

Given a position in the document, one determines the lowest level node at that position by first setting the current node to the root node and then checking if any of the current node's children contain the position. All child nodes' ranges are disjoint and children are sorted in ascending order, enabling this to be done through binary search, though in practice, nodes often have few children and range checks are cheap so that this usually does not yield a significant speedup. Should such a child be found, the current node is set to that child and the search is iterated. Otherwise, the current node is returned as the most concrete node for the position.

Using the AST and position-based lookups within it, many language features can be implemented with only minor additional logic, making the front-end the core component of the language server. One example of this would be the definition feature, which can be implemented by looking up the node at the specified position and following the associated pointer to the node which marks its definition and returning its extents.



**Figure 4.2:** Schematic overview of the analysis performed by the language server

The annotated AST is computed in two main phases: First, the AST with all nodes and associated ranges is constructed by a parser through syntactic analysis of the source code where syntactic correctness is checked, then, the AST is annotated with semantic information during multiple traversals and semantic correctness is verified.

### 4.2.1 Syntactic analysis

To construct the base AST, the underlying source code must be syntactically analyzed, a node created for every syntactical element, and the nodes be built up into a tree according to the language's grammar.

This is done by a parser. Here, the lexer or tokenizer is viewed as part of the parser.

SKILL has an LL(1) deterministic context-free grammar [Fel17, section 2.2], which means it can be efficiently parsed in a single pass with a lookahead of 1 and is free of left recursions.

A time-efficient way of implementing parsers is to leverage parser generators, such as ANTLR [Par17], JavaCC [Mic17], or others, that can generate parser implementations in target languages based on a typically tool-dependent specification of the language's grammar.

This way of implementing parsers has the advantage of being able to change the language's grammar and corresponding implementation easily and the ability to implement a reasonably efficient parser without too much manual work. It comes at the cost of having less control over the exact implementation as well as the AST and node layout. Additionally, it may complicate error handling and the way incomplete and incorrect constructs are represented in the AST.

---

**Algorithm 4.2** Parse a user declaration. `getCharacter` returns the character at the current position and `advance` advances the current position by one.

---

```

1: function PARSEDECLARATIONUSER : NodeDeclarationUser
2:   startPosition ← getPosition()
3:
4:   comment ← parseComment()
5:   identifier ← parseIdentifier()
6:   parents ← parseParents()
7:   if getCharacter() = '{' then
8:     advance()
9:   else
10:    error("Expected '{'")
11:  end if
12:  fields ← parseFields()
13:  if getCharacter() = '}' then
14:    advance()
15:  else
16:    error("Expected '}")
17:  end if
18:
19:  nodeDeclarationUser ← NodeDeclarationUser(comment, identifier, parents, fields)
20:  nodeDeclarationUser.setExtents(startPosition, getPosition())
21:
22:  return nodeDeclarationUser
23: end function

```

---

Alternatively, a parser can be implemented manually. For the SKiIL grammar, which is LL(1), a recursive descent parser is a sensible choice. It is a type of top-down parser that can be implemented through functions, each of which parses a single nonterminal and returns a node for it by reading the next character (lookahead of 1), making decisions based on its value, and then consuming it or calling other functions to parse nonterminals. Whenever another nonterminal is parsed, its node is added as a child of the current nonterminal being parsed. This procedure builds up the AST through recursion and the AST for the entire document can be parsed by calling the function parsing the root node. An example for a function parsing a user declaration is given in algorithm 4.2.

This approach gives more fine-grained control over the parsing process and the layout of the AST, simplifying the logic for handling incomplete and incorrect inputs and data structure access during semantic analysis and language feature execution. In part, it is enabled by the simplicity of the SKiIL grammar and while it takes more effort to adjust the implementation for a changed grammar, the SKiIL specification has only changed marginally over the span of its lifetime of several years, making one of the approach's disadvantages less of an issue.

In practice, it is sometimes useful to look ahead more than one character for convenience of implementation, such as when making a decision based on a keyword identifier with a length longer than 1, or in order to avoid splitting up non-terminals with potentially common prefixes to keep AST nodes representing language constructs directly, or to perform error recovery, such as determining the position of the next opening brace after a closing brace was omitted. The latter application could lead to an arbitrarily large lookahead, but is limited to the length of the document in the worst case, which is tolerable in practice.

Despite these advantages, using parser generators is by no means an unviable approach to the problem.

Early in the development cycle an idea, inspired by the LSP allowing only modified ranges of a document to be transmitted rather than the document in its entirety, came up to only parse the altered parts of the document and re-use the previous AST for all nodes whose representation in the code was unaltered. This could be accomplished by identifying the largest node affected by a change, such as a type, a field or an entire declaration, for each altered range and re-parse it accordingly, inserting it into the otherwise unaltered AST. The idea was ultimately scrapped, as the effort associated with supporting such partial analysis was considerably large and the gains in analysis speed were deemed mostly irrelevant. This stems from the fact that even large specifications with many declarations could already be parsed quickly enough, which is supported by the results in section 5.2.

Apart from building up the correct AST for well-formed specifications, the parser for a language server should also handle incomplete constructs with valid prefixes as well as syntactically incorrect constructs gracefully, in that the AST for incomplete constructs matches that of the AST of completed constructs, where all nodes whose representation in the source is completely missing and those that are partially missing, but could be completed to more than one type of node, are pruned. Here, completion refers to insertion not only at the end, but also in the middle of a construct.

This allows language features, particularly completion but also formatting and other features like renaming, to work as well as possible even when parts of the input are missing or incorrect.

### 4.2.2 Semantic analysis

After the parser has built up an AST, it is annotated with semantic information, such as references or definitions, and checked for semantic correctness.

SKiLL supports cyclic dependencies through includes and ill-formed specifications might feature cyclic inheritance, which still has to be dealt with gracefully. Not only does this complicate propagating errors through includes, but necessitates multiple passes during analysis and careful ordering of analyzing imported files and parent types.

In the implementation this is done in two main steps for each file, where first all symbols declared in that file, which are computable with an unannotated AST, are recorded, then all imported files are analyzed recursively and finally, all remaining semantic analysis can be done given that of all of the imported files.

The analysis begins from an arbitrary root file and works its way up and back down through all imports, yielding annotated ASTs and diagnostic messages for each visited file. After traversing all declarations and recording the normalized identifier name and associated node for each declaration and their fields, analysis is recursively done for all imported files. Then, all types and other references in the current file can be resolved with the information from included files.

As dependencies can be cyclic, the requested files during an analysis run are recorded and requests for a file that were previously requested, result in returning a reference to the previous instance rather than triggering analysis anew, similar to the functionality of include guards, but without the need to load or analyze the file again.

Care has to be taken as to produce a deterministic output of semantic annotation and warnings or errors, so that they are the same no matter what root file the analysis started from. Otherwise, opening files in a different order could alter the result of the semantic analysis which is not desirable.

To check validity of the specification, additional semantic checks have to be performed, such as ensuring the absence of nested container types and inheritance from multiple user types or inheritance from oneself or built-in types.

## 4.3 Language features

### 4.3.1 Overview

Most language features that can be supported for SKiLL specifications, not including features like color or link support that are not relevant to language, were implemented in the SKiLL language server. Namely the language features diagnostics, definition, hover, highlight, references, symbols, rename, completion, formatting and range formatting, workspace symbols, and folding range were implemented.

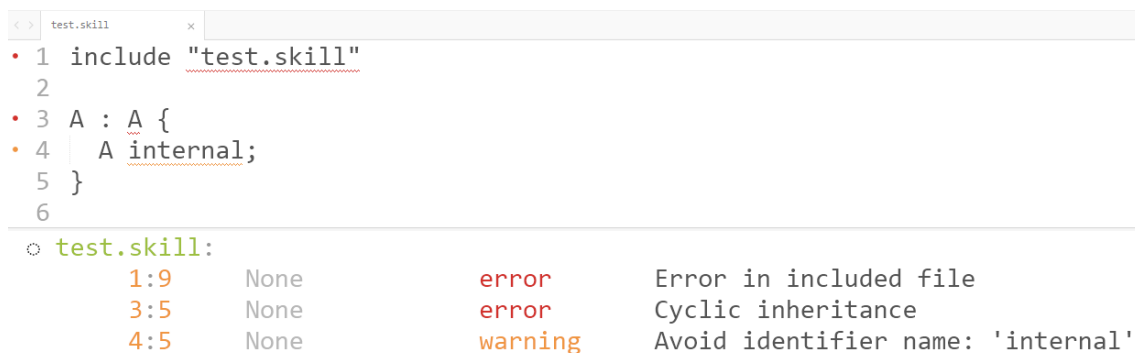
The following subsections go into detail about which information and functionality has to be present to support each of these language features, in particular for SKiLL, and what challenges this poses as well as how they can be implemented in general and how they were implemented for the SKiLL language server.

### 4.3.2 Diagnostics

The analysis records all syntactic and semantic errors as well as various warnings, such as using identifier names that are reserved in target languages, for a given file with the associated range. These diagnostics are published for all files managed by the language client.

Errors are propagated through imports, to indicate when files are erroneous due to their dependencies, even if all of their declarations are correct.

It is useful to output diagnostics sorted by ascending start position. As errors can come from several sources in multiple passes in the semantic analysis or the syntactic analysis, it is easier to record all diagnostics and subsequently sort them rather than continually ensuring this condition.



The screenshot shows a code editor window titled 'test.skill' with the following code:

```

1 include "test.skill"
2
3 A : A {
4   A internal;
5 }
6

```

Below the code, a diagnostics panel is visible, listing the following errors and warnings:

Range	Severity	Message
1:9	error	Error in included file
3:5	error	Cyclic inheritance
4:5	warning	Avoid identifier name: 'internal'

**Figure 4.3:** Diagnostics in Sublime Text 3

The choice of the range associated with a diagnostic is not always a clear one. Diagnostics can only be displayed at ranges that are present in the analyzed document and should preferably not extend over white space or multiple lines.

An example of an unclear situation is the following: the terminating semicolon for a field was omitted. This error occurs during syntactical analysis, where the AST is still being built up and is not available in its entirety yet. The omission could be indicated by returning an error with the associated range beginning at the next possible non-line break character with a length of 1. This represents the position the semicolon was expected to be and the position can be trivially computed without the need to store additional positions or computing the last node not omitted in the source. However, this strategy fails when there is no next character to be found when the end of the document has been reached, as some IDEs fail to display ranges starting at the last character in the document. Another option is to return the last non-white space position before the semicolon, though this may require determining the last non-omitted node in the construct and may not ideally convey the omission of a semicolon. A potentially better range to return for this error might either be that of the field's identifier, but it may also have been omitted, or the entire range for the field, which has the issue of making it difficult to tell if there are more diagnostics in the same range.

---

```
Person {  
    string firstName  
  
    string lastName;  
}
```

---

**Listing 4.1:** Example of two different ranges that could be associated with a missing semicolon for the first field

During semantic analysis and the verification of semantic rules, the range for indicating where in the source rules were violated are also not always clear. If the violation can be traced back to a single offending node unambiguously, the choice of the range for the diagnostic is simple. For example, this is the case for user types which inherit from themselves, where the offending node is the one specifying itself as a parent. In other cases this is less clear: When restrictions are violated, such as a range or related restriction being applied to a constant field, the assigned value is deemed offending when it is not allowable due to the restriction rather than the restriction itself. Repeated or redundant type annotations are indicated at the positions encountered later by the analysis.

### 4.3.3 Definition

The definition feature can be used to follow all referenced declarations, such as in type expressions or as arguments to type annotations, field references, as are used for the view field type, and for included files to allow easy traversal into imports. There is a related type definition feature, that only follows references in the case of declarations referenced as types, but not fields or includes.

The definition feature can be implemented directly through looking up nodes in the annotated AST. Given the goto definition request from a client with the document identifier and the cursor position, one can use the routine *getNodeAt* defined in algorithm 4.1 to look up the lowest-level node in the document's AST at that position, such as an identifier, and follow the pointer to the defining declaration available after semantic analysis to return the starting position for that declaration to the client.

The process of identifying the defining node for the construct at a given position is a common operation used by other features such as hover, reference, highlight, and rename. As such, it is extracted as a subroutine reusable for other language features.

### 4.3.4 Documentation

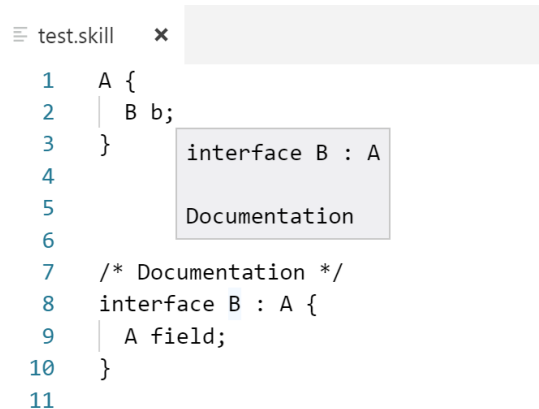
Some language features, such as hover and completion, can include documentation associated with a type or symbol for display in the language client.

It is therefore useful to be able to look up the documentation associated with a certain node. This can be done easily by following the node's pointer to the definition of the symbol and extracting the types and documentation accessible from the definition.

Additionally, a symbol kind can be provided in some contexts, which range from functions over local variables to fields and classes and can be directly deduced from the node's type.

### 4.3.5 Hover

The hover feature can return documentation associated with an element as well as an optional range that is highlighted upon hovering and is usually shown as a pop-up upon hovering over an element. Its implementation can be built based on that of the definition feature by first determining the defining node for the given cursor position and then generating a documentation string as described in the previous section, then returning the documentation and the range associated with the defining node.



**Figure 4.4:** The hover feature in VSCode

### 4.3.6 References

The reference feature can be used to return all references in the workspace to the symbol at a given position. This requires loading all specifications in the current workspace ahead of time. The feature can be implemented by identifying the defining node for the node at a given position and then returning all references to it. All references to a declaration are computed during the semantic analysis: whenever a type identifier is resolved, a reference to it is added for the declaring definition resolved to.

### 4.3.7 Symbols

The symbols feature allows all symbols declared in a document to be listed, optionally with a container they were declared in, such as a class, and a symbol kind, such as a field, an enum value, or a class. It is often requested by clients directly after opening a file and is used to show an outline of the declared symbols and containers in the file.

The feature can be implemented by enumerating all types declared in the specification, and their fields, if available, while emitting symbol information containing their range, kind, and for fields their container. Fields that are not declared in a type definition but only inherited were chosen not to be included, as this may make the list of symbols crowded and the range associated with inherited symbols lies outside of the type declaration they are contained in and could have been declared in a different specification entirely.



### 4.3.8 Rename

The rename feature can be used to rename a symbol and all its occurrences in the workspace. It can be implemented by determining the declared symbol at the given position, similar to the definition feature, and getting all references to the symbol in the workspace to finally send text edits renaming all occurrences appropriately.



**Figure 4.5:** Example of the rename feature in Eclipse

Future versions of the LSP will additionally add a `prepareRename` request that can be used to determine whether there is a renameable symbol at a given position and if so, compute the range of that symbol.

### 4.3.9 Formatting

The SKiLL specification provides a style guide [Fel17, section 2.4] with recommendations and best practice rules for writing skill specifications. This style guide is extended by section 3.2.2.

Some of the specified rules, such as naming guides for identifiers with camelCase or PascalCase and alphabetic characters can be implemented as warnings when identifiers don't match these rules. Others, such as indentation and recommendations for line breaks can automatically enforced by formatting a document.

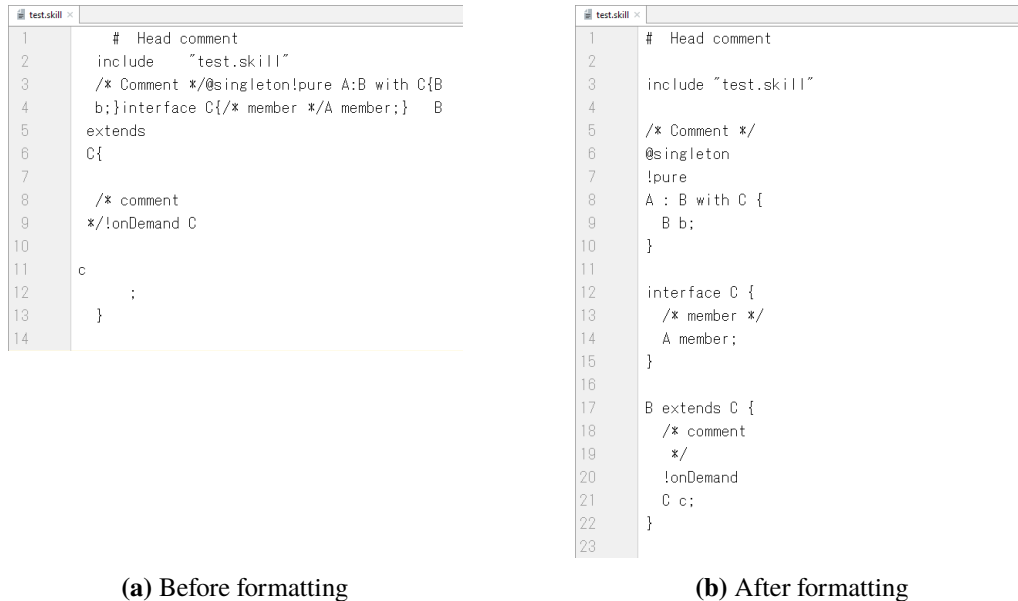
The style guide goes on to mention rules, which can not easily be captured by an automatic tool, such as using speaking names for identifiers, or breaking up code into logical units. These rules of software engineering have to be left up the authors of specifications, as current tooling has difficulties with implementing these.

The guide also recommends declaring super types first wherever possible, which can be ensured by automatic formatting through topological sorting, but alters a document beyond spaces and newlines and may not always be desirable.

Finally, it includes rules easily encompassed by formatting, such as how white spaces and new lines should be used for different constructs.

Formatting can be implemented by recursively formatting the root node, where formatting is specialized for each node type and nodes recursively invoke formatting for their child nodes. This is akin to the recursive descent parser using functions, the traversal following the same pattern.

## 4 Implementation



**Figure 4.6:** Formatting in IntelliJ IDEA

In general, formatting should never change the semantics of a program, which can be captured by the stronger condition of not changing the resulting AST, additionally preventing re-ordering, or even more strictly, by forbidding changes to characters other than white spaces, tabs, and newlines between syntactic elements, exceptions to this may include alternatives such as using or omitting braces for nested constructs or omitting unnecessary semicolons or trailing commas in some languages, though these do not apply to SKILL due to its grammar. Thus, a desirable quality for automatic formatting is that if two documents have equivalent ASTs, the formatted output for them should be the same. For SKILL, this implies that the only differences between the unformatted input and formatted output may be in white space, tabs, and line breaks between syntactical elements. This quality, or a weaker superset, is easily verifiable through testing and fuzzing.

An issue with this definition lies in the equivalence of ASTs, which is not clearly defined. Though ranges and positions of nodes should not be a criterion for differentiation, the contents of the document for that range may well be. This is the case for identifiers, incomplete keywords, and comments.

Multi-line comments are a special case, as it may or may not be desirable to break their contents differently across lines. Further, special rules for indentation and treatment of leading asterisks, in Javadoc-style comments, can be applied to comments.

The LSP also allows range formatting, which formats a specified range rather than an entire document. Given the implementation for formatting an entire document as was described previously, one can support such range formatting by performing the same traversal and formatting, but only writing formatted output to the buffer if the node being formatted is completely in the given range. Nodes that are not completely, but partially, inside the range can be copied verbatim from the input to yield correct formatting for the subset of the input specified by the given range.

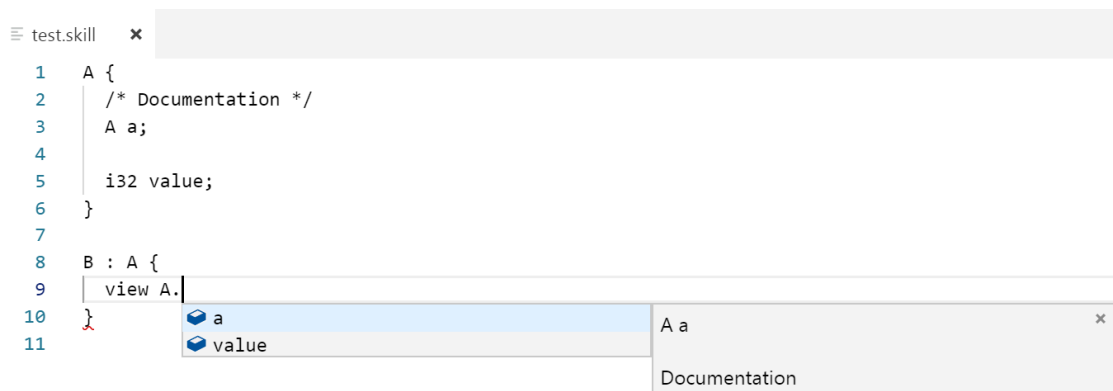
As with many of the language features, the question of how to deal with incomplete or incorrect inputs when formatting is both important and non-trivial. For incomplete inputs, which can be identified as the pruned versions of their complete inputs, the formatted output can easily be deduced to that of the complete output, removing any parts also omitted in the input. Other invalid constructs, for which this can not be done, prove to be more of a challenge. One could either not format a document containing these at all, format all content before the invalid constructs and stop formatting and use the original content upon reaching the first occurrence of these, or attempt to skip the invalid constructs based on the AST and format all valid constructs following them, only replacing the invalid constructs with their original contents. The last option may yield undesirable results, due to incorrect representation of the user's intended construct in the AST.

### 4.3.10 Completion

Completion is the most involved language feature implemented, requiring special handling for all constructs in the language. It also features the least straight-forward goal, as good completions or suggestions are often hard to define exactly, making optimization towards them difficult.

The choice of representation for incomplete constructs in the AST simplifies completion greatly and is a large part of the reason this representation was chosen.

Simple suggestion possibilities include suggesting a list of types or symbols when the syntactical context requires this, which can be derived through the AST. This can be further refined through the larger syntactical context and semantic information available, such as only suggesting integer types for const fields or restricting suggested types and fields for view to those of parents and their respective fields. Similar inference can be done for the arguments to hints and restrictions.



**Figure 4.7:** Screenshot showing the completion feature being used for a SKILL specification in VSCode

Even with these restrictions, the list of suggestions can grow to excessive lengths, including all types or symbols visible recursively through dependencies, overwhelming the user and diminishing their usefulness. This can be combated when a small prefix of the desired symbol or type name is already known, cutting the large list down to a more manageable size, often going down to a single

element with a prefix of even a few letters. Such a prefix can be provided by the user by typing it in before or while using the completion functionality and is usually known by the user at the time of writing the specification.

Though this is a possible workflow through IDE support, which is usually present, by invoking the completion functionality and writing such a prefix, it can also be implemented in the language server when completing an incomplete identifier. After feedback that long lists diminish the usefulness of the feature, completions for types and symbols were restricted to completing identifiers which already have a prefix, thereby generally avoiding the problem of very long lists. Keeping this functionality even without prefixes present does not seem unreasonable though and adds the option of exploring all possible continuations at a certain position in the document.

It is also possible to further take identifier names and surrounding context into account to guess which completions are more likely to follow, but such advanced attempts to understand the logic of the specification are very much heuristic based and often inaccurate on top of taking significant amounts of effort to implement.

The top-down implementation approach using functions for each non-terminal also used for parsing and formatting can once again be utilized here. In this case, nodes whose ranges do not contain the cursor position and are not adjacent to it need not be considered and the process can be exited early once certain completions have been found. This is similar to the lookup of the corresponding node for a position adding the condition of adjacency, but may terminate before reaching a node with no children or continue even after a node with no children has been visited in order to enable considering more context and possibilities, as more than one node could be completed at a single position, due to missing nodes caused by incomplete constructs or being able to complete both the node before and after the current position.

Additional feedback yielded that users generally want to complete incomplete elements before the cursor position, rather than getting suggestions following an incomplete construct, and suggestions for insertion after the cursor position did not fit their mental model and were undesirable. As a result, suggestions are now not given for constructs after the cursor position if there is an incomplete construct before it.

### 4.3.11 Automatic compilation

Compilation of code is part of most programmer's workflow: Code is commonly written, compiled, and then run, at which point the output of the program may prompt starting the cycle again by writing more code or altering it. Many IDEs support this workflow by allowing compilation to be easily invoked for the currently opened documents, or even automatically invoking compilation, when appropriate. For SKiL, compilation means that bindings for target languages are generated. Incorporating compilation and automatic compilation into the SKiL language server allows target bindings to be continually updated while work is done on SKiL specifications.

To execute compilation, the skill compiler [Fel13] is invoked with appropriate arguments. Determining the appropriate arguments for invocation requires a project-like configuration, as is described in the next section, that determines the main specification and other potentially target language-specific options.

Using the configuration, all files in the include hierarchy of the main specification are automatically compiled upon saving them. Additionally, there is a command available to compile the configuration for the current workspace.

### 4.3.12 Configuration

Though the LSP allows configuration options to be set for language servers, these configuration options are independent of the workspace and can not be used as a project configuration. Further, how these options are saved and accessed differs between language clients, which makes this unviable as a portable solution for project configurations.

Instead, `.skil` configuration files with the format of Java `.properties` files are used, similar to clang-format [LLVa] configurations. The configuration for a directory and the files contained in it is determined as follows: The existence of a `.skil` file is checked for the current directory and then iteratively for its parent directories until the root directory is reached. The first configuration file found is used to determine the configuration options. As a fallback, the configuration file in the directory of the language server executable is used to fill all options not specified in the previously found configuration. This allows setting defaults for the language server in addition to project-specific options.

The options available include treating underscores as a word separator, as required by the SKiL specification, a command for the SKiL compiler, and an option for the main specification of the directory.



# 5 Evaluation

## 5.1 Testing

Sufficiently complex code can not feasibly be riddled from all bugs and incorrect behavior. However, common scenarios and inputs representative of the most common use cases of the software can be tested to assure their, at least apparent, functionality. Apart from running the software, going over different scenarios for all of its use cases and watching for incorrect behavior, which is a part of quality assurance, automated testing methods can be employed to quickly test the current state of software without any human intervention. This comes at the advantage of requiring significantly less effort to run tests and allows them to be executed repeatedly, such as running continuously after all changes to the software. For example the language specification might change and code in the front end may need to be adjusted. Barring changes that modify correct test results, if much of the functionality is unchanged, the new implementation with all changes can be easily tested and either be used to find newly arisen errors or increase the confidence in the correct functionality of the program.

Testing, both manual and automatic, can not be exhaustive due to the large space of possible scenarios and inputs and the difficulty of determining the correct output for an arbitrary input, which is equivalent to the task of the code under test. It can however increase the confidence in the correctness of code and reasonably ensure correct behavior for commonly performed scenarios.

To test the SKiLL language server during development and to check the basic functionality, such automated methods of testing were employed in addition to manually testing the language server in several IDEs. In the following sections, two different, but related, approaches of automated testing used to test the SKiLL language server are presented.

### 5.1.1 Unit tests

Unit tests are an established way of automated testing. They are manually written by programmers and check the correct functionality of an interface or a piece of code by choosing representative inputs and checking the output of the code under test against a predetermined expected result to cover the space of possible inputs with a test input for each equivalence class determined by the programmer.

The highest level interface for the SKiLL language server, as used in practice, is that of a language server through a communication channel, such as the standard input and output. Though this interface should be tested, as it is the one used by language clients in practice, the communication at this level is largely the responsibility of the underlying binding for the LSP, in this case the LSP4J library. The next-lowest interface is implemented by the language server directly as methods

implementing the different features and calls defined by the LSP and brought into Java by LSP4J. Further down, the front-end and various other components, such as a formatter and utilities for handling files and other utilities, can be tested.

The majority of tests are written against the language server, without the overhead of communication through a channel, as it is nearly identical to the interface used by language clients, but comes without the additional overhead of serialization, deserialization, and transmission of messages through a channel. There is a test case interfacing at this high level interface to cover tests for connection and transmission of messages. There are further tests written against smaller components, such as the front-end and the formatter, to test these smaller components in isolation while exposing more details, allowing to test for more correctness conditions and enabling to differentiate between errors in the front-end and errors in the implementation of language features.

Concrete examples of units tests include:

- Well-formed and sometimes syntactically, but mostly semantically, ill-formed specifications that need to pass without any errors or with errors respectively. They are largely taken from the front-end tests of the SKiLL compiler and provided a starting point for conformance to the SKiLL specification and also served as an indication of progress and correctness when implementing more and more of the SKiLL language constructs in the front-end.
- Pairs of unformatted inputs and formatted outputs that were used to test the formatter implementing the functionality of the formatting feature.
- Tests for specific language features, such as definition, hover, references, and others that use simple predetermined specification files to test the actual results, such as ranges and documentation, against the expected results. A concrete example of this is invoking the definition feature at a position of a declaration reference in the code and comparing the result to the known range of the associated declaration.
- Tests for utilities used, such as tests for comparing positions for being larger, equal, or smaller as well as tests for the file-handling utilities.
- Regression tests, derived from errors found through testing. They were largely found through fuzzing, as elaborated in section 5.1.2.

Additionally, all tests mentioned also implicitly check for any internal errors or exceptions occurred in the front-end or language server.

Using unit tests, a test coverage of over 80% was achieved. This was measured using JaCoCo [jac13]. The code not covered by unit tests primarily consists of code for handling internal errors and exceptions, that should not occur unless the language server exhibits incorrect behavior and are not recorded by the code coverage tool used for measurements, and code that is not easily testable, such as the main class used to start the language server or parts of the file manager that were mocked out.



### 5.1.2 Fuzzing

Another automated method of testing is Fuzz testing, or fuzzing. As with unit testing, fuzzing is not exhaustive and will likely not find all inputs that cause incorrect behavior or even be able to detect all incorrect behavior, but has the advantage of exploring a much larger space of inputs than possible with unit testing automatically. Unlike brute-forcing or enumerating all inputs, fuzzing can be guided by metrics chosen freely to improve the runtime and more effectively find relevant inputs. The most common metric used is a form of code coverage, usually already available through tooling in the compiler or external tooling. After every input is evaluated, the coverage is recorded and used to influence the generation of following inputs to maximize coverage. Other automatically measurable metrics, such as stack size or memory usage can also be used and allow finding different types of bugs more effectively.

Fuzzers have successfully been used on compilers and front-ends, including GCC and clang, before to uncover bugs.

A downside of fuzzing is that unlike programmers writing unit tests, it does not have any understanding of the structure of inputs, thus finding relevant inputs for heavily structured data may take time. It would be possible to combine fuzzing with symbolic execution to steer inputs further, but this done only more rarely as it is much more complex and brings other issues with it.

Instead, a set of meaningful inputs, as a so-called corpus, can be provided by programmers that is then used as the basis for generation of inputs through mutation and combination.

Fuzzing is not a replacement for unit testing, but rather another element in the cycle of testing. After unit tests have been written, the inputs created for unit tests can be compiled into a corpus. Using fuzzing, new inputs are generated based on the corpus, expanding it by inputs that increase overall coverage for the code under test. Should new inputs that invoke incorrect behavior be found through fuzzing, a regression test case can be added based on the input. Inputs that increase coverage or are new in another way that is not covered by the existing unit tests are also candidates for the creation of new test cases.

Fuzzing can be used to test a wide variety of conditions for all examined inputs. Often, the absence of internal errors or exceptions is used as one such conditions. In languages with appropriate tooling, fuzzing can be combined with sanitizers effectively checking a variety of correctness conditions.

A problem that occurs for more structured input, such as code, is that input creation and mutation strategies often operate on bytes, which is not a model very well representative of the input. It is possible to create mappings that allow the usage of different base constructs, such as syntactic elements, and to use custom mutation strategies operating directly on such structured data, such as ASTs.

Directly interpreting bytes as characters can still be used to detect anomalous cases well and is reasonably effective for other types of errors, but could be improved by a representation closer to the nature of typical inputs. This direct mapping causes code inputs to be syntactically ill-formed in the vast majority of cases, additionally making verification of program's legality according to the specification and the front-end infeasible and shifting the focus on internal errors and exceptions occurring in the front-end. As checking whether a program is well-formed for an arbitrary input is equivalent to the task of the front-end, verification of the front-end against the specification should instead be done through unit tests.

Well-known implementations of fuzzers include AFL [Zal13] and libFuzzer [LLVb]. To fuzz code running on the JVM, Tribble [Sel17], a code coverage-based fuzzer, was used to fuzz test the SKiLL language server.

Fuzz testing of the SKiLL language server has revealed various errors, such as anomalous cases like a file containing only a single opening brace or forward slash, as well as cut off or otherwise incomplete inputs. It also detected incorrect formatting and missing ranges for a node type recently implemented for which setting the associated ranges was forgotten.

Conditions that were checked for inputs include the absence of any internal errors and exceptions in the language server, all nodes in the parsed AST, except for comments and strings not containing any white space, tabs, or newlines, and language features such as definition, formatting, or completion executing without any exceptions in a small amount of time. Additionally, both the unformatted input and the output of the formatter are stripped of all white spaces, tabs, and newlines and then compared to ensure that formatting does not create, move, or delete any characters other than white space.

It would also be possible to fuzz the front-end of the SKiLL language server against that of the existing SKiLL compiler, checking whether they agree on the validity of the supplied specification. This could uncover discrepancies in treatment between the two front-ends in addition to errors in both of them.

### 5.2 Performance

Performance optimization was largely disregarded during the implementation of the SKiLL language server in favor of readability and ease of implementation. The resulting implementation was still claimed to be performant enough for all practical uses. This claim is investigated and supported by the following measurements.

All measurements were performed on a system with a single i7-2600k processor and 16 Gigabytes of RAM using Java 8 through the HotSpot JVM on Windows. It is not unreasonable to assume a typical system of a programmer to have comparable or better hardware.

A common optimization technique used by JVM implementations is just-in-time compilation, where Java bytecode is first interpreted directly and if certain pieces of code are executed repeatedly, they are compiled natively and executed directly instead of being interpreted. As benchmarks and especially microbenchmarks generally run a piece of code repeatedly in a loop to determine its average runtime, most iterations of such a benchmark will run optimized native code as opposed to Java bytecode. This may not be completely representative of the performance of an application, such as the SKiLL language server, especially directly after startup. Despite that, such benchmarks are still useful for determining the relative performance of two or more pieces of code and can resemble the performance of an application after warmup. Elapsed time was additionally measured for only a single iteration directly after the startup of the JVM. These measurements may be less accurate and inflated, but are better estimates of upper bounds on the actual runtime as experienced by users.

Operation	Execution time	Operation	Execution time
Syntactic analysis	4.03ms	Syntactic analysis	0.33ms
Semantic analysis	11.58ms	Semantic analysis	3.89ms
Open file	18.71ms	Open file	4.36ms
Formatting	2.54ms	Formatting	0.05ms
Definition	6.73ms	Definition	0.01ms
Hover	15.48ms	Hover	0.01ms
Completion	22.44ms	Completion	0.01ms

(a) Results for a single iteration, yielding inflated, less accurate results, but are a better approximation of upper bounds as experienced by users

(b) Results for many iterations, yielding more accurate and comparable results, but are further removed from actual usage patterns of the language server

**Table 5.1:** Benchmark results for various operations of the language server

Table 5.1 shows the results of core operation and exemplaric language feature runtimes. Opening a file encompasses analyzing it syntactically and semantically as well as sending diagnostics to the language client. The benchmark was performed on a large SKiL specification (over 14000 characters long, containing over 50 type declarations) and language features requiring a position were supplied with a meaningful position for the feature towards the end of the document, nested at least two levels deep, to estimate an upper bound for realistic inputs.

The approximated upper bounds on runtime lie under 16ms for most and under 32ms for all operations when measuring a single iteration. After warmup, all operations other than analysis of files take less than 0.1ms and file analysis is done in under 5ms. As a single frame is displayed for over 16ms at the common refresh rate of 60fps, users should see the result of all requests displayed often within a single frame and after a few frames at worst, even accounting for additional time taken by IDEs, which is deemed to be responsive enough for all requests and language features.

As analysis, in particular semantic analysis, is the most time-consuming operation, loading many specifications, with potentially many imports, such as when all files in a workspace are first analyzed, might still cause unwanted delays on startup. This could be improved by allowing independent specifications to be analyzed in parallel and loading files in the workspace not currently open in the IDE asynchronously. With the set of test specifications from [Fel13], which are not highly interdependent, the entire startup process never took longer than a few seconds.

### 5.3 Task fulfillment

The task posed for the implementation of the SKiL language server was to support at least the language features definition, completion, and formatting as well as demonstrating its functionality in at least two IDEs.

The features definition, completion, and formatting were implemented and the optional goal of supporting additional language features was also met with the implementation of the diagnostics, hover, highlight, references, symbols, rename, range formatting, workspace symbols, and folding range features. The other optional goal of providing a convenient way to invoke the SKiL compiler for the active project was also met through automatic compilation on save.

## 6 Conclusions and future work

### 6.1 Conclusions

SKiL can be used to serialize and deserialize even large amounts of data independent of platform and target language, with generators for bindings available in many target languages that allow idiomatic integration of handling this data with associated documentation.

In the process of examining the SKiL specification to determine the behavior of the language server, several small issues and not maximally clear statements were identified, for which improving changes and clarifications were presented in section 3.2.

Prior to this work, the tooling available for programmers wanting to write SKiL specifications with IDE support was limited to skilled [Stu15], but not available for any other IDEs nor easily extensible to them. The LSP was seen as a possible solution for the problem, but it was not clear whether it was suitable for the task of support language analysis features for domain-specific languages and what approach should be taken for its support.

The LSP was identified to be a viable solution to the task of implementing language analysis features across multiple IDEs already with good language support and lacking, but growing, IDE support. It covers many language-specific analysis features and allows them to be implemented with existing front-ends or analysis tools with a reasonable amount of effort. It does not come without its problems, such as missing features, like syntax highlighting, and insufficiently specified details, especially for integrating language servers into clients, resulting in additional code having to be written for different IDEs in the worst case. Despite this, most functionality is completely IDE-agnostic and the code that has to be written for specific IDEs is usually very simple and contained, meaning that this effort still dwarfs the alternative of integrating language analysis features into multiple IDEs, each with their own concepts and APIs, manually.

A language server for SKiL was developed that supports most language features currently available in the LSP that are relevant to SKiL and can be used in multiple IDEs supporting the LSP. Rising adoption of the LSP in IDEs and the improvement of language client implementations in the future will further improve this support without a need to modify the language server. The SKiL language server includes a front-end that handles failure gracefully and shows meaningful error messages even beyond the first syntactic or semantic error.

It is an attractive option for programmers writing skill specifications, enabling them to work on SKiL specifications alongside code in target language in an IDE of their choosing, while making use of language analysis features to support their work.

## 6.2 Future work

The SKiLL specification could be revised to incorporate solutions to the issues identified in section 3.2.

The LSP could and will most likely be expanded to include missing features, such as syntax highlighting. It could further be adjusted to address the issues with lacking specification of communication channels and integration of language clients amongst other issues that cause IDE dependent behavior to emerge.

Existing language client implementations could be extended to support all features the LSP has to offer and could improve the usability and ease of installation for language servers. Further, IDEs that do not support the LSP yet could benefit from gaining a language client implementation.

The SKiLL language server could be improved by ideas mentioned in chapter 4, especially the completion feature as a more complex could benefit from further adjustments. Features such as code lens, code action, and commands could be expanded upon to support automatic solutions to warnings and errors or allow generation of commonly used snippets automatically, such as automatically generating a view field renaming or retyping a highlighted field. Though not deemed especially critical here, the performance of the language server leaves room for improvement, especially as this was not deemed to be a main concern. In particular, the analysis of files could be done asynchronously and in parallel for independent files.

The existing compiler for SKiLL could be augmented with the features the language server's front-end possesses, improving returned diagnostics.

A user study could be conducted to investigate potential productivity gains achieved through usage of the language server and uncover outstanding usability and functionality issues users have with it.

## List of Figures

2.1	Exemplaric messages sent between the language client and server, based on the first figure in [Mic16d]. A more detailed overview of exchanged messages including concrete parameters can be found on the LSP Inspector page [Mic16c]. . . . .	9
4.1	Schematic annotated AST with <b>D</b> eclarations, fields, and <b>I</b> Dentifiers. Arrows indicate children, dotted arrows represent semantic annotations. . . . .	25
4.2	Schematic overview of the analysis performed by the language server . . . . .	26
4.3	Diagnostics in Sublime Text 3 . . . . .	30
4.4	The hover feature in VSCode . . . . .	32
4.5	Example of the rename feature in Eclipse . . . . .	33
4.6	Formatting in IntelliJ IDEA . . . . .	34
4.7	Screenshot showing the completion feature being used for a SKiL specification in VSCode . . . . .	35





## List of Tables

2.1	Feature matrix comparing different language client implementations currently available in IDEs. A ticked cell indicates the language client implementation possessing the respective capability or feature. . . . .	12
5.1	Benchmark results for various operations of the language server . . . . .	43



## List of Listings

3.1	Example of a specification file with a head comment and an include . . . . .	15
3.2	Example code for a user type inheriting from another user type and an interface .	16
3.3	Simple interface example, A represents a user type . . . . .	16
3.4	Example code with an enumeration of colors . . . . .	16
3.5	Examples of type definitions. Here, range is a restriction, restricting the range of possible values to [1, 12] . . . . .	17
3.6	Example of various built-in types . . . . .	17
3.7	Example of a user type with a string field and a constant integer field . . . . .	18
3.8	Examples of renaming and retyping using a view field . . . . .	18
3.9	Example of a custom field targeting the java generator, excerpt taken from tests specifications used to test [Fel13] . . . . .	18
3.10	Example of a user type with type annotations and a field with type annotations . .	19
3.11	Example of using the range and related restrictions to limit the possible set of values for fields . . . . .	19
3.12	Example usage of the oneOf restriction . . . . .	20
4.1	Example of two different ranges that could be associated with a missing semicolon for the first field . . . . .	31



## List of Algorithms

4.1	Determine the AST node at a given position in a document . . . . .	25
4.2	Parse a user declaration. getCharacter returns the character at the current position and advance advances the current position by one. . . . .	27



## List of Abbreviations

**AST** abstract syntax tree. 24, 25, 26, 27, 28, 29, 30, 31, 34, 35, 41, 42, 47, 53

**IDE** integrated development environment. 3, 4, 7, 9, 10, 11, 12, 13, 23, 30, 36, 39, 43, 45, 46, 49

**JSON** JavaScript Object Notation. 10, 55

**JSON-RPC** JSON remote procedure call. 10

**JVM** Java virtual machine. 23, 42

**LSP** Language Server Protocol. 3, 4, 7, 8, 9, 11, 12, 13, 23, 24, 28, 33, 34, 37, 39, 40, 45, 46, 47

**SKiIL** Serialization Killer Language. 3, 4, 7, 8, 11, 13, 15, 17, 20, 21, 23, 24, 27, 28, 29, 33, 34, 35, 36, 37, 39, 40, 42, 43, 44, 45, 46, 47

**URI** Uniform Resource Identifier. 23, 24

**VSCoDe** Visual Studio Code. 12, 32, 35, 47





## Bibliography

- [Con17] Consoli. Iro. 2017. URL: <https://eeyo.io/iro/> (cit. on p. 13).
- [Ecl16] Eclipse. Eclipse LSP4J. 2016. URL: <https://github.com/eclipse/lsp4j> (cit. on p. 23).
- [Fel13] T. Felden. SKiLL. 2013. URL: <https://github.com/skill-lang/skill> (cit. on pp. 18, 21, 36, 43).
- [Fel17] T. Felden. The SKiLL Language V1.0. Deutsch. Technischer Bericht Informatik 2017/01. Universität Stuttgart, Institut für Softwaretechnologie, Programmiersprachen und Übersetzerbau: Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Germany, 2017-01, p. 64. URL: [http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL\\_view.pl?id=TR-2017-01&engl=](http://www2.informatik.uni-stuttgart.de/cgi-bin/NCSTRL/NCSTRL_view.pl?id=TR-2017-01&engl=) (cit. on pp. 7, 15, 20, 21, 27, 33).
- [Hee16] D. van Heesch. Doxygen. 2016. URL: <http://www.stack.nl/~dimitri/doxygen/> (cit. on p. 15).
- [ISO11] ISO/IEC. Information technology – Programming languages – C. Standard. Geneva, CH: International Organization for Standardization, 2011-12 (cit. on p. 21).
- [jac13] jacoco. JaCoCo - Java Code Coverage Library. 2013. URL: <https://github.com/jacoco/jacoco> (cit. on p. 40).
- [KS05] R. B. Kline, A. Seffah. “Evaluation of integrated software development environments: Challenges and results from three empirical studies”. In: 63.6 (2005), pp. 607–627 (cit. on p. 7).
- [LLVa] LLVM. ClangFormat. URL: <https://clang.llvm.org/docs/ClangFormat.html> (cit. on p. 37).
- [LLVb] LLVM. libFuzzer – a library for coverage-guided fuzz testing. URL: <https://llvm.org/docs/LibFuzzer.html> (cit. on p. 42).
- [Mac] MacroMates. TextMate Manual - Language Grammars. URL: [https://macromates.com/manual/en/language\\_grammars](https://macromates.com/manual/en/language_grammars) (cit. on p. 13).
- [Mic16a] Microsoft. Language Server Protocol Specification. 2016. URL: <https://microsoft.github.io/language-server-protocol/specification> (cit. on pp. 7, 9).
- [Mic16b] Microsoft. Language Servers. 2016. URL: <https://microsoft.github.io/language-server-protocol/implementors/servers/> (cit. on p. 11).
- [Mic16c] Microsoft. LSP Inspector. 2016. URL: <https://microsoft.github.io/language-server-protocol/inspector/> (cit. on p. 9).
- [Mic16d] Microsoft. LSP Overview. 2016. URL: <https://microsoft.github.io/language-server-protocol/overview> (cit. on p. 9).

- [Mic16e] Microsoft. SDKs for the LSP. 2016. URL: <https://microsoft.github.io/language-server-protocol/implementors/sdks/> (cit. on p. 23).
- [Mic16f] Microsoft. Tools supporting the LSP. 2016. URL: <https://microsoft.github.io/language-server-protocol/implementors/tools/> (cit. on p. 12).
- [Mic17] S. Microsystems. JavaCC – The Java Parser Generator. 2017. URL: <https://javacc.org/> (cit. on p. 27).
- [Ora14] Oracle. Javadoc Technology. 2014. URL: <https://docs.oracle.com/javase/8/docs/technotes/guides/javadoc/index.html> (cit. on p. 15).
- [Par17] T. Parr. ANTLR. 2017. URL: <http://www.antlr.org/> (cit. on p. 27).
- [Sel17] W. Selwood. Tribble. 2017. URL: <https://github.com/SatelliteApplicationsCatapult/tribble> (cit. on p. 42).
- [Sou16a] Sourcegraph. Files extensions to LSP. 2016. URL: <https://github.com/sourcegraph/language-server-protocol/blob/master/extension-files.md> (cit. on p. 23).
- [Sou16b] Sourcegraph. Langserver.org. 2016. URL: <https://langserver.org/> (cit. on p. 12).
- [Stu15] U. Stuttgart. skilled. 2015. URL: <https://github.com/skill-lang/skilled> (cit. on pp. 7, 45).
- [TM05] R. F. T. Berners-Lee, L. Masinter. Uniform Resource Identifier (URI): Generic Syntax. RFC 3986. RFC Editor, 2005-01. URL: <https://www.rfc-editor.org/rfc/rfc3986.txt> (cit. on p. 23).
- [Zal13] M. Zalewski. american fuzzy lop. 2013. URL: <http://lcamtuf.coredump.cx/afl/> (cit. on p. 42).

### **Declaration**

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature